GAP - Reference Manual

Release 4.7.9, 29-Nov-2015

The GAP Group

The GAP Group Email: support@gap-system.org
Homepage: http://www.gap-system.org

Copyright

Copyright © (1987-2015) for the core part of the GAP system by the GAP Group.

Most parts of this distribution, including the core part of the GAP system are distributed under the terms of the GNU General Public License, see `http://www.gnu.org/licenses/gpl.html` or the file `GPL` in the `etc` directory of the GAP installation.

More detailed information about copyright and licenses of parts of this distribution can be found in Section 1.4 of this manual.

GAP is developed over a long time and has many authors and contributors. More detailed information can be found in Section 1.2 of this manual.

Contents

1 Preface **23**
- 1.1 The GAP System . . . 23
- 1.2 Authors and Maintainers . . . 25
- 1.3 Acknowledgements . . . 25
- 1.4 Copyright and License . . . 26
- 1.5 Further Information about GAP . . . 27

2 The Help System **28**
- 2.1 Invoking the Help . . . 28
- 2.2 Browsing through the Sections . . . 28
- 2.3 Changing the Help Viewer . . . 29
- 2.4 The Pager Command . . . 31

3 Running GAP **33**
- 3.1 Command Line Options . . . 33
- 3.2 The gap.ini and gaprc files . . . 38
- 3.3 Saving and Loading a Workspace . . . 41
- 3.4 Testing for the System Architecture . . . 42
- 3.5 Global Values that Control the GAP Session . . . 42
- 3.6 Coloring the Prompt and Input . . . 43

4 The Programming Language **45**
- 4.1 Language Overview . . . 45
- 4.2 Lexical Structure . . . 46
- 4.3 Symbols . . . 46
- 4.4 Whitespaces . . . 47
- 4.5 Keywords . . . 48
- 4.6 Identifiers . . . 48
- 4.7 Expressions . . . 49
- 4.8 Variables . . . 50
- 4.9 More About Global Variables . . . 51
- 4.10 Namespaces for GAP packages . . . 54
- 4.11 Function Calls . . . 54
- 4.12 Comparisons . . . 56
- 4.13 Arithmetic Operators . . . 57
- 4.14 Statements . . . 58

4.15 Assignments . . . 59
4.16 Procedure Calls . . . 60
4.17 If . . . 60
4.18 While . . . 61
4.19 Repeat . . . 62
4.20 For . . . 63
4.21 Break . . . 64
4.22 Continue . . . 65
4.23 Function . . . 65
4.24 Return (With or without Value) . . . 68

5 Functions **70**
5.1 Information about a function . . . 70
5.2 Calling a function with a list argument that is interpreted as several arguments . . . 72
5.3 Functions that do nothing . . . 73
5.4 Function Types . . . 74
5.5 Naming Conventions . . . 75

6 Main Loop and Break Loop **77**
6.1 Main Loop . . . 77
6.2 Special Rules for Input Lines . . . 79
6.3 View and Print . . . 79
6.4 Break Loops . . . 83
6.5 Variable Access in a Break Loop . . . 87
6.6 Error and ErrorCount . . . 89
6.7 Leaving GAP . . . 90
6.8 Line Editing . . . 90
6.9 Editing using the `readline` library . . . 93
6.10 Editing Files . . . 96
6.11 Editor Support . . . 96
6.12 Changing the Screen Size . . . 97
6.13 Teaching Mode . . . 97

7 Debugging and Profiling Facilities **99**
7.1 Recovery from NoMethodFound-Errors . . . 99
7.2 Inspecting Applicable Methods . . . 101
7.3 Tracing Methods . . . 101
7.4 Info Functions . . . 103
7.5 Assertions . . . 105
7.6 Timing . . . 106
7.7 Profiling . . . 107
7.8 Information about the version used . . . 112
7.9 Test Files . . . 113
7.10 Debugging Recursion . . . 117
7.11 Global Memory Information . . . 118

8 Options Stack **121**
8.1 Functions Dealing with the Options Stack . . . 121
8.2 Options Stack – an Example . . . 123

9 Files and Filenames **124**
9.1 Portability . . . 124
9.2 GAP Root Directories . . . 124
9.3 Directories . . . 125
9.4 File Names . . . 127
9.5 Special Filenames . . . 128
9.6 File Access . . . 128
9.7 File Operations . . . 129

10 Streams **133**
10.1 Categories for Streams and the StreamsFamily . . . 133
10.2 Operations applicable to All Streams . . . 135
10.3 Operations for Input Streams . . . 135
10.4 Operations for Output Streams . . . 138
10.5 File Streams . . . 141
10.6 User Streams . . . 142
10.7 String Streams . . . 143
10.8 Input-Output Streams . . . 143
10.9 Dummy Streams . . . 145
10.10 Handling of Streams in the Background . . . 145
10.11 Comma separated files . . . 146

11 Processes **148**
11.1 Process and Exec . . . 148

12 Objects and Elements **151**
12.1 Objects . . . 151
12.2 Elements as equivalence classes . . . 151
12.3 Sets . . . 152
12.4 Domains . . . 152
12.5 Identical Objects . . . 152
12.6 Mutability and Copyability . . . 154
12.7 Duplication of Objects . . . 156
12.8 Other Operations Applicable to any Object . . . 157

13 Types of Objects **159**
13.1 Families . . . 159
13.2 Filters . . . 160
13.3 Categories . . . 162
13.4 Representation . . . 164
13.5 Attributes . . . 165
13.6 Setter and Tester for Attributes . . . 166
13.7 Properties . . . 168

13.8 Other Filters . . . 170
13.9 Types . . . 170

14 Integers **172**
14.1 Integers: Global Variables . . . 172
14.2 Elementary Operations for Integers . . . 173
14.3 Quotients and Remainders . . . 176
14.4 Prime Integers and Factorization . . . 179
14.5 Residue Class Rings . . . 184
14.6 Check Digits . . . 186
14.7 Random Sources . . . 187

15 Number Theory **190**
15.1 InfoNumtheor (Info Class) . . . 190
15.2 Prime Residues . . . 190
15.3 Primitive Roots and Discrete Logarithms . . . 192
15.4 Roots Modulo Integers . . . 193
15.5 Multiplicative Arithmetic Functions . . . 196
15.6 Continued Fractions . . . 197
15.7 Miscellaneous . . . 198

16 Combinatorics **200**
16.1 Combinatorial Numbers . . . 200
16.2 Combinations, Arrangements and Tuples . . . 203
16.3 Fibonacci and Lucas Sequences . . . 212
16.4 Permanent of a Matrix . . . 213

17 Rational Numbers **215**
17.1 Rationals: Global Variables . . . 215
17.2 Elementary Operations for Rationals . . . 216

18 Cyclotomic Numbers **218**
18.1 Operations for Cyclotomics . . . 218
18.2 Infinity and negative Infinity . . . 223
18.3 Comparisons of Cyclotomics . . . 224
18.4 ATLAS Irrationalities . . . 224
18.5 Galois Conjugacy of Cyclotomics . . . 228
18.6 Internally Represented Cyclotomics . . . 230

19 Floats **233**
19.1 A sample run . . . 233
19.2 Methods . . . 234
19.3 High-precision-specific methods . . . 239
19.4 Complex arithmetic . . . 239
19.5 Interval-specific methods . . . 239

20 Booleans **240**
20.1 IsBool (Filter) . . . 240
20.2 Fail (Variable) . . . 240
20.3 Comparisons of Booleans . . . 241
20.4 Operations for Booleans . . . 241

21 Lists **244**
21.1 List Categories . . . 244
21.2 Basic Operations for Lists . . . 246
21.3 List Elements . . . 247
21.4 List Assignment . . . 248
21.5 IsBound and Unbind for Lists . . . 252
21.6 Identical Lists . . . 252
21.7 Duplication of Lists . . . 254
21.8 Membership Test for Lists . . . 255
21.9 Enlarging Internally Represented Lists . . . 256
21.10 Comparisons of Lists . . . 257
21.11 Arithmetic for Lists . . . 258
21.12 Filters Controlling the Arithmetic Behaviour of Lists . . . 258
21.13 Additive Arithmetic for Lists . . . 260
21.14 Multiplicative Arithmetic for Lists . . . 262
21.15 Mutability Status and List Arithmetic . . . 265
21.16 Finding Positions in Lists . . . 266
21.17 Properties and Attributes for Lists . . . 270
21.18 Sorting Lists . . . 272
21.19 Sorted Lists and Sets . . . 274
21.20 Operations for Lists . . . 277
21.21 Advanced List Manipulations . . . 287
21.22 Ranges . . . 289
21.23 Enumerators . . . 291

22 Boolean Lists **293**
22.1 IsBlist (Filter) . . . 293
22.2 Boolean Lists Representing Subsets . . . 294
22.3 Set Operations via Boolean Lists . . . 295
22.4 Function that Modify Boolean Lists . . . 296
22.5 More about Boolean Lists . . . 297

23 Row Vectors **299**
23.1 IsRowVector (Filter) . . . 299
23.2 Operators for Row Vectors . . . 300
23.3 Row Vectors over Finite Fields . . . 301
23.4 Coefficient List Arithmetic . . . 303
23.5 Shifting and Trimming Coefficient Lists . . . 304
23.6 Functions for Coding Theory . . . 305
23.7 Vectors as coefficients of polynomials . . . 306

24 Matrices **309**
24.1 InfoMatrix (Info Class) . . . 309
24.2 Categories of Matrices . . . 309
24.3 Operators for Matrices . . . 310
24.4 Properties and Attributes of Matrices . . . 313
24.5 Matrix Constructions . . . 315
24.6 Random Matrices . . . 317
24.7 Matrices Representing Linear Equations and the Gaussian Algorithm . . . 318
24.8 Eigenvectors and eigenvalues . . . 320
24.9 Elementary Divisors . . . 321
24.10 Echelonized Matrices . . . 323
24.11 Matrices as Basis of a Row Space . . . 325
24.12 Triangular Matrices . . . 326
24.13 Matrices as Linear Mappings . . . 327
24.14 Matrices over Finite Fields . . . 329
24.15 Inverse and Nullspace of an Integer Matrix Modulo an Ideal . . . 331
24.16 Special Multiplication Algorithms for Matrices over GF(2) . . . 332
24.17 Block Matrices . . . 333

25 Integral matrices and lattices **334**
25.1 Linear equations over the integers and Integral Matrices . . . 334
25.2 Normal Forms over the Integers . . . 336
25.3 Determinant of an integer matrix . . . 339
25.4 Decompositions . . . 339
25.5 Lattice Reduction . . . 341
25.6 Orthogonal Embeddings . . . 343

26 Vector and matrix objects **345**
26.1 Fundamental ideas and rules . . . 345
26.2 Categories of vectors and matrices . . . 346
26.3 Constructing vector and matrix objects . . . 346
26.4 Operations for row vector objects . . . 346
26.5 Operations for row list matrix objects . . . 346
26.6 Operations for flat matrix objects . . . 346

27 Strings and Characters **347**
27.1 IsChar and IsString . . . 347
27.2 Special Characters . . . 349
27.3 Internally Represented Strings . . . 351
27.4 Recognizing Characters . . . 352
27.5 Comparisons of Strings . . . 353
27.6 Operations to Produce or Manipulate Strings . . . 354
27.7 Character Conversion . . . 359
27.8 Operations to Evaluate Strings . . . 360
27.9 Calendar Arithmetic . . . 362
27.10 Obtaining LaTeX Representations of Objects . . . 364

28 Dictionaries and General Hash Tables **366**
28.1 Using Dictionaries . . . 366
28.2 Dictionaries . . . 368
28.3 Dictionaries via Binary Lists . . . 368
28.4 General Hash Tables . . . 369
28.5 Hash keys . . . 370
28.6 Dense hash tables . . . 370
28.7 Sparse hash tables . . . 370

29 Records **372**
29.1 IsRecord and RecNames . . . 372
29.2 Accessing Record Elements . . . 373
29.3 Record Assignment . . . 374
29.4 Identical Records . . . 374
29.5 Comparisons of Records . . . 376
29.6 IsBound and Unbind for Records . . . 377
29.7 Record Access Operations . . . 378

30 Collections **379**
30.1 IsCollection (Filter) . . . 379
30.2 Collection Families . . . 379
30.3 Lists and Collections . . . 380
30.4 Attributes and Properties for Collections . . . 386
30.5 Operations for Collections . . . 388
30.6 Membership Test for Collections . . . 390
30.7 Random Elements . . . 391
30.8 Iterators . . . 392

31 Domains and their Elements **396**
31.1 Operational Structure of Domains . . . 396
31.2 Equality and Comparison of Domains . . . 397
31.3 Constructing Domains . . . 398
31.4 Changing the Structure . . . 399
31.5 Changing the Representation . . . 399
31.6 Domain Categories . . . 400
31.7 Parents . . . 401
31.8 Constructing Subdomains . . . 402
31.9 Operations for Domains . . . 402
31.10 Attributes and Properties of Elements . . . 403
31.11 Comparison Operations for Elements . . . 407
31.12 Arithmetic Operations for Elements . . . 408
31.13 Relations Between Domains . . . 409
31.14 Useful Categories of Elements . . . 412
31.15 Useful Categories for all Elements of a Family . . . 416

32 Mappings **419**
32.1 IsDirectProductElement (Filter) . . . 419
32.2 Creating Mappings . . . 420
32.3 Properties and Attributes of (General) Mappings . . . 422
32.4 Images under Mappings . . . 424
32.5 Preimages under Mappings . . . 426
32.6 Arithmetic Operations for General Mappings . . . 428
32.7 Mappings which are Compatible with Algebraic Structures . . . 428
32.8 Magma Homomorphisms . . . 429
32.9 Mappings that Respect Multiplication . . . 429
32.10 Mappings that Respect Addition . . . 430
32.11 Linear Mappings . . . 431
32.12 Ring Homomorphisms . . . 432
32.13 General Mappings . . . 433
32.14 Technical Matters Concerning General Mappings . . . 434

33 Relations **437**
33.1 General Binary Relations . . . 437
33.2 Properties and Attributes of Binary Relations . . . 438
33.3 Binary Relations on Points . . . 440
33.4 Closure Operations and Other Constructors . . . 440
33.5 Equivalence Relations . . . 442
33.6 Attributes of and Operations on Equivalence Relations . . . 443
33.7 Equivalence Classes . . . 443

34 Orderings **445**
34.1 IsOrdering (Filter) . . . 445
34.2 Building new orderings . . . 445
34.3 Properties and basic functionality . . . 446
34.4 Orderings on families of associative words . . . 447

35 Magmas **452**
35.1 Magma Categories . . . 452
35.2 Magma Generation . . . 453
35.3 Magmas Defined by Multiplication Tables . . . 456
35.4 Attributes and Properties for Magmas . . . 458

36 Words **462**
36.1 Categories of Words and Nonassociative Words . . . 462
36.2 Comparison of Words . . . 464
36.3 Operations for Words . . . 465
36.4 Free Magmas . . . 466
36.5 External Representation for Nonassociative Words . . . 467

37 Associative Words **468**
37.1 Categories of Associative Words . . . 468
37.2 Free Groups, Monoids and Semigroups . . . 469
37.3 Comparison of Associative Words . . . 470
37.4 Operations for Associative Words . . . 471
37.5 Operations for Associative Words by their Syllables . . . 474
37.6 Representations for Associative Words . . . 475
37.7 The External Representation for Associative Words . . . 477
37.8 Straight Line Programs . . . 477
37.9 Straight Line Program Elements . . . 483

38 Rewriting Systems **485**
38.1 Operations on rewriting systems . . . 485
38.2 Operations on elements of the algebra . . . 487
38.3 Properties of rewriting systems . . . 488
38.4 Rewriting in Groups and Monoids . . . 488
38.5 Developing rewriting systems . . . 489

39 Groups **491**
39.1 Group Elements . . . 491
39.2 Creating Groups . . . 492
39.3 Subgroups . . . 494
39.4 Closures of (Sub)groups . . . 497
39.5 Expressing Group Elements as Words in Generators . . . 498
39.6 Structure Descriptions . . . 500
39.7 Cosets . . . 502
39.8 Transversals . . . 504
39.9 Double Cosets . . . 504
39.10 Conjugacy Classes . . . 506
39.11 Normal Structure . . . 509
39.12 Specific and Parametrized Subgroups . . . 511
39.13 Sylow Subgroups and Hall Subgroups . . . 514
39.14 Subgroups characterized by prime powers . . . 516
39.15 Group Properties . . . 516
39.16 Numerical Group Attributes . . . 523
39.17 Subgroup Series . . . 524
39.18 Factor Groups . . . 529
39.19 Sets of Subgroups . . . 530
39.20 Subgroup Lattice . . . 533
39.21 Specific Methods for Subgroup Lattice Computations . . . 536
39.22 Special Generating Sets . . . 539
39.23 1-Cohomology . . . 541
39.24 Schur Covers and Multipliers . . . 543
39.25 Tests for the Availability of Methods . . . 547

40 Group Homomorphisms **549**
40.1 Creating Group Homomorphisms . . . 549
40.2 Operations for Group Homomorphisms . . . 552
40.3 Efficiency of Homomorphisms . . . 553
40.4 Homomorphism for very large groups . . . 554
40.5 Nice Monomorphisms . . . 555
40.6 Group Automorphisms . . . 556
40.7 Groups of Automorphisms . . . 558
40.8 Calculating with Group Automorphisms . . . 559
40.9 Searching for Homomorphisms . . . 560
40.10 Representations for Group Homomorphisms . . . 563

41 Group Actions **566**
41.1 About Group Actions . . . 566
41.2 Basic Actions . . . 567
41.3 Action on canonical representatives . . . 571
41.4 Orbits . . . 571
41.5 Stabilizers . . . 573
41.6 Elements with Prescribed Images . . . 575
41.7 The Permutation Image of an Action . . . 575
41.8 Action of a group on itself . . . 577
41.9 Permutations Induced by Elements and Cycles . . . 578
41.10 Tests for Actions . . . 580
41.11 Block Systems . . . 582
41.12 External Sets . . . 583

42 Permutations **589**
42.1 IsPerm (Filter) . . . 589
42.2 Comparison of Permutations . . . 590
42.3 Moved Points of Permutations . . . 591
42.4 Sign and Cycle Structure . . . 592
42.5 Creating Permutations . . . 593

43 Permutation Groups **595**
43.1 IsPermGroup (Filter) . . . 595
43.2 The Natural Action . . . 595
43.3 Computing a Permutation Representation . . . 596
43.4 Symmetric and Alternating Groups . . . 597
43.5 Primitive Groups . . . 598
43.6 Stabilizer Chains . . . 599
43.7 Randomized Methods for Permutation Groups . . . 600
43.8 Construction of Stabilizer Chains . . . 602
43.9 Stabilizer Chain Records . . . 605
43.10 Operations for Stabilizer Chains . . . 606
43.11 Low Level Routines to Modify and Create Stabilizer Chains . . . 608
43.12 Backtrack . . . 610
43.13 Working with large degree permutation groups . . . 611

44 Matrix Groups **613**
44.1 IsMatrixGroup (Filter) . . . 613
44.2 Attributes and Properties for Matrix Groups . . . 614
44.3 Actions of Matrix Groups . . . 615
44.4 GL and SL . . . 615
44.5 Invariant Forms . . . 617
44.6 Matrix Groups in Characteristic 0 . . . 618
44.7 Acting OnRight and OnLeft . . . 621

45 Polycyclic Groups **622**
45.1 Polycyclic Generating Systems . . . 622
45.2 Computing a Pcgs . . . 623
45.3 Defining a Pcgs Yourself . . . 624
45.4 Elementary Operations for a Pcgs . . . 624
45.5 Elementary Operations for a Pcgs and an Element . . . 625
45.6 Exponents of Special Products . . . 627
45.7 Subgroups of Polycyclic Groups - Induced Pcgs . . . 628
45.8 Subgroups of Polycyclic Groups – Canonical Pcgs . . . 630
45.9 Factor Groups of Polycyclic Groups – Modulo Pcgs . . . 631
45.10 Factor Groups of Polycyclic Groups in their Own Representation . . . 633
45.11 Pcgs and Normal Series . . . 634
45.12 Sum and Intersection of Pcgs . . . 638
45.13 Special Pcgs . . . 639
45.14 Action on Subfactors Defined by a Pcgs . . . 641
45.15 Orbit Stabilizer Methods for Polycyclic Groups . . . 643
45.16 Operations which have Special Methods for Groups with Pcgs . . . 643
45.17 Conjugacy Classes in Solvable Groups . . . 643

46 Pc Groups **645**
46.1 The family pcgs . . . 646
46.2 Elements of pc groups . . . 647
46.3 Pc groups versus fp groups . . . 647
46.4 Constructing Pc Groups . . . 648
46.5 Computing Pc Groups . . . 651
46.6 Saving a Pc Group . . . 652
46.7 Operations for Pc Groups . . . 652
46.8 2-Cohomology and Extensions . . . 652
46.9 Coding a Pc Presentation . . . 656
46.10 Random Isomorphism Testing . . . 657

47 Finitely Presented Groups **658**
47.1 IsSubgroupFpGroup and IsFpGroup . . . 659
47.2 Creating Finitely Presented Groups . . . 660
47.3 Comparison of Elements of Finitely Presented Groups . . . 661
47.4 Preimages in the Free Group . . . 662
47.5 Operations for Finitely Presented Groups . . . 664
47.6 Coset Tables and Coset Enumeration . . . 664

47.7 Standardization of coset tables . . . 668
47.8 Coset tables for subgroups in the whole group . . . 669
47.9 Augmented Coset Tables and Rewriting . . . 670
47.10 Low Index Subgroups . . . 671
47.11 Converting Groups to Finitely Presented Groups . . . 672
47.12 New Presentations and Presentations for Subgroups . . . 675
47.13 Preimages under Homomorphisms from an FpGroup . . . 676
47.14 Quotient Methods . . . 677
47.15 Abelian Invariants for Subgroups . . . 680
47.16 Testing Finiteness of Finitely Presented Groups . . . 682

48 Presentations and Tietze Transformations 684
48.1 Creating Presentations . . . 684
48.2 Subgroup Presentations . . . 687
48.3 Relators in a Presentation . . . 691
48.4 Printing Presentations . . . 692
48.5 Changing Presentations . . . 694
48.6 Tietze Transformations . . . 695
48.7 Elementary Tietze Transformations . . . 698
48.8 Tietze Transformations that introduce new Generators . . . 700
48.9 Tracing generator images through Tietze transformations . . . 704
48.10 The Decoding Tree Procedure . . . 706
48.11 Tietze Options . . . 709

49 Group Products 712
49.1 Direct Products . . . 712
49.2 Semidirect Products . . . 713
49.3 Subdirect Products . . . 715
49.4 Wreath Products . . . 715
49.5 Free Products . . . 717
49.6 Embeddings and Projections for Group Products . . . 718

50 Group Libraries 719
50.1 Basic Groups . . . 719
50.2 Classical Groups . . . 723
50.3 Conjugacy Classes in Classical Groups . . . 729
50.4 Constructors for Basic Groups . . . 730
50.5 Selection Functions . . . 731
50.6 Transitive Permutation Groups . . . 732
50.7 Small Groups . . . 733
50.8 Finite Perfect Groups . . . 737
50.9 Primitive Permutation Groups . . . 743
50.10 Index numbers of primitive groups . . . 745
50.11 Irreducible Solvable Matrix Groups . . . 746
50.12 Irreducible Maximal Finite Integral Matrix Groups . . . 747

51 Semigroups and Monoids **757**
51.1 Semigroups . . . 757
51.2 Monoids . . . 761
51.3 Inverse semigroups and monoids . . . 763
51.4 Properties of Semigroups . . . 765
51.5 Ideals of semigroups . . . 767
51.6 Congruences for semigroups . . . 767
51.7 Quotients . . . 768
51.8 Green's Relations . . . 768
51.9 Rees Matrix Semigroups . . . 771

52 Finitely Presented Semigroups and Monoids **779**
52.1 IsSubsemigroupFpSemigroup (Filter) . . . 781
52.2 Creating Finitely Presented Semigroups . . . 782
52.3 Comparison of Elements of Finitely Presented Semigroups . . . 783
52.4 Preimages in the Free Semigroup . . . 783
52.5 Finitely presented monoids . . . 785
52.6 Rewriting Systems and the Knuth-Bendix Procedure . . . 785
52.7 Todd-Coxeter Procedure . . . 788

53 Transformations **789**
53.1 The family and categories of transformations . . . 790
53.2 Creating transformations . . . 791
53.3 Changing the representation of a transformation . . . 794
53.4 Operators for transformations . . . 796
53.5 Attributes for transformations . . . 798
53.6 Displaying transformations . . . 807
53.7 Semigroups of transformations . . . 808

54 Partial permutations **812**
54.1 The family and categories of partial permutations . . . 814
54.2 Creating partial permutations . . . 814
54.3 Attributes for partial permutations . . . 818
54.4 Changing the representation of a partial permutation . . . 826
54.5 Operators and operations for partial permutations . . . 828
54.6 Displaying partial permutations . . . 831
54.7 Semigroups and inverse semigroups of partial permutations . . . 833

55 Additive Magmas **838**
55.1 (Near-)Additive Magma Categories . . . 838
55.2 (Near-)Additive Magma Generation . . . 840
55.3 Attributes and Properties for (Near-)Additive Magmas . . . 842
55.4 Operations for (Near-)Additive Magmas . . . 843

56 Rings **844**
56.1 Generating Rings . . . 844
56.2 Ideals in Rings . . . 847
56.3 Rings With One . . . 850
56.4 Properties of Rings . . . 852
56.5 Units and Factorizations . . . 853
56.6 Euclidean Rings . . . 856
56.7 Gcd and Lcm . . . 857
56.8 Homomorphisms of Rings . . . 860

57 Modules **862**
57.1 Generating modules . . . 862
57.2 Submodules . . . 864
57.3 Free Modules . . . 865

58 Fields and Division Rings **868**
58.1 Generating Fields . . . 868
58.2 Subfields of Fields . . . 870
58.3 Galois Action . . . 872

59 Finite Fields **876**
59.1 Finite Field Elements . . . 876
59.2 Operations for Finite Field Elements . . . 878
59.3 Creating Finite Fields . . . 881
59.4 Frobenius Automorphisms . . . 882
59.5 Conway Polynomials . . . 883
59.6 Printing, Viewing and Displaying Finite Field Elements . . . 884

60 Abelian Number Fields **886**
60.1 Construction of Abelian Number Fields . . . 886
60.2 Operations for Abelian Number Fields . . . 888
60.3 Integral Bases of Abelian Number Fields . . . 889
60.4 Galois Groups of Abelian Number Fields . . . 891
60.5 Gaussians . . . 893

61 Vector Spaces **894**
61.1 IsLeftVectorSpace (Filter) . . . 894
61.2 Constructing Vector Spaces . . . 894
61.3 Operations and Attributes for Vector Spaces . . . 896
61.4 Domains of Subspaces of Vector Spaces . . . 896
61.5 Bases of Vector Spaces . . . 897
61.6 Operations for Vector Space Bases . . . 899
61.7 Operations for Special Kinds of Bases . . . 901
61.8 Mutable Bases . . . 902
61.9 Row and Matrix Spaces . . . 904
61.10 Vector Space Homomorphisms . . . 908
61.11 Vector Spaces Handled By Nice Bases . . . 911

61.12 How to Implement New Kinds of Vector Spaces 913

62 Algebras **915**
62.1 InfoAlgebra (Info Class) 915
62.2 Constructing Algebras by Generators 915
62.3 Constructing Algebras as Free Algebras 916
62.4 Constructing Algebras by Structure Constants 917
62.5 Some Special Algebras 920
62.6 Subalgebras 922
62.7 Ideals 923
62.8 Categories and Properties of Algebras 924
62.9 Attributes and Operations for Algebras 926
62.10 Homomorphisms of Algebras 934
62.11 Representations of Algebras 939

63 Finitely Presented Algebras **949**

64 Lie Algebras **950**
64.1 Lie Objects 950
64.2 Constructing Lie algebras 952
64.3 Distinguished Subalgebras 955
64.4 Series of Ideals 957
64.5 Properties of a Lie Algebra 958
64.6 Semisimple Lie Algebras and Root Systems 959
64.7 Semisimple Lie Algebras and Weyl Groups of Root Systems 962
64.8 Restricted Lie algebras 965
64.9 The Adjoint Representation 967
64.10 Universal Enveloping Algebras 969
64.11 Finitely Presented Lie Algebras 969
64.12 Modules over Lie Algebras and Their Cohomology 971
64.13 Modules over Semisimple Lie Algebras 974
64.14 Admissible Lattices in UEA 975
64.15 Tensor Products and Exterior and Symmetric Powers 978

65 Magma Rings **980**
65.1 Free Magma Rings 981
65.2 Elements of Free Magma Rings 982
65.3 Natural Embeddings related to Magma Rings 983
65.4 Magma Rings modulo Relations 984
65.5 Magma Rings modulo the Span of a Zero Element 985
65.6 Technical Details about the Implementation of Magma Rings 986

66 Polynomials and Rational Functions **987**
66.1 Indeterminates 987
66.2 Operations for Rational Functions 990
66.3 Comparison of Rational Functions 991
66.4 Properties and Attributes of Rational Functions 992

66.5 Univariate Polynomials . . . 995
66.6 Polynomials as Univariate Polynomials in one Indeterminate . . . 997
66.7 Multivariate Polynomials . . . 999
66.8 Minimal Polynomials . . . 1000
66.9 Cyclotomic Polynomials . . . 1000
66.10 Polynomial Factorization . . . 1000
66.11 Polynomials over the Rationals . . . 1001
66.12 Factorization of Polynomials over the Rationals . . . 1002
66.13 Laurent Polynomials . . . 1003
66.14 Univariate Rational Functions . . . 1004
66.15 Polynomial Rings and Function Fields . . . 1004
66.16 Univariate Polynomial Rings . . . 1007
66.17 Monomial Orderings . . . 1008
66.18 Groebner Bases . . . 1012
66.19 Rational Function Families . . . 1013
66.20 The Representations of Rational Functions . . . 1014
66.21 The Defining Attributes of Rational Functions . . . 1015
66.22 Creation of Rational Functions . . . 1017
66.23 Arithmetic for External Representations of Polynomials . . . 1018
66.24 Cancellation Tests for Rational Functions . . . 1019

67 Algebraic extensions of fields 1020
67.1 Creation of Algebraic Extensions . . . 1020
67.2 Elements in Algebraic Extensions . . . 1021

68 p-adic Numbers (preliminary) 1023
68.1 Pure p-adic Numbers . . . 1023
68.2 Extensions of the p-adic Numbers . . . 1024

69 The MeatAxe 1027
69.1 MeatAxe Modules . . . 1027
69.2 Module Constructions . . . 1028
69.3 Selecting a Different MeatAxe . . . 1028
69.4 Accessing a Module . . . 1028
69.5 Irreducibility Tests . . . 1029
69.6 Decomposition of modules . . . 1029
69.7 Finding Submodules . . . 1030
69.8 Induced Actions . . . 1032
69.9 Module Homomorphisms . . . 1033
69.10 Module Homomorphisms for irreducible modules . . . 1033
69.11 MeatAxe Functionality for Invariant Forms . . . 1034
69.12 The Smash MeatAxe . . . 1035
69.13 Smash MeatAxe Flags . . . 1037

70 Tables of Marks **1039**
70.1 More about Tables of Marks . . . 1039
70.2 Table of Marks Objects in GAP . . . 1040
70.3 Constructing Tables of Marks . . . 1041
70.4 Printing Tables of Marks . . . 1042
70.5 Sorting Tables of Marks . . . 1044
70.6 Technical Details about Tables of Marks . . . 1045
70.7 Attributes of Tables of Marks . . . 1046
70.8 Properties of Tables of Marks . . . 1051
70.9 Other Operations for Tables of Marks . . . 1051
70.10 Accessing Subgroups via Tables of Marks . . . 1056
70.11 The Interface between Tables of Marks and Character Tables . . . 1058
70.12 Generic Construction of Tables of Marks . . . 1060
70.13 The Library of Tables of Marks . . . 1061

71 Character Tables **1062**
71.1 Some Remarks about Character Theory in GAP . . . 1062
71.2 History of Character Theory Stuff in GAP . . . 1064
71.3 Creating Character Tables . . . 1065
71.4 Character Table Categories . . . 1068
71.5 Conventions for Character Tables . . . 1069
71.6 The Interface between Character Tables and Groups . . . 1070
71.7 Operators for Character Tables . . . 1073
71.8 Attributes and Properties for Groups and Character Tables . . . 1073
71.9 Attributes and Properties only for Character Tables . . . 1077
71.10 Normal Subgroups Represented by Lists of Class Positions . . . 1081
71.11 Operations Concerning Blocks . . . 1085
71.12 Other Operations for Character Tables . . . 1088
71.13 Printing Character Tables . . . 1092
71.14 Computing the Irreducible Characters of a Group . . . 1096
71.15 Representations Given by Modules . . . 1099
71.16 The Dixon-Schneider Algorithm . . . 1100
71.17 Advanced Methods for Dixon-Schneider Calculations . . . 1100
71.18 Components of a Dixon Record . . . 1102
71.19 An Example of Advanced Dixon-Schneider Calculations . . . 1103
71.20 Constructing Character Tables from Others . . . 1105
71.21 Sorted Character Tables . . . 1108
71.22 Automorphisms and Equivalence of Character Tables . . . 1111
71.23 Storing Normal Subgroup Information . . . 1113

72 Class Functions **1116**
72.1 Why Class Functions? . . . 1116
72.2 Basic Operations for Class Functions . . . 1119
72.3 Comparison of Class Functions . . . 1120
72.4 Arithmetic Operations for Class Functions . . . 1120
72.5 Printing Class Functions . . . 1124
72.6 Creating Class Functions from Values Lists . . . 1125

72.7 Creating Class Functions using Groups . . . 1126
72.8 Operations for Class Functions . . . 1127
72.9 Restricted and Induced Class Functions . . . 1134
72.10 Reducing Virtual Characters . . . 1136
72.11 Symmetrizations of Class Functions . . . 1143
72.12 Molien Series . . . 1146
72.13 Possible Permutation Characters . . . 1148
72.14 Computing Possible Permutation Characters . . . 1151
72.15 Operations for Brauer Characters . . . 1156
72.16 Domains Generated by Class Functions . . . 1157

73 Maps Concerning Character Tables 1158
73.1 Power Maps . . . 1158
73.2 Orbits on Sets of Possible Power Maps . . . 1162
73.3 Class Fusions between Character Tables . . . 1163
73.4 Orbits on Sets of Possible Class Fusions . . . 1169
73.5 Parametrized Maps . . . 1169
73.6 Subroutines for the Construction of Power Maps . . . 1178
73.7 Subroutines for the Construction of Class Fusions . . . 1182

74 Unknowns 1185
74.1 More about Unknowns . . . 1185

75 Monomiality Questions 1188
75.1 InfoMonomial (Info Class) . . . 1189
75.2 Character Degrees and Derived Length . . . 1189
75.3 Primitivity of Characters . . . 1190
75.4 Testing Monomiality . . . 1192
75.5 Minimal Nonmonomial Groups . . . 1196

76 Using GAP Packages 1197
76.1 Installing a GAP Package . . . 1197
76.2 Loading a GAP Package . . . 1198
76.3 Functions for GAP Packages . . . 1200

77 Replaced and Removed Command Names 1208
77.1 Group Actions – Name Changes . . . 1208
77.2 Package Interface – Obsolete Functions and Name Changes . . . 1209
77.3 Normal Forms of Integer Matrices – Name Changes . . . 1209
77.4 Miscellaneous Name Changes or Removed Names . . . 1209
77.5 The former .gaprc file . . . 1210

78 Method Selection 1211
78.1 Operations and Methods . . . 1211
78.2 Method Installation . . . 1211
78.3 Applicable Methods and Method Selection . . . 1212
78.4 Partial Methods . . . 1213
78.5 Redispatching . . . 1213

78.6 Immediate Methods 1214
78.7 Logical Implications 1215
78.8 Operations and Mathematical Terms 1215

79 Creating New Objects 1217
79.1 Creating Categories 1217
79.2 Creating Representations 1218
79.3 Creating Attributes and Properties 1218
79.4 Creating Other Filters 1219
79.5 Creating Operations 1220
79.6 Creating Constructors 1220
79.7 Creating Families 1220
79.8 Creating Types 1222
79.9 Creating Objects 1222
79.10 Component Objects 1223
79.11 Positional Objects 1224
79.12 Implementing New List Objects 1226
79.13 Example – Constructing Enumerators 1226
79.14 Example – Constructing Iterators 1229
79.15 Arithmetic Issues in the Implementation of New Kinds of Lists 1230
79.16 External Representation 1231
79.17 Mutability and Copying 1232
79.18 Global Variables in the Library 1234
79.19 Declaration and Implementation Part 1237

80 Examples of Extending the System 1239
80.1 Addition of a Method 1239
80.2 Extending the Range of Definition of an Existing Operation 1241
80.3 Enforcing Property Tests 1241
80.4 Adding a new Operation 1242
80.5 Adding a new Attribute 1242
80.6 Adding a new Representation 1243
80.7 Components versus Attributes 1245
80.8 Adding new Concepts 1245
80.9 Creating Own Arithmetic Objects 1248

81 An Example – Residue Class Rings 1251
81.1 A First Attempt to Implement Elements of Residue Class Rings 1251
81.2 Why Proceed in a Different Way? 1252
81.3 A Second Attempt to Implement Elements of Residue Class Rings 1253
81.4 Compatibility of Residue Class Rings with Prime Fields 1264
81.5 Further Improvements in Implementing Residue Class Rings 1270

82 An Example – Designing Arithmetic Operations 1272
82.1 New Arithmetic Operations vs. New Objects 1272
82.2 Designing new Multiplicative Objects 1273

83 Library Files **1279**
83.1 File Types 1279
83.2 Finding Implementations in the Library 1279
83.3 Undocumented Variables 1280

84 Interface to the GAP Help System **1282**
84.1 Installing and Removing a Help Book 1282
84.2 The manual.six File 1283
84.3 The Help Book Handler 1283
84.4 Introducing new Viewer for the Online Help 1285

85 Function-Operation-Attribute Triples **1286**
85.1 Key Dependent Operations 1286
85.2 In Parent Attributes 1287
85.3 Operation Functions 1288

86 Weak Pointers **1292**
86.1 Weak Pointer Objects 1292
86.2 Low Level Access Functions for Weak Pointer Objects 1293
86.3 Accessing Weak Pointer Objects as Lists 1294
86.4 Copying Weak Pointer Objects 1294
86.5 The GASMAN Interface for Weak Pointer Objects 1295

87 More about Stabilizer Chains **1296**
87.1 Generalized Conjugation Technique 1296
87.2 The General Backtrack Algorithm with Ordered Partitions 1297
87.3 Stabilizer Chains for Automorphisms Acting on Enumerators 1305

References **1318**

Index **1319**

Chapter 1

Preface

Welcome to GAP. This is one of three manuals documenting the core part of GAP, the other being the *GAP Tutorial* . and the document called *"GAP - Changes from Earlier Versions"* .

This preface serves not only to introduce "The GAP Reference Manual", but also as an introduction to the whole system.

GAP stands for *Groups, Algorithms and Programming*. The name was chosen to reflect the aim of the system, which is introduced in this reference manual. Since that choice, the system has become somewhat broader, and you will also find information about algorithms and programming for other algebraic structures, such as semigroups and algebras.

This manual, the *GAP reference manual* contains the official definitions of GAP functions. It should contain all the information needed to use GAP, and is not intended to be read cover-to-cover.

To get started a new user may first look at parts of the *GAP Tutorial* .

A lot of the functionality of the system and a number of contributed extensions are provided as "GAP packages" which are developed independently of the core part of GAP and can be loaded into a GAP session. Each package comes with a its own manual which is also available through the GAP help system.

This manual is divided into chapters, sections and subsections. Chapter 2 describes the *help system*, which provides access to all the manuals from a running GAP session. Chapter 3 gives technical advice for *running* GAP. Chapter 4 introduces the GAP language, and the next chapters deal with the *environment* provided by GAP for the user. These are followed by the main bulk of chapters which are devoted to the various mathematical structures that GAP can handle.

Subsequent sections of this preface explain the structure of the system and provide copyright and licensing information.

1.1 The GAP System

GAP is a *free*, *open* and *extensible* software package for computation in discrete abstract algebra. The terms "free" and "open" describe the conditions under which the system is distributed – in brief, it is *free of charge* (except possibly for the immediate costs of delivering it to you), you are *free to pass it on* within certain limits, and all of the workings of the system are *open for you to examine and change*. Details of these conditions can be found in Section (**Reference: Copyright and License**).

The system is "extensible" in that you can write your own programs in the GAP language, and use them in just the same way as the programs which form part of the system (the "library"). Indeed, we actively support the contribution, refereeing and distribution of extensions to the system, in the

form of "GAP packages". Further details of this can be found in chapter (**Reference: Using GAP Packages**), and on our website.

Development of GAP began at Lehrstuhl D für Mathematik, RWTH-Aachen, under the leadership of Joachim Neubüser in 1985. Version 2.4 was released in 1988 and version 3.1 in 1992. In 1997 coordination of GAP development, now very much an international effort, was transferred to St Andrews. A complete internal redesign and almost complete rewrite of the system was completed over the following years and version 4.1 was released in July 1999. A sign of the further internationalization of the project was the GAP 4.4 release in 2004, which has been coordinated from Colorado State University, Fort Collins.

More information on the motivation and development of GAP to date, can be found on our Web pages in a section entitled "Release history and Prefaces".

For those readers who have used an earlier version of GAP, an overview of the changes from GAP 4.4 and a brief summary of changes from earlier versions is given in a separate manual (**Changes: Changes between GAP 4.4 and GAP 4.5**).

The system that you are getting now consists of a "core system" and a number of packages. The core system consists of four main parts.

1. A *kernel*, written in C, which provides the user with

   - automatic dynamic storage management, which the user needn't bother about in his programming;
   - a set of time-critical basic functions, e.g. "arithmetic", operations for integers, finite fields, permutations and words, as well as natural operations for lists and records;
   - an interpreter for the GAP language, an untyped imperative programming language with functions as first class objects and some extra built-in data types such as permutations and finite field elements. The language supports a form of object-oriented programming, similar to that supported by languages like C++ and Java but with some important differences.
   - a small set of system functions allowing the GAP programmer to handle files and execute external programs in a uniform way, regardless of the particular operating system in use.
   - a set of programming tools for testing, debugging, and timing algorithms.
   - a "read-eval-view" style user interface.

2. A much larger *library of GAP functions* that implement algebraic and other algorithms. Since this is written entirely in the GAP language, the GAP language is both the main implementation language and the user language of the system. Therefore the user can as easily as the original programmers investigate and vary algorithms of the library and add new ones to it, first for own use and eventually for the benefit of all GAP users.

3. A *library of group theoretical data* which contains various libraries of groups, including the library of small groups (containing all groups of order at most 2000, except those of order 1024) and others. Large libraries of ordinary and Brauer character tables and Tables of Marks are included as packages.

4. The *documentation*. This is available as on-line help, as printable files in PDF format and as HTML for viewing with a Web browser.

Also included with the core system are some test files and a few small utilities which we hope you will find useful.

GAP *packages* are self-contained extensions to the core system. A package contains GAP code and its own documentation and may also contain data files or external programs to which the GAP code provides an interface. These packages may be loaded into GAP using the `LoadPackage` (**Reference: LoadPackage**) command, and both the package and its documentation are then available just as if they were parts of the core system. Some packages may be loaded automatically, when GAP is started, if they are present. Some packages, because they depend on external programs, may only be available on the operating systems where those programs are available (usually UNIX). You should note that, while the packages included with this release are the most recent versions ready for release at this time, new packages and new versions may be released at any time and can be easily installed in your copy of GAP.

With GAP there are two packages (the library of ordinary and Brauer character tables, and the library of tables of marks) which contain functionality developed from parts of the GAP core system. These have been moved into packages for ease of maintenance and to allow new versions to be released independently of new releases of the core system. The library of small groups should also be regarded as a package, although it does not currently use the standard package mechanism. Other packages contain functionality which has never been part of the core system, and may extend it substantially, implementing specific algorithms to enhance its capabilities, providing data libraries, interfaces to other computer algebra systems and data sources such as the electronic version of the Atlas of Finite Group Representations; therefore, installation and usage of packages is recommended.

Further details about GAP packages can be found in chapter (**Reference: Using GAP Packages**), and on the GAP website here: `http://www.gap-system.org/Packages/packages.html`.

1.2 Authors and Maintainers

GAP is the work of very many people, many of whom still maintain parts of the system. A complete list of authors, and an approximation to the current list of maintainers can be found on the GAP World Wide Web site at `http://www.gap-system.org/Contacts/People/authors.html` and `http://www.gap-system.org/Contacts/People/modules.html`. All GAP packages have their own authors and maintainers. It should however be noted that some packages provide interfaces between GAP and an external program, a copy of which is included for convenience, and that, in these cases, we do not claim that the package authors or maintainers wrote, or maintain, this external program. Similarly, the system and some packages include large data libraries that may have been computed by many people. We try to make clear in each case what credit is attributable to whom.

We have, for some time, operated a refereeing system for contributed packages, both to ensure the quality of the software we distribute, and to provide recognition for the authors. We now consider this to be a refereeing system for modules, and we would note, in particular that, although it does not use the standard package interface, the library of small groups has been refereed and accepted on exactly the same basis as the accepted packages.

We also include with this distribution a number of packages which have not (yet) gone through our refereeing process. Some may be accepted in the future, in other cases the authors have chosen not to submit them. More information can be found on our World Wide Web site (see Section 1.5).

1.3 Acknowledgements

Very many people have worked on, and contributed to, GAP over the years since its inception. On our Web site you will find the prefaces to the previous releases, each of which acknowledges people who

have made special contributions to that release. Even so, it is appropriate to mention here Joachim Neubüser whose vision of a free, open and extensible system for computational algebra inspired GAP in the first place, and Martin Schönert, who was the technical architect of GAP 3 and GAP 4.

1.4 Copyright and License

Copyright © (1987-2015) by the GAP Group,

incorporating the Copyright © 1999, 2000 by School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland

being the Copyright © 1992 by Lehrstuhl D für Mathematik, RWTH, 52056 Aachen, Germany, transferred to St Andrews on July 21st, 1997.

except for files in the distribution, which have an explicit different copyright statement. In particular, the copyright of packages distributed with GAP is usually with the package authors or their institutions.

GAP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. For details, see the file `GPL` in the `etc` directory of the GAP distribution or see `http://www.gnu.org/licenses/gpl.html`.

If you obtain GAP please send us a short notice to that effect, e.g., an e-mail message to the address `support@gap-system.org`. This helps us to keep track of the number of GAP users.

If you publish a mathematical result that was partly obtained using GAP, please cite GAP, just as you would cite another paper that you used (see below for sample citation). Also we would appreciate if you could inform us about such a paper, which we will add to the GAP bibliography.

Specifically, please refer to

```
[GAP] The GAP Group, GAP - Groups, Algorithms, and Programming,
      Version 4.7.9; 2015 (http://www.gap-system.org)
```

You are permitted to modify and redistribute GAP, but you are not allowed to restrict further redistribution. That is to say proprietary modifications will not be allowed. We want all versions of GAP to remain free.

If you modify any part of GAP and redistribute it, you must supply a `README` document. This should specify what modifications you made in which files. We do not want to take credit or be blamed for your modifications.

Of course we are interested in all of your modifications. In particular we would like to see bugfixes, improvements and new functions. So again we would appreciate it if you would inform us about all modifications you make.

In addition to the general copyright for GAP set forth above, the following terms apply to the versions of GAP for Windows.

The executable of GAP for Windows that we distribute was compiled with the `gcc` compiler supplied with Cygwin installation (`http://cygwin.com/`).

The GNU C compiler is

Copyright © 2010 Free Software Foundation, Inc.

under the terms of the GNU General Public License (GPL).

The Cygwin API library is also covered by the GNU GPL. The executable we provide is linked against this library (and in the process includes GPL'd Cygwin glue code). This means that the executable falls under the GPL too, which it does anyhow.

The cyggcc_s-1.dll, cygncurses-10.dll, cygncursesw-10.dll, cygpanel-10.dll, cygpopt-0.dll, cygreadline7.dll, cygstart.exe, cygwin1.dll, libW11.dll, mintty.exe, rxvt.exe and regtool.exe are taken unmodified from the Cygwin distribution. They are copyright by RedHat Software and released under the GPL. For more information on Cygwin, see http://www.cygwin.com.

Please contact support@gap-system.org if you need further information.

1.5 Further Information about GAP

Information about GAP is best obtained from the GAP website

http://www.gap-system.org

There you will find, amongst other things

- directions to the sites from which you can download the current GAP distribution, all accepted and deposited GAP packages, and a selection of other contributions.
- the GAP manual and an archive of the gap-forum mailing list, formatted for reading with a Web browser, and indexed for searching.
- information about GAP developers, and about the email addresses available for comment, discussion and support.

We would particularly ask you to note the following things:

- The GAP Forum – an email discussion forum for comments, discussions or questions about GAP. You must subscribe to the list before you can post to it, see the website for details. In particular we will announce new releases in this mailing list.
- The email address support@gap-system.org to which you are asked to send any questions or bug reports which do not seem likely to be of interest to the whole GAP Forum. Please give a (short, if possible) self-contained excerpt of a GAP session containing both input and output that illustrates your problem (including comments of why you think it is a bug) and state the type of the machine, operating system, (compiler used, if UNIX/Linux) and the version of GAP you are using (the first line after the GAP 4 banner starting GAP, Version 4...).
- We also ask you to send a brief message to support@gap-system.org when you install GAP.
- The correct form of citation of GAP, which we ask you use whenever you publish scientific results obtained using GAP.

It finally remains for us to wish you all pleasure and success in using GAP, and to invite your constructive comment and criticism.

The GAP Group,
29-Nov-2015

Chapter 2

The Help System

This chapter describes the GAP help system. The help system lets you read the documentation interactively.

2.1 Invoking the Help

The basic command to read GAP's documentation from within a GAP session is as follows.

`?[book:][?]topic`

For an explanation and some examples see (**Tutorial: Help**).

Note that the first question mark must appear in the *first position* after the `gap>` prompt. The search strings *book* and *topic* are normalized in a certain way (see the end of this section for details) before the search starts. This makes the search case insensitive and there can be arbitrary white space after the first question mark.

When there are several manual sections that match the query a numbered list of topics is displayed. These matches can be accessed with `?number`.

There are some further specially handled commands which start with a question mark. They are explained in Section 2.2.

By default GAP shows the help sections as text in the terminal (window), page by page if the shown text does not fit on the screen. But there are several other choices to read (other formats of) the documents: via a viewer for `pdf` files or via a web browser. This is explained below in Section 2.3.

Details of the string normalization process

Here is a precise description how the search strings *book* and *topic* are normalized before a search starts: backslashes and double or single quotes are removed, parentheses and braces are substituted by blanks, non-ASCII characters are considered as ISO-latin1 characters and the accented letters are substituted by their non-accented counterpart. Finally white space is normalized.

2.2 Browsing through the Sections

Help books for GAP are organized in chapters, sections, and subsections. There are a few special commands starting with a question mark (in the first position after the `gap>` prompt) which allow browsing a book section or chapter wise.

`?>`

`?<`

The two help commands ?< and ?> allow one to browse through a whole help book. ?< displays the section or subsection preceding the previously shown (sub)section, and ?> takes you to the section or subsection following the previously shown one.

?>>

?<<

?<< takes you back to the beginning of the current chapter. If you are already at the start of a chapter ?<< takes you to the beginning of the previous chapter. ?>> takes you to the beginning of the next chapter.

?-

?+

GAP remembers the last few sections that you have read. ?- takes you to the one that you have read before the current one, and displays it again. Further applications of ?- take you further back in this history. ?+ reverses this process, i.e., it takes you back to the section that you have read after the current one. It is important to note that ?- and ?+ do not alter the history like the other help commands.

?books

This command shows a list of the books which are currently known to the help system. For each book there is a short name which is used with the *book* part of the basic help query and there is a long name which hopefully tells you what this book is about.

A short name which ends in (not loaded) refers to a GAP package whose documentation is loaded but which needs a call of LoadPackage (76.2.1) before you can use the described functions.

?[*book*:]sections

?[*book*:][chapters]

These commands show tables of contents for all available, respectively the matching books. For some books these commands show the same, namely the whole table of contents.

?

?&

These commands redisplay the last shown help section. In the form ?& the next preferred help viewer is used for the display (provided one has chosen several viewers), see SetHelpViewer (2.3.1) below.

2.3 Changing the Help Viewer

Books of the GAP help system or package manuals can be available in several formats. Currently the following formats occur (not all of them may be available for all books):

text This is used for display in the terminal window in which GAP is running. Complicated mathematical expressions may not be easy to read in this format.

pdf Adobe's pdf format. Can be used for printing and onscreen reading on most current systems (with freely available software). Some manual books contain hyperlinks in this format.

HTML

The format of web pages. Can be used with any web browser. There may be hyperlink information available which allows a convenient browsing through the book via cross-references. This format has the problem that complicated formulae may be not be easy to read since there is no

syntax for formulae in HTML. (Some older manual books use special symbol fonts for formulae and need a particular configuration of the web browser for correct display. Some manuals may use technology for quite sophisticated formula display.)

Depending on your operating system and available additional software you can use several of these formats with GAP's help system. This is configured with the following command.

2.3.1 SetHelpViewer

▷ `SetHelpViewer(`*viewer1, viewer2, ...*`)` (function)

This command takes an arbitrary number of arguments which must be strings describing a viewer. The recognized viewers are explained below. A call with no arguments shows the current setting.

The first given arguments are those with higher priority. So, if a help section is available in the format needed by *viewer1*, this viewer is used. If not, availability of the format for *viewer2* is checked and so on. Recall that the command `?&` displays the last seen section again but with the next possible viewer in your list, see 2.2.

The viewer `"screen"` (see below) is always silently appended since we assume that each help book is available in text format.

If you want to change the default setting you can use a call of `SetUserPreference( "HelpViewers", [ ... ] );` (the list in the second argument containing the viewers you want) in your `gap.ini` file (see 3.2).

`"screen"`
: This is the default setting. The help is shown in text format using the `Pager` (2.4.1) command. Hint: Text versions of manuals are formatted assuming that your terminal displays at least 80 characters per line, if this is not the case some sections may look very bad. We suggest to use a terminal in `UTF-8` encoding with a fixed width font (this is the default on most modern Linux/Windows/Mac systems anyway). Terminals in `ISO-8859-X` encoding will also work reasonably well (so far, since we do not yet use many special characters which such terminals could not display).

`"firefox"`, `chrome`, `"mozilla"`, `"netscape"`, `"konqueror"`
: If a book is available in HTML format this is shown using the corresponding web browser. How well this works, for example by using a running instance of this browser, depends on your particular start script of this browser. (Note, that for some old books the browser must be configured to use symbol fonts.)

`"browser"`
: (for MS Windows) If a book is available in HTML format, it will be opened using the Windows default application (typically, a web browser).

`"links2"`, `"w3m"`, `"lynx"`
: If a book is available in HTML format this is shown using the text based `"links2"` (in graphics mode), `w3m` or `lynx` web browser, respectively, inside the terminal running GAP. (Formulae in some older books which use symbol fonts may be unreadable.)

`"mac default browser"`, `"browser"`, `"safari"`, `"firefox"`
: (for Mac OS X) If a book is available in HTML format this is shown in a web browser. The

options "safari" and "firefox" use the corresponding browsers. The other two options use the program default browser (which can be set in Safari's preferences, in the "General" tab).

"xpdf"
: (on X-windows systems) If a book is available in pdf format it is shown with the onscreen viewer program xpdf (which must be installed on your system). This is a nice program, once it is running it is reused by GAP for the next displays of help sections.

"acroread"
: If a book is available in pdf format it is shown with the onscreen viewer program acroread (which must be available on your system). This program does not allow remote commands or startup with a given page. Therefore the page numbers you have to visit are just printed on the screen. When you are looking at several sections of the same book, this viewer assumes that the acroread window still exists. When you go to another book a new acroread window is launched.

"pdf viewer", "skim", "preview", "adobe reader"
: (for Mac OS X) If a book is available in pdf format this is shown in a pdf viewer. The options "skim", "preview" and "adobe reader" use the corresponding viewers. The other two options use the pdf viewer which you have chosen to open pdf files from the Finder. Note that only "Skim" seems to be capable to open a pdf file on a given page. For the other help viewers, the page numbers where the information can be found will just be printed on the screen. None of the help viewers seems to be capable of opening a pdf at a given named destination (i. e., jump to precisely the place where the information can be found). The pdf viewer "Skim" is open source software, it can be downloaded from http://skim-app.sourceforge.net/.

"less" **or** "more"
: This is the same as "screen" but additionally the user preferences "Pager" and ""PagerOptions" are set, see the section 2.4 for more details.

Please, send ideas for further viewer commands to support@gap-system.org.

2.4 The Pager Command

GAP contains a builtin pager which shows a text string which does not fit on the screen page by page. Its functionality is very rudimentary and self-explaining. This is because (at least under UNIX) there are powerful external standard programs which do this job.

2.4.1 Pager

▷ Pager(*lines*) (function)

This function can be used to display a text on screen using a pager, i.e., the text is shown page by page.

There is a default builtin pager in GAP which has very limited capabilities but should work on any system.

At least on a UNIX system one should use an external pager program like less or more. GAP assumes that this program has a command line option +nr which starts the display of the text with line number nr.

Which pager is used can be controlled by setting the user preference `"Pager"`. The default value is `"builtin"` which means that the internal pager is used.

On UNIX systems you probably want to set the user preference `"Pager"` to the value `"less"` or `"more"`, you can do this for example in your `gap.ini` file (see 3.2). In that case you can also tell GAP a list of standard options for the external pager, via the user preference `"PagerOptions"`.

Example

```
SetUserPreference( "Pager", "less" );
SetUserPreference( "PagerOptions", ["-f","-r","-a","-i","-M","-j2"] );
```

The argument *lines* can have one of the following forms:

1. a string (i.e., lines are separated by newline characters)
2. a list of strings (without newline characters) which are interpreted as lines of the text to be shown
3. a record with component `lines` as in 1. or 2. and optional further components

In case 3. currently the following additional components are used:

`formatted`
: can be `false` or `true`. If set to `true` the builtin pager tries to show the text exactly as it is given (avoiding GAP's automatic line breaking),

`start`
: must be a positive integer. This is interpreted as the number of the first line shown by the pager (one may see the beginning of the text via back scrolling).

`exitAtEnd`
: can be `false` or `true`. If set to `true` (the default), the builtin pager is terminated as soon as the end of the list is shown; otherwise entering the `Q` key is necessary in order to return from the pager.

The `Pager` command is used by GAP's help system for displaying help sections in text format. But, of course, it may be used for other purposes as well.

Example

```
gap> s6 := SymmetricGroup(6);;
gap> words := ["This", "is", "a", "very", "stupid", "example"];;
gap> l := List(s6, p-> Permuted(words, p));;
gap> Pager(List(l, a-> JoinStringsWithSeparator(a," ")));;
```

Chapter 3

Running GAP

This chapter informs about command line options for GAP (see 3.1), some files in user specific GAP root directory (see 3.2) and saving and loading a GAP workspace (see 3.3).

3.1 Command Line Options

When you start GAP from a command line or from a script you may specify a number of options on the command-line to change the default behaviour of GAP. All these options start with a hyphen `-`, followed by a single letter. Options must not be grouped, e.g., `gap -gq` is invalid, use `gap -g -q` instead. Some options require an argument, this must follow the option and must be separated by whitespace, e.g., `gap -m 256m`, it is not correct to say `gap -m256m` instead. Certain Boolean options (`-b`, `-q`, `-e`, `-r`, `-A`, `-D`, `-M`, `-T`, `-X`, `-Y`) toggle the current value so that `gap -b -b` is equivalent to `gap` and to `gap -b -q -b -q` etc.

GAP for UNIX will distinguish between upper and lower case options.

As described in the GAP installation instructions (see the `INSTALL` file in the GAP root directory, or at `http://www.gap-system.org/Download/INSTALL`), usually you will not execute GAP directly. Instead you will call a (shell) script, with the name `gap`, which in turn executes GAP. This script sets some options which are necessary to make GAP work on your system. This means that the default settings mentioned below may not be what you experience when you execute GAP on your system.

During a GAP session, one can find the current values of command line options in the record `GAPInfo.CommandLineOptions` (see `GAPInfo` (3.5.1)), whose component names are the command line options (without the leading `-`).

`-A` By default, some needed and suggested GAP packages (see 76) are loaded, if present, into the GAP session when it starts. This option disables (actually toggles) the loading of suggested packages, which can be useful for debugging or testing. The needed packages (and their needed packages, and so on) are loaded in any case.

`-a` *memory*
GASMAN, the storage manager of GAP uses `sbrk` to get blocks of memory from (certain) operating systems and it is required that subsequent calls to `sbrk` produce adjacent blocks of memory in this case because GAP only wants to deal with one large block of memory. If the C function `malloc` is called for whatever reason, it is likely that `sbrk` will no longer produce adjacent blocks, therefore GAP does not use `malloc` itself.

However some operating systems insist on calling `malloc` to create a buffer when a file is opened, or for some other reason. In order to catch these cases GAP preallocates a block of memory with `malloc` which is immediately freed. The amount preallocated can be controlled with the `-a` option. (Most users do not need this option.)

The option argument *`memory`* is specified as with the `-m` option.

`-B` *`architecture`*
Executable binary files that form part of GAP or of a GAP package are kept in a subdirectory of the `bin` directory within the GAP or package root directory. The subdirectory name is determined from the operating system, processor and compiler details when GAP (resp. the package) is installed. Under rare circumstances, it may be necessary to override this name, and this can be done using the `-B` option.

`-b` tells GAP to suppress the banner. That means that GAP immediately prints the prompt. This is useful when, after a while, you get tired of the banner. This option can be repeated to enable the banner; each `-b` toggles the state of banner display.

`-D` The `-D` option tells GAP to print short messages when it is reading files or loading modules. This option may be repeated to toggle this behavior on and off. The message,

Example
```
#I  READ_GAP_ROOT: loading 'lib/kernel.g' as GAP file
```

tells you that GAP has started to read the library file `lib/kernel.g`.

Example
```
#I  READ_GAP_ROOT: loading 'lib/kernel.g' statically
```

tells you that GAP has used the compiled version of the library file `lib/kernel.g`. This compiled module was statically linked to the GAP kernel at the time the kernel was created.

Example
```
#I  READ_GAP_ROOT: loading 'lib/kernel.g' dynamically
```

tells you that GAP has loaded the compiled version of the library file `lib/kernel.g`. This compiled module was dynamically loaded to the GAP kernel at runtime from a corresponding `.so` file.

Obviously, this is a debugging option and most users will not need it.

`-E` If your GAP installation uses the `readline` library for command line editing (see 6.9), this may be disabled by using `-E` option. This option may be repeated to toggle this behavior on and off. If your GAP installation does not use the `readline` library (you can check by `IsBound(GAPInfo.UseReadline);` if this is the case), this option will have no effect at all.

`-e` tells GAP not to quit when receiving a CTRL-D on an empty input line (see 6.4.1). This option should not be used when the input is a file or pipe. This option may be repeated to toggle this behavior on and off.

`-f` tells GAP to enable the line editing and history (see 6.8).

In general line editing will be enabled if the input is connected to a terminal. There are rare circumstances, for example when using a remote session with a corrupted telnet implementation, when this detection fails. Try using `-f` in this case to enable line editing. This option does not toggle; you must use `-n` to disable line editing.

`-g` tells GAP to print a message every time a full garbage collection is performed.

Example

```
#G  FULL 44580/2479kb live   57304/4392kb dead   734/4096kb free
```

For example, this tells you that there are 44580 live objects that survived a full garbage collection, that 57304 unused objects were reclaimed by it, and that 734 kilobytes from a total allocated memory of 4096 kilobytes are available afterwards.

`-g -g`
If you give the option `-g` twice, GAP prints a information message every time a partial or full garbage collection is performed. The message,

Example

```
#G  PART 9405/961kb+live   7525/1324kb+dead   2541/4096kb free
```

for example, tells you that 9405 objects survived the partial garbage collection and 7525 objects were reclaimed, and that 2541 kilobytes from a total allocated memory of 4096 kilobytes are available afterwards.

`-h` tells GAP to print a summary of all available options (`-h` is mnemonic for "help"). GAP exits after printing the summary, all other options are ignored.

`-i` *filename*
changes the name of the init file from the default `init.g` to *filename*. (Usually not needed.)

`-K` *memory*
is like the `-o` option. But while the latter actually allocates more memory if the system allows it and then prints a warning inside a break loop the `-K` options tells GAP not even to try to allocate more memory. Instead GAP just exits with an appropriate message. The default is that this feature is switched off. You have to set it explicitly when you want to enable it.

`-L` *filename*
The option `-L` tells GAP to load a saved workspace. See section 3.3.

`-l` *path_list*
can be used to set or modify GAP's list of root directories (see 9.2). The default if no `-l` option is given is the current directory `./`. This option can be used several times. Depending on the `-r` option a further user specific path is prepended to the list of root directories (the path in `GAPInfo.UserGapRoot`).

path_list should be a list of directories separated by semicolons. No whitespace is permitted before or after a semicolon. If *path_list* does not start or end with a semicolon, then *path_list* replaces the existing list of root directories. If *path_list* starts with a semicolon, then *path_list* is appended to the existing list of root directories. If *path_list* ends with

a semicolon and does not start with one, then the new list of root directories is the concatenation of *path_list* and the existing list of root directories. After GAP has completed its startup procedure and displays the prompt, the list of root directories can be seen in the variable `GAPInfo.RootPaths`, see `GAPInfo` (3.5.1).

Usually this option is used inside a startup script to specify where GAP is installed on the system. The `-l` option can also be used by individual users to tell GAP about privately installed modifications of the library, additional GAP packages and so on. Section 9.2 explains how several root paths can be used to do this.

GAP will attempt to read the file *root_dir*`/lib/init.g` during startup where *root_dir* is one of the directories in its list of root directories. If GAP cannot find its `init.g` file it will print the following warning.

Example

```
gap: hmm, I cannot find 'lib/init.g' maybe use option '-l <gaproot>'?
```

It is not possible to use GAP without the library files, so you must not ignore this warning. You should leave GAP and start it again, specifying the correct root path using the `-l` option.

`-M` tells GAP not to check for, nor to use, compiled versions of library files. This option may be repeated to toggle this behavior on and off.

`-m` *memory*
tells GAP to allocate *memory* bytes at startup time. If the last character of *memory* is `k` or `K` it is taken as kilobytes, if the last character is `m` or `M` *memory* is taken as megabytes and if it is `g` or `G` it is taken as gigabytes.

This amount of memory should be large enough so that computations do not require too many garbage collections. On the other hand, if GAP allocates more memory than is physically available, it will spend most of the time paging.

`-n` tells GAP to disable the line editing and history (see 6.8).

You may want to do this if the command line editing is incompatible with another program that is used to run GAP. For example if GAP is run from inside a GNU Emacs shell window, `-n` should be used since otherwise every input line will be echoed twice, once by Emacs and once by GAP. This option does not toggle; you must use `-f` to enable line editing.

`-O` disables loading obsolete variables (see Chapter 77). This option is used mainly for testing purposes, for example in order to make sure that a GAP package or one's own GAP code does not rely on the obsolete variables.

`-o` *memory*
tells GAP to allocate at most *memory* bytes without asking. The option argument *memory* is specified as with the `-m` option.

If more than this amount is required during the GAP session, GAP prints an error message and enters a break loop. In that case you can enter `return;` which implicitly doubles the amount given with this option.

`-q` tells GAP to be quiet. This means that GAP displays neither the banner nor the prompt `gap>`. This is useful if you want to run GAP as a filter with input and output redirection and want to

avoid the banner and the prompts appearing in the output file. This option may be repeated to disable quiet mode; each `-q` toggles quiet mode.

`-R` The option `-R` tells GAP not to load a saved workspace previously specified via the `-L` option. This option does not toggle.

`-r` The option `-r` tells GAP to ignore any user specific configuration files. In particular, the user specific root directory `GAPInfo.UserGapRoot` is not added to the GAP root directories and so `gap.ini` and `gaprc` files that may be contained in that directory are not read, see 3.2. Multiple `-r` options toggle this behaviour.

`-s` *`memory`*
With this option GAP does not use `sbrk` to get memory from the operating system. Instead it uses `mmap`, `malloc` or some other command for the amount given with this option to allocate space for the `GASMAN` memory manager. Usually GAP does not really use all of this memory, the options `-m`, `-o`, `-K` still work as documented. This feature assumes that the operating system only assigns physical memory to the GAP process when it is accessed, so that specifying a large amount of memory with `-s` should not cause any performance problem. The advantage of using this option is that GAP can work together with kernel modules which allocate a lot of memory with `malloc`.

The option argument *`memory`* is specified as with the `-m` option.

`-T` suppresses the usual break loop behaviour of GAP. With this option GAP behaves as if the user `quit` immediately from every break loop. This is intended for automated testing of GAP. This option may be repeated to toggle this behavior on and off.

`-X` tells GAP to do a consistency check of the library file and the corresponding compiled module when loading the compiled module. This option may be repeated to toggle this behavior on and off.

`-x` *`length`*
With this option you can tell GAP how long lines are. GAP uses this value to decide when to split long lines. After starting GAP you may use `SizeScreen` (6.12.1) to alter the line length.

The default value is 80, unless another value can be obtained from the Operating System, which is the right value if you have a standard terminal application. If you have a larger monitor, or use a smaller font, or redirect the output to a printer, you may want to increase this value.

`-y` *`length`*
With this option you can tell GAP how many lines your screen has. GAP uses this value to decide after how many lines of on-line help it should wait. After starting GAP you may use `SizeScreen` (6.12.1) to alter the number of lines.

The default value is 24, unless another value can be obtained from the Operating System, which is the right value if you have a standard terminal application. If you have a larger monitor, or use a smaller font, or redirect the output to a printer, you may want to increase this value.

`filename` **...**
Further arguments are taken as filenames of files that are read by GAP during startup, after the system and private init files are read, but before the first prompt is printed. The files are read in

the order in which they appear on the command line. GAP only accepts up to 14 filenames on the command line. If a file cannot be opened GAP will print an error message and will abort.

Additional options, `-C`, `-U`, `-P`, `-W`, `-p` and `-z` are used internally by the gac script (see 76.3.9) and/or on specific operating systems.

3.2 The gap.ini and gaprc files

When you start GAP, it looks for files with the names `gap.ini` and `gaprc` in its root directories (see 9.2), and reads the first `gap.ini` and the first `gaprc` file it finds. These files are used for certain initializations, as follows.

The file `gap.ini` is read early in the startup process. Therefore, the parameters set in this file can influence the startup process, such as which packages are automatically loaded (see `LoadPackage` (76.2.1)) and whether library files containing obsolete variables are read (see Chapter 77). On the other hand, only calls to a restricted set of GAP functions are allowed in a `gap.ini` file. Usually, it should only contain calls of `SetUserPreference` (3.2.3). This file can be generated (or updated when new releases introduce further user preferences) with the command `WriteGapIniFile` (3.2.3). This file is read whenever GAP is started, with or without a workspace.

The file `gaprc` is read after the startup process, before the first input file given on the command line (see 3.1). So the contents of this file cannot influence the startup process, but all GAP library functions can be called in this file. When GAP is started with a workspace then the file is read only if no `gaprc` file had been read before the workspace was created. (With this setup, it is on the one hand possible that administrators provide a GAP workspace for several users such that the user's `gaprc` file is read when GAP is started with the workspace, and on the other hand one can start GAP, read one's `gaprc` file, save a workspace, and then start from this workspace *without* reading one's `gaprc` file again.)

Note that by default, the user specific GAP root directory `GAPInfo.UserGapRoot` is the first GAP root directory. So you can put your `gap.ini` and `gaprc` files into this directory.

This mechanism substitutes the much less flexible reading of a users `.gaprc` file in versions of GAP up to 4.4. For compatibility this `.gaprc` file is still read if the directory `GAPInfo.UserGapRoot` does not exist, see 77.5 how to migrate your old setup.

3.2.1 The gap.ini file

The file `gap.ini` is read after the declaration part of the GAP library is read, before the declaration parts of the packages needed and suggested by GAP are read, and before the implementation parts of GAP and of the packages are read.

The file `gap.ini` is expected to consist of calls to the function `SetUserPreference` (3.2.3), see Section `SetUserPreference` (3.2.3).

Since the file `gap.ini` is read before the implementation part of GAP is read, not all GAP functions may be called in the file. Assignments of numbers, lists, and records are admissible as well as calls to basic functions such as `Concatenation` (21.20.1) and `JoinStringsWithSeparator` (27.6.17).

Note that the file `gap.ini` is read also when GAP is started with a workspace.

3.2.2 The gaprc file

If a file gaprc is found it is read after GAP's init.g, but before any of the files mentioned on the command line are read. You can use this file for your private customizations. (Many users may be happy with using just user preferences in the gap.ini file (see above) for private customization.) For example, if you have a file containing functions or data that you always need, you could read this from gaprc. Or if you find some of the names in the library too long, you could define abbreviations for those names in gaprc. The following sample gaprc file does both.

Example

```
Read( "/usr/you/dat/mygroups.grp" );
Ac := Action;
AcHom := ActionHomomorphism;
RepAc := RepresentativeAction;
```

Note that only one gaprc file is read when GAP is started. When a workspace is created in a GAP session after a gaprc file has been read then no more gaprc file will be read when GAP is started with this workspace.

Also note that the file must be called gaprc. If you use a Windows text editor, in particular if your default is not to show file suffixes, you might accidentally create a file gaprc.txt or gaprc.doc which GAP will not recognize.

3.2.3 Configuring User preferences

▷ SetUserPreference(*[package,]name, value*) (function)
▷ UserPreference(*[package,]name*) (function)
▷ ShowUserPreferences(*package1, package2, ...*) (function)
▷ WriteGapIniFile(*[dir,][ignorecurrent]*) (function)

Some aspects of the behaviour of GAP can be customized by the user via *user preferences*. Examples include the way help sections are displayed or the use of colors in the terminal.

User preferences are specified via a pair of strings, the first is the (case insensitive) name of a package (or "GAP" for the core GAP library) and the second is some arbitrary case sensitive string.

User preferences can be set to some *value* with SetUserPreference. The current value of a user preference can be found with UserPreference. In both cases, if no package name is given the default "GAP" is used. If a user preference is not known or not set then UserPreference returns fail.

The function ShowUserPreferences with no argument shows in a pager an overview of all known user preferences together with some explanation and the current value. If one or more strings *package1*, ... are given then only the user preferences for these packages are shown.

The easiest way to make use of user preferences is probably to use the function WriteGapIniFile, usually without argument. This function creates a file gap.ini in your user specific GAP root directory (GAPInfo.UserGapRoot). If such a file already exists the function will make a backup of it first. This newly created file contains descriptions of all known user preferences and also calls of SetUserPreference for those user preferences which currently do not have their default value. You can then edit that file to customize (further) the user preferences for future GAP sessions.

Should a later version of GAP or some packages introduce new user preferences then you can call WriteGapIniFile again since it will set the previously known user preferences to their current values.

Optionally, a different directory for the resulting gap.ini file can be specified as argument *dir* to WriteGapIniFile. Another optional argument is the boolean value true, if this is given, the settings of all user preferences in the current session are ignored.

Note that your gap.ini file is read by GAP very early during its startup process. A consequence is that the *value* argument in a call of SetUserPreference must be some very basic GAP object, usually a boolean, a number, a string or a list of those. A few user preferences support more complicated settings. For example, the user preference "UseColorPrompt" admits a record as its value whose components are available only after the GAPDoc package has been loaded, see ColorPrompt (3.6.1). If you want to specify such a complicated value, then move the corresponding call of SetUserPreference from your gap.ini file into your gaprc file (also in the directory GAPInfo.UserGapRoot). This file is read much later.

Example

```
gap> SetUserPreference( "Pager", "less" );
gap> SetUserPreference("PagerOptions",
>                      [ "-f", "-r", "-a", "-i", "-M", "-j2" ] );
gap> UserPreference("Pager");
"less"
```

The first two lines of this example will cause GAP to use the programm less as a pager. This is highly recommended if less is available on your system. The last line displays the current setting.

3.2.4 DeclareUserPreference

▷ DeclareUserPreference(*record*) (function)

This function can be used (also in packages) to introduce new user preferences. It declares a user preference, determines a default value and contains documentation of the user preference. After declaration a user preference will be shown with ShowUserPreferences (3.2.3) and WriteGapIniFile (3.2.3).

When this declaration is evaluated it is checked, if this user preference is already set in the current session. If not the value of the user preference is set to its default. (Do not use fail as default value since this indicated that a user preference is not set.)

The argument *record* of DeclareUserPreference must be a record with the following components.

name
: a string or a list of strings, the latter meaning several preferences which belong together,

description
: a list of strings describing the preference(s), one string for each paragraph; if several preferences are declared together then the description refers to all of them,

default
: the default value that is used, or a function without arguments that computes this default value; if several preferences are declared together then the value of this component must be the list of default values for the individual preferences.

The following components of *record* are optional.

`check`
: a function that takes a value as its argument and returns either `true` or `false`, depending on whether the given value is admissible for this preference; if several preferences are declared together then the number of arguments of the function must equal the length of the `name` list,

`values`
: the list of admissible values, or a function without arguments that returns this list,

`multi`
: `true` or `false`, depending on whether one may choose several values from the given list or just one; needed (and useful only) if the `values` component is present,

`package`
: the name of the GAP package to which the preference is assigned; if the declaration happens inside a file that belongs to this package then the value of this component is computed, using `GAPInfo.PackageCurrent`; otherwise, the default value for `package` is `"GAP"`,

`omitFromGapIniFile`
: if the value is `true` then this user preference is ignored by `WriteGapIniFile` (3.2.3).

Example

```
gap> UserPreference( "MyFavouritePrime" );
fail
gap> DeclareUserPreference( rec(
>        name:= "MyFavouritePrime",
>        description:= [ "is not used, serves as an example" ],
>        default:= 2,
>        omitFromGapIniFile:= true ) );
gap> UserPreference( "MyFavouritePrime" );
2
gap> SetUserPreference( "MyFavouritePrime", 17 );
gap> UserPreference( "MyFavouritePrime" );
17
```

3.3 Saving and Loading a Workspace

GAP workspace files are binary files that contain the data of a GAP session. One can produce a workspace file with `SaveWorkspace` (3.3.1), and load it into a new GAP session using the `-L` command line option, see Section 3.1.

One purpose of workspace files is of course the possibility to save a "snapshot" image of the current GAP workspace in a file.

The recommended way to start GAP is to load an existing workspace file, because this reduces the startup time of GAP drastically. So if you have installed GAP yourself then you should think about creating a workspace file immediately after you have started GAP, and then using this workspace file later on, whenever you start GAP. If your GAP installation is shared between several users, the system administrator should think about providing such a workspace file.

3.3.1 SaveWorkspace

▷ SaveWorkspace(*filename*) (function)

will save a "snapshot" image of the current GAP workspace in the file *filename*. This image then can be loaded by another copy of GAP which then will behave as at the point when SaveWorkspace was called.

Example

```
gap> a:=1;
gap> SaveWorkspace("savefile");
true
gap> quit;
```

SaveWorkspace can only be used at the main gap> prompt. It cannot be included in the body of a loop or function, or called from a break loop.

3.4 Testing for the System Architecture

3.4.1 ARCH_IS_UNIX

▷ ARCH_IS_UNIX() (function)

tests whether GAP is running on a UNIX system (including Mac OS X).

3.4.2 ARCH_IS_MAC_OS_X

▷ ARCH_IS_MAC_OS_X() (function)

tests whether GAP is running on Mac OS X. Note that on Mac OS X, also ARCH_IS_UNIX (3.4.1) will be true.

3.4.3 ARCH_IS_WINDOWS

▷ ARCH_IS_WINDOWS() (function)

tests whether GAP is running on a Windows system.

3.5 Global Values that Control the GAP Session

3.5.1 GAPInfo

▷ GAPInfo (global variable)

Several global values control the GAP session, such as the command line, the architecture, or the information about available and loaded packages. Many of these values are accessible as components of the global record GAPInfo. Typically, these components are set and read in low level GAP functions, so changing the values of existing components of GAPInfo "by hand" is not recommended.

Important components are documented via index entries, try the input ??GAPInfo for getting an overview of these components.

3.6 Coloring the Prompt and Input

GAP provides hooks for functions which are called when the prompt is to be printed and when an input line is finished.

An example of using this feature is the following function.

3.6.1 ColorPrompt

▷ `ColorPrompt(`*`bool[, optrec]`*`)` (function)

With `ColorPrompt(true);` GAP changes its user interface: The prompts and the user input are displayed in different colors. Switch off the colored prompts with `ColorPrompt(false);`.

Note that this will only work if your terminal emulation in which you run GAP understands the so called ANSI color escape sequences –almost all terminal emulations on current UNIX/Linux (`xterm`, `rxvt`, `konsole`, ...) systems do so.

The colors shown depend on the terminal configuration and cannot be forced from an application. If your terminal follows the ANSI conventions you see the standard prompt in bold blue and the break loop prompt in bold red, as well as your input in red.

If it works for you and you like it, put a call of `SetUserPreference("UseColorPrompt", true);` in your `gap.ini` file. If you want a more complicated setting as explained below then put your `SetUserPreference("UseColorPrompt", rec( ... ) );` call into your `gaprc` file.

The optional second argument *`optrec`* allows one to further customize the behaviour. It must be a record from which the following components are recognized:

`MarkupStdPrompt`
: a string or no argument function returning a string containing the escape sequence used for the main prompt `gap>` .

`MarkupContPrompt`
: a string or no argument function returning a string containing the escape sequence used for the continuation prompt `>` .

`MarkupBrkPrompt`
: a string or no argument function returning a string containing the escape sequence used for the break prompt `brk...>` .

`MarkupInput`
: a string or no argument function returning a string containing the escape sequence used for user input.

`TextPrompt`
: a no argument function returning the string with the text of the prompt, but without any escape sequences. The current standard prompt is returned by `CPROMPT()`. But note that changing the standard prompts makes the automatic removal of prompts from input lines impossible (see 6.2).

`PrePrompt`
: a function called before printing a prompt.

Here is an example.

```
LoadPackage("GAPDoc");
timeSHOWMIN := 100;
ColorPrompt(true, rec(
   # usually cyan bold, see ?TextAttr
   MarkupStdPrompt := Concatenation(TextAttr.bold, TextAttr.6),
   MarkupContPrompt := Concatenation(TextAttr.bold, TextAttr.6),
   PrePrompt := function()
     # show the 'time' automatically if at least timeSHOWMIN
     if CPROMPT() = "gap> " and time >= timeSHOWMIN then
       Print("Time of last command: ", time, " ms\n");
     fi;
   end)    );
```

Chapter 4

The Programming Language

This chapter describes the GAP programming language. It should allow you in principle to predict the result of each and every input. In order to know what we are talking about, we first have to look more closely at the process of interpretation and the various representations of data involved.

4.1 Language Overview

First we have the input to GAP, given as a string of characters. How those characters enter GAP is operating system dependent, e.g., they might be entered at a terminal, pasted with a mouse into a window, or read from a file. The mechanism does not matter. This representation of expressions by characters is called the *external representation* of the expression. Every expression has at least one external representation that can be entered to get exactly this expression.

The input, i.e., the external representation, is transformed in a process called *reading* to an internal representation. At this point the input is analyzed and inputs that are not legal external representations, according to the rules given below, are rejected as errors. Those rules are usually called the *syntax* of a programming language.

The internal representation created by reading is called either an *expression* or a *statement*. Later we will distinguish between those two terms. However for now we will use them interchangeably. The exact form of the internal representation does not matter. It could be a string of characters equal to the external representation, in which case the reading would only need to check for errors. It could be a series of machine instructions for the processor on which GAP is running, in which case the reading would more appropriately be called compilation. It is in fact a tree-like structure.

After the input has been read it is again transformed in a process called *evaluation* or *execution*. Later we will distinguish between those two terms too, but for the moment we will use them interchangeably. The name hints at the nature of this process, it replaces an expression with the value of the expression. This works recursively, i.e., to evaluate an expression first the subexpressions are evaluated and then the value of the expression is computed from those values according to rules given below. Those rules are usually called the *semantics* of a programming language.

The result of the evaluation is, not surprisingly, called a *value*. Again the form in which such a value is represented internally does not matter. It is in fact a tree-like structure again.

The last process is called *printing*. It takes the value produced by the evaluation and creates an external representation, i.e., a string of characters again. What you do with this external representation is up to you. You can look at it, paste it with the mouse into another window, or write it to a file.

Lets look at an example to make this more clear. Suppose you type in the following string of 8 characters

```
1 + 2 * 3;
```

GAP takes this external representation and creates a tree-like internal representation, which we can picture as follows

```
  +
 / \
1   *
   / \
  2   3
```

This expression is then evaluated. To do this GAP first evaluates the right subexpression `2*3`. Again, to do this GAP first evaluates its subexpressions 2 and 3. However they are so simple that they are their own value, we say that they are self-evaluating. After this has been done, the rule for * tells us that the value is the product of the values of the two subexpressions, which in this case is clearly 6. Combining this with the value of the left operand of the +, which is self-evaluating, too, gives us the value of the whole expression 7. This is then printed, i.e., converted into the external representation consisting of the single character 7.

In this fashion we can predict the result of every input when we know the syntactic rules that govern the process of reading and the semantic rules that tell us for every expression how its value is computed in terms of the values of the subexpressions. The syntactic rules are given in sections 4.2, 4.3, 4.4, 4.5, and 4.6, the semantic rules are given in sections 4.7, 4.8, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.23, and the chapters describing the individual data types.

4.2 Lexical Structure

Most input of GAP consists of sequences of the following characters.

Digits, uppercase and lowercase letters, SPACE, TAB, NEWLINE, RETURN and the special characters

```
"   `   (   )   *   +   ,   -   #
.   /   :   ;   <   =   >   ~
[   \   ]   ^   _   {   }   !
```

It is possible to use other characters in identifiers by escaping them with backslashes, but we do not recommend to use this feature. Inside strings (see section 4.3 and chapter 27) and comments (see 4.4) the full character set supported by the computer is allowed.

4.3 Symbols

The process of reading, i.e., of assembling the input into expressions, has a subprocess, called *scanning*, that assembles the characters into symbols. A *symbol* is a sequence of characters that form a lexical unit. The set of symbols consists of keywords, identifiers, strings, integers, and operator and delimiter symbols.

A *keyword* is a reserved word (see 4.5). An *identifier* is a sequence of letters, digits and underscores (or other characters escaped by backslashes) that contains at least one non-digit and is not a keyword (see 4.6). An integer is a sequence of digits (see 14), possibly prepended by - and + sign characters. A *string* is a sequence of arbitrary characters enclosed in double quotes (see 27).

Operator and delimiter symbols are

```
+    -    *    /    ^    ~   !.
=    <>   <    <=   >    >=  ![
:=   .    ..   ->   ,    ;   !{
[    ]    {    }    (    )    :
```

Note also that during the process of scanning all whitespace is removed (see 4.4).

4.4 Whitespaces

The characters SPACE, TAB, NEWLINE, and RETURN are called *whitespace characters*. Whitespace is used as necessary to separate lexical symbols, such as integers, identifiers, or keywords. For example `Thorondor` is a single identifier, while `Th or ondor` is the keyword `or` between the two identifiers `Th` and `ondor`. Whitespace may occur between any two symbols, but not within a symbol. Two or more adjacent whitespace characters are equivalent to a single whitespace. Apart from the role as separator of symbols, whitespace characters are otherwise insignificant. Whitespace characters may also occur inside a string, where they are significant. Whitespace characters should also be used freely for improved readability.

A *comment* starts with the character #, which is sometimes called sharp or hatch, and continues to the end of the line on which the comment character appears. The whole comment, including # and the NEWLINE character is treated as a single whitespace. Inside a string, the comment character # loses its role and is just an ordinary character.

For example, the following statement

```
if i<0 then a:=-i;else a:=i;fi;
```

is equivalent to

```
if i < 0 then   # if i is negative
  a := -i;      #   take its additive inverse
else            # otherwise
  a := i;       #   take itself
fi;
```

(which by the way shows that it is possible to write superfluous comments). However the first statement is *not* equivalent to

```
ifi<0thena:=-i;elsea:=i;fi;
```

since the keyword `if` must be separated from the identifier `i` by a whitespace, and similarly `then` and `a`, and `else` and `a` must be separated.

4.5 Keywords

Keywords are reserved words that are used to denote special operations or are part of statements. They must not be used as identifiers. The list of keywords is contained in the `GAPInfo.Keywords` component of the `GAPInfo` record (see 3.5.1). We will show how to print it in a nice table, demonstrating at the same time some list manipulation techniques:

Example

```
gap> keys:=SortedList( GAPInfo.Keywords );; l:=Length( keys );;
gap> arr:= List( [ 0 .. Int( l/4 )-1 ], i-> keys{ 4*i + [ 1 .. 4 ] } );;
gap> if l mod 4 <> 0 then Add( arr, keys{[ 4*Int(l/4) + 1 .. l ]} ); fi;
gap> Length( keys ); PrintArray( arr );
32
[ [         Assert,           Info,        IsBound,           QUIT ],
  [  TryNextMethod,         Unbind,            and,          break ],
  [       continue,             do,          elif,           else ],
  [            end,          false,             fi,            for ],
  [       function,             if,             in,          local ],
  [            mod,            not,             od,             or ],
  [           quit,            rec,         repeat,         return ],
  [           then,           true,          until,          while ] ]
```

Note that (almost) all keywords are written in lowercase and that they are case sensitive. For example `else` is a keyword; `Else`, `eLsE`, `ELSE` and so forth are ordinary identifiers. Keywords must not contain whitespace, for example `el if` is not the same as `elif`.

Note: Several tokens from the list of keywords above may appear to be normal identifiers representing functions or literals of various kinds but are actually implemented as keywords for technical reasons. The only consequence of this is that those identifiers cannot be re-assigned, and do not actually have function objects bound to them, which could be assigned to other variables or passed to functions. These keywords are `true`, `false`, `Assert` (7.5.3), `IsBound` (4.8.1), `Unbind` (4.8.2), `Info` (7.4.5) and `TryNextMethod` (78.4.1).

4.6 Identifiers

An *identifier* is used to refer to a variable (see 4.8). An identifier usually consists of letters, digits, underscores `_`, and "at"-characters `@`, and must contain at least one non-digit. An identifier is terminated by the first character not in this class. Note that the "at"-character `@` is used to implement namespaces, see Section 4.10 for details.

Examples of valid identifiers are

```
a           foo         aLongIdentifier
hello       Hello       HELLO
x100        100x       _100
some_people_prefer_underscores_to_separate_words
WePreferMixedCaseToSeparateWords
abc@def
```

Note that case is significant, so the three identifiers in the second line are distinguished.

The backslash \ can be used to include other characters in identifiers; a backslash followed by a character is equivalent to the character, except that this escape sequence is considered to be an ordinary letter. For example

```
G\(2\,5\)
```

is an identifier, not a call to a function `G`.

An identifier that starts with a backslash is never a keyword, so for example `\*` and `\mod` are identifiers.

The length of identifiers is not limited, however only the first 1023 characters are significant. The escape sequence \NEWLINE is ignored, making it possible to split long identifiers over multiple lines.

4.6.1 IsValidIdentifier

▷ `IsValidIdentifier(`*`str`*`)` (function)

returns `true` if the string *`str`* would form a valid identifier consisting of letters, digits and underscores; otherwise it returns `false`. It does not check whether *`str`* contains characters escaped by a backslash \.

Note that the "at"-character is used to implement namespaces for global variables in packages. See 4.10 for details.

4.7 Expressions

An *expression* is a construct that evaluates to a value. Syntactic constructs that are executed to produce a side effect and return no value are called *statements* (see 4.14). Expressions appear as right hand sides of assignments (see 4.15), as actual arguments in function calls (see 4.11), and in statements.

Note that an expression is not the same as a value. For example `1 + 11` is an expression, whose value is the integer 12. The external representation of this integer is the character sequence `12`, i.e., this sequence is output if the integer is printed. This sequence is another expression whose value is the integer 12. The process of finding the value of an expression is done by the interpreter and is called the *evaluation* of the expression.

Variables, function calls, and integer, permutation, string, function, list, and record literals (see 4.8, 4.11, 14, 42, 27, 4.23, 21, 29), are the simplest cases of expressions.

Expressions, for example the simple expressions mentioned above, can be combined with the operators to form more complex expressions. Of course those expressions can then be combined further with the operators to form even more complex expressions. The *operators* fall into three classes. The *comparisons* are `=`, `<>`, `<`, `<=`, `>`, `>=`, and `in` (see 4.12 and 30.6). The *arithmetic operators* are `+`, `-`, `*`, `/`, `mod`, and `^` (see 4.13). The *logical operators* are `not`, `and`, and `or` (see 20.4).

The following example shows a very simple expression with value 4 and a more complex expression.

Example

```
gap> 2 * 2;
4
gap> 2 * 2 + 9 = Fibonacci(7) and Fibonacci(13) in Primes;
true
```

For the precedence of operators, see 4.12.

4.8 Variables

A *variable* is a location in a GAP program that points to a value. We say the variable is *bound* to this value. If a variable is evaluated it evaluates to this value.

Initially an ordinary variable is not bound to any value. The variable can be bound to a value by *assigning* this value to the variable (see 4.15). Because of this we sometimes say that a variable that is not bound to any value has no assigned value. Assignment is in fact the only way by which a variable, which is not an argument of a function, can be bound to a value. After a variable has been bound to a value an assignment can also be used to bind the variable to another value.

A special class of variables is the class of *arguments* of functions. They behave similarly to other variables, except they are bound to the value of the actual arguments upon a function call (see 4.11).

Each variable has a name that is also called its *identifier*. This is because in a given scope an identifier identifies a unique variable (see 4.6). A *scope* is a lexical part of a program text. There is the *global scope* that encloses the entire program text, and there are local scopes that range from the `function` keyword, denoting the beginning of a function definition, to the corresponding `end` keyword. A *local scope* introduces new variables, whose identifiers are given in the formal argument list and the `local` declaration of the function (see 4.23). Usage of an identifier in a program text refers to the variable in the innermost scope that has this identifier as its name. Because this mapping from identifiers to variables is done when the program is read, not when it is executed, GAP is said to have *lexical scoping*. The following example shows how one identifier refers to different variables at different points in the program text.

```
g := 0;      # global variable g
x := function ( a, b, c )
  local  y;
  g := c;     # c refers to argument c of function x
  y := function ( y )
    local d, e, f;
    d := y;   # y refers to argument y of function y
    e := b;   # b refers to argument b of function x
    f := g;   # g refers to global variable g
    return d + e + f;
  end;
  return y( a ); # y refers to local y of function x
end;
```

It is important to note that the concept of a variable in GAP is quite different from the concept of a variable in most compiled programming languages.

In those languages a variable denotes a block of memory. The value of the variable is stored in this block. So in those languages two variables can have the same value, but they can never have identical values, because they denote different blocks of memory. Note that some languages have the concept of a reference argument. It seems as if such an argument and the variable used in the actual function call have the same value, since changing the argument's value also changes the value of the variable used in the actual function call. But this is not so; the reference argument is actually a pointer to the variable used in the actual function call, and it is the compiler that inserts enough magic to make the pointer invisible. In order for this to work the compiler needs enough information to compute the amount of memory needed for each variable in a program, which is readily available in the declarations.

In GAP on the other hand each variable just points to a value, and different variables can share the same value.

4.8.1 IsBound (for a global variable)

▷ `IsBound(`*`ident`*`)` (function)

`IsBound` returns `true` if the variable *`ident`* points to a value, and `false` otherwise.

For records and lists `IsBound` can be used to check whether components or entries, respectively, are bound (see Chapters 29 and 21).

4.8.2 Unbind (unbind a variable)

▷ `Unbind(`*`ident`*`)` (function)

deletes the identifier *`ident`*. If there is no other variable pointing to the same value as *`ident`* was, this value will be removed by the next garbage collection. Therefore `Unbind` can be used to get rid of unwanted large objects.

For records and lists `Unbind` can be used to delete components or entries, respectively (see Chapters 29 and 21).

4.9 More About Global Variables

The vast majority of variables in GAP are defined at the outer level (the global scope). They are used to access functions and other objects created either in the GAP library or packages or in the user's code.

Note that for packages there is a mechanism to implement package local namespaces on top of this global namespace. See Section 4.10 for details.

Certain special facilities are provided for manipulating global variables which are not available for other types of variable (such as local variables or function arguments).

First, such variables may be marked *read-only*. In which case attempts to change them will fail. Most of the global variables defined in the GAP library are so marked.

Second, a group of functions are supplied for accessing and altering the values assigned to global variables. Use of these functions differs from the use of assignment, `Unbind` (4.8.2) and `IsBound` (4.8.1) statements, in two ways. First, these functions always affect global variables, even if local variables of the same names exist. Second, the variable names are passed as strings, rather than being written directly into the statements.

Note that the functions `NamesGVars` (4.9.8), `NamesSystemGVars` (4.9.9), `NamesUserGVars` (4.9.10), and `TemporaryGlobalVarName` (4.9.11) deal with the *global namespace*.

4.9.1 IsReadOnlyGlobal

▷ `IsReadOnlyGlobal(`*`name`*`)` (function)

returns `true` if the global variable named by the string *`name`* is read-only and `false` otherwise (the default).

4.9.2 MakeReadOnlyGlobal

▷ `MakeReadOnlyGlobal(`*name*`)` (function)

marks the global variable named by the string *name* as read-only.
A warning is given if *name* has no value bound to it or if it is already read-only.

4.9.3 MakeReadWriteGlobal

▷ `MakeReadWriteGlobal(`*name*`)` (function)

marks the global variable named by the string *name* as read-write.
A warning is given if *name* is already read-write.

Example

```
gap> xx := 17;
17
gap> IsReadOnlyGlobal("xx");
false
gap> xx := 15;
15
gap> MakeReadOnlyGlobal("xx");
gap> xx := 16;
Variable: 'xx' is read only
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' after making it writable to continue
brk> quit;
gap> IsReadOnlyGlobal("xx");
true
gap> MakeReadWriteGlobal("xx");
gap> xx := 16;
16
gap> IsReadOnlyGlobal("xx");
false
```

4.9.4 ValueGlobal

▷ `ValueGlobal(`*name*`)` (function)

returns the value currently bound to the global variable named by the string *name*. An error is raised if no value is currently bound.

4.9.5 IsBoundGlobal

▷ `IsBoundGlobal(`*name*`)` (function)

returns `true` if a value currently bound to the global variable named by the string *name* and `false` otherwise.

4.9.6 UnbindGlobal

▷ `UnbindGlobal(`*`name`*`)` (function)

removes any value currently bound to the global variable named by the string *name*. Nothing is returned.

A warning is given if *name* was not bound. The global variable named by *name* must be writable, otherwise an error is raised.

4.9.7 BindGlobal

▷ `BindGlobal(`*`name, val`*`)` (function)

sets the global variable named by the string *name* to the value *val*, provided it is writable, and makes it read-only. If *name* already had a value, a warning message is printed.

This is intended to be the normal way to create and set "official" global variables (such as operations and filters).

Caution should be exercised in using these functions, especially `BindGlobal` and `UnbindGlobal` (4.9.6) as unexpected changes in global variables can be very confusing for the user.

Example

```
gap> xx := 16;
16
gap> IsReadOnlyGlobal("xx");
false
gap> ValueGlobal("xx");
16
gap> IsBoundGlobal("xx");
true
gap> BindGlobal("xx",17);
#W BIND_GLOBAL: variable ‘xx’ already has a value
gap> xx;
17
gap> IsReadOnlyGlobal("xx");
true
gap> MakeReadWriteGlobal("xx");
gap> Unbind(xx);
```

4.9.8 NamesGVars

▷ `NamesGVars()` (function)

This function returns an immutable (see 12.6) sorted (see 21.19) list of all the global variable names known to the system. This includes names of variables which were bound but have now been unbound and some other names which have never been bound but have become known to the system by various routes.

4.9.9 NamesSystemGVars

▷ `NamesSystemGVars()` (function)

This function returns an immutable sorted list of all the global variable names created by the GAP library when GAP was started.

4.9.10 NamesUserGVars

▷ `NamesUserGVars()` (function)

This function returns an immutable sorted list of the global variable names created since the library was read, to which a value is currently bound.

4.9.11 TemporaryGlobalVarName

▷ `TemporaryGlobalVarName(`*`[prefix]`*`)` (function)

returns a string that can be used as the name of a global variable that is not bound at the time when `TemporaryGlobalVarName` is called. The optional argument *`prefix`* can specify a string with which the name of the global variable starts.

4.10 Namespaces for GAP packages

As mentioned in Section 4.9 above all global variables share a common namespace. This can relatively easily lead to name clashes, in particular when many GAP packages are loaded at the same time. To give package code a way to have a package local namespace without breaking backward compatibility of the GAP language, the following simple rule has been devised:

If in package code a global variable that ends with an "at"-character `@` is accessed in any way, the name of the package is appended before accessing it. Here, "package code" refers to everything which is read with `ReadPackage` (76.3.1). As the name of the package the entry `PackageName` in its `PackageInfo.g` file is taken. As for all identifiers, this name is case sensitive.

For example, if the following is done in the code of a package with name `xYz`:

Example

```
gap> a@ := 12;
```

Then actually the global variable `a@xYz` is assigned. Further accesses to `a@` within the package code will all be redirected to `a@xYz`. This includes all the functions described in Section 4.9 and indeed all the functions described Section 79.18 like for example `DeclareCategory` (79.18.1). Note that from code in the same package it is still possible to access the same global variable via `a@xYz` explicitly.

All other code outside the package as well as interactive user input that wants to refer to that variable `a@xYz` must do so explicitly by using `a@xYz`.

Since in earlier releases of GAP the "at"-character `@` was not a legal character (without using backslashes), this small extension of the language does not break any old code.

4.11 Function Calls

4.11.1 Function Call With Arguments

`function-var``( [`*`arg-expr`*`[, `*`arg-expr`*`, ...]]  )`

The function call has the effect of calling the function *function-var*. The precise semantics are as follows.

First GAP evaluates the *function-var*. Usually *function-var* is a variable, and GAP does nothing more than taking the value of this variable. It is allowed though that *function-var* is a more complex expression, such as a reference to an element of a list (see Chapter 21) *list-var*[*int-expr*], or to a component of a record (see Chapter 29) *record-var*.*ident*. In any case GAP tests whether the value is a function. If it is not, GAP signals an error.

Next GAP checks that the number of actual arguments *arg-expr*s agrees with the number of *formal arguments* as given in the function definition. If they do not agree GAP signals an error. An exception is the case when there is exactly one formal argument with the name `arg`, in which case any number of actual arguments is allowed (see 4.23 for examples).

Now GAP allocates for each formal argument and for each *formal local* (that is, the identifiers in the `local` declaration) a new variable. Remember that a variable is a location in a GAP program that points to a value. Thus for each formal argument and for each formal local such a location is allocated.

Next the arguments *arg-expr*s are evaluated, and the values are assigned to the newly created variables corresponding to the formal arguments. Of course the first value is assigned to the new variable corresponding to the first formal argument, the second value is assigned to the new variable corresponding to the second formal argument, and so on. However, GAP does not make any guarantee about the order in which the arguments are evaluated. They might be evaluated left to right, right to left, or in any other order, but each argument is evaluated once. An exception again occurs if the function has only one formal argument with the name `arg`. In this case the values of all the actual arguments are stored in a list and this list is assigned to the new variable corresponding to the formal argument `arg`.

The new variables corresponding to the formal locals are initially not bound to any value. So trying to evaluate those variables before something has been assigned to them will signal an error.

Now the body of the function, which is a statement, is executed. If the identifier of one of the formal arguments or formal locals appears in the body of the function it refers to the new variable that was allocated for this formal argument or formal local, and evaluates to the value of this variable.

If during the execution of the body of the function a `return` statement with an expression (see 4.24) is executed, execution of the body is terminated and the value of the function call is the value of the expression of the `return`. If during the execution of the body a `return` statement without an expression is executed, execution of the body is terminated and the function call does not produce a value, in which case we call this call a procedure call (see 4.16). If the execution of the body completes without execution of a `return` statement, the function call again produces no value, and again we talk about a procedure call.

Example

```
gap> Fibonacci( 11 );
89
```

The above example shows a call to the function `Fibonacci` (16.3.1) with actual argument `11`, the following one shows a call to the operation `RightCosets` (39.7.2) where the second actual argument is another function call.

Example

```
gap> RightCosets( G, Intersection( U, V ) );;
```

4.11.2 Function Call With Options

function-var`(` *arg-expr*`[,` *arg-expr*`, ...][ :  [` *option-expr* `[,`*option-expr*`, ....]]])`

As well as passing arguments to a function, providing the mathematical input to its calculation, it is sometimes useful to supply "hints" suggesting to GAP how the desired result may be computed more quickly, or specifying a level of tolerance for random errors in a Monte Carlo algorithm.

Such hints may be supplied to a function-call *and to all subsidiary functions called from that call* using the options mechanism. Options are separated from the actual arguments by a colon `:` and have much the same syntax as the components of a record expression. The one exception to this is that a component name may appear without a value, in which case the value `true` is silently inserted.

The following example shows a call to `Size` (30.4.6) passing the options `hard` (with the value `true`) and `tcselection` (with the string `"external"` as value).

Example

```
gap> Size( fpgrp : hard, tcselection := "external" );
```

Options supplied with function calls in this way are passed down using the global options stack described in chapter 8, so that the call above is exactly equivalent to

Example

```
gap> PushOptions( rec( hard := true, tcselection := "external") );
gap> Size( fpgrp );
gap> PopOptions( );
```

Note that any option may be passed with any function, whether or not it has any actual meaning for that function, or any function called by it. The system provides no safeguard against misspelled option names.

4.12 Comparisons

left-expr `=` *right-expr*

left-expr `<>` *right-expr*

The operator = tests for equality of its two operands and evaluates to `true` if they are equal and to `false` otherwise. Likewise <> tests for inequality of its two operands. For each type of objects the definition of equality is given in the respective chapter. Objects in different families (see 13.1) are never equal, i.e., = evaluates in this case to `false`, and <> evaluates to `true`.

left-expr `<` *right-expr*

left-expr `>` *right-expr*

left-expr `<=` *right-expr*

left-expr `>=` *right-expr*

< denotes less than, <= less than or equal, > greater than, and >= greater than or equal of its two operands. For each kind of objects the definition of the ordering is given in the respective chapter.

Note that < implements a *total ordering* of objects (which can be used for example to sort a list of elements). Therefore in general < will not be compatible with any inclusion relation (which can be tested using `IsSubset` (30.5.1)). (For example, it is possible to compare permutation groups with < in a total ordering of all permutation groups, but this ordering is not compatible with the relation of being a subgroup.)

Only for the following kinds of objects, an ordering via < of objects in *different* families (see 13.1) is supported. Rationals (see `IsRat` (17.2.1)) are smallest, next are cyclotomics (see `IsCyclotomic` (18.1.3)), followed by finite field elements (see `IsFFE` (59.1.1)); finite field elements in different characteristics are compared via their characteristics, next are permutations (see `IsPerm` (42.1.1)), followed by the boolean values `true`, `false`, and `fail` (see `IsBool` (20.1.1)), characters (such as `{}a{'}'`, see `IsChar` (27.1.1)), and lists (see `IsList` (21.1.1)) are largest; note that two lists can be compared with < if and only if their elements are again objects that can be compared with <.

For other objects, GAP does *not* provide an ordering via <. The reason for this is that a total ordering of all GAP objects would be hard to maintain when new kinds of objects are introduced, and such a total ordering is hardly used in its full generality.

However, for objects in the filters listed above, the ordering via < has turned out to be useful. For example, one can form *sorted lists* containing integers and nested lists of integers, and then search in them using `PositionSorted` (see 21.16).

Of course it would in principle be possible to define an ordering via < also for certain other objects, by installing appropriate methods for the operation `\<`. But this may lead to problems at least as soon as one loads GAP code in which the same is done, under the assumption that one is completely free to define an ordering via < for other objects than the ones for which the "official" GAP provides already an ordering via <.

Comparison operators, including the operator `in` (see 21.8), are not associative, Hence it is not allowed to write `a = b <> c = d`, you must use `(a = b) <> (c = d)` instead. The comparison operators have higher precedence than the logical operators (see 20.4), but lower precedence than the arithmetic operators (see 4.13). Thus, for instance, `a * b = c and d` is interpreted as `((a * b) = c) and d)`.

The following example shows a comparison where the left operand is an expression.

Example

```
gap> 2 * 2 + 9 = Fibonacci(7);
true
```

For the underlying operations of the operators introduced above, see 31.11.

4.13 Arithmetic Operators

`+` *right-expr*
`-` *right-expr*
left-expr `+` *right-expr*
left-expr `-` *right-expr*
left-expr `*` *right-expr*
left-expr `/` *right-expr*
left-expr `mod` *right-expr*
left-expr `^` *right-expr*

The arithmetic operators are `+`, `-`, `*`, `/`, `mod`, and `^`. The meanings (semantics) of those operators generally depend on the types of the operands involved, and they are defined in the various chapters describing the types. However basically the meanings are as follows.

`a + b` denotes the addition of additive elements `a` and `b`.

`a - b` denotes the addition of `a` and the additive inverse of `b`.

`a * b` denotes the multiplication of multiplicative elements `a` and `b`.

`a / b` denotes the multiplication of `a` with the multiplicative inverse of `b`.

`a mod b`, for integer or rational left operand `a` and for non-zero integer right operand `b`, is defined as follows. If `a` and `b` are both integers, `a mod b` is the integer `r` in the integer range `0 .. |b| - 1` satisfying `a = r + bq`, for some integer `q` (where the operations occurring have their usual meaning over the integers, of course).

If `a` is a rational number and `b` is a non-zero integer, and `a = m / n` where `m` and `n` are coprime integers with `n` positive, then `a mod b` is the integer `r` in the integer range `0 .. |b| - 1` such that `m` is congruent to `rn` modulo `b`, and `r` is called the "modular remainder" of `a` modulo `b`. Also, `1 / n mod b` is called the "modular inverse" of `n` modulo `b`. (A pair of integers is said to be *coprime* (or *relatively prime*) if their greatest common divisor is 1.)

With the above definition, `4 / 6 mod 32` equals `2 / 3 mod 32` and hence exists (and is equal to 22), despite the fact that 6 has no inverse modulo 32.

Note: For rational `a`, `a mod b` could have been defined to be the non-negative rational `c` less than `|b|` such that `a - c` is a multiple of `b`. However this definition is seldom useful and *not* the one chosen for GAP.

`+` and `-` can also be used as unary operations. The unary `+` is ignored. The unary `-` returns the additive inverse of its operand; over the integers it is equivalent to multiplication by `-1`.

`^` denotes powering of a multiplicative element if the right operand is an integer, and is also used to denote the action of a group element on a point of a set if the right operand is a group element.

The *precedence* of those operators is as follows. The powering operator `^` has the highest precedence, followed by the unary operators `+` and `-`, which are followed by the multiplicative operators `*`, `/`, and `mod`, and the additive binary operators `+` and `-` have the lowest precedence. That means that the expression `-2 ^ -2 * 3 + 1` is interpreted as `(-(2 ^ (-2)) * 3) + 1`. If in doubt use parentheses to clarify your intention.

The *associativity* of the arithmetic operators is as follows. `^` is not associative, i.e., it is invalid to write `2^3^4`, use parentheses to clarify whether you mean `(2^3)^4` or `2^(3^4)`. The unary operators `+` and `-` are right associative, because they are written to the left of their operands. `*`, `/`, `mod`, `+`, and `-` are all left associative, i.e., `1-2-3` is interpreted as `(1-2)-3` not as `1-(2-3)`. Again, if in doubt use parentheses to clarify your intentions.

The arithmetic operators have higher precedence than the comparison operators (see 4.12 and 30.6) and the logical operators (see 20.4). Thus, for example, `a * b = c and d` is interpreted, `((a * b) = c) and d`.

Example

```
gap> 2 * 2 + 9;  # a very simple arithmetic expression
13
```

For other arithmetic operations, and for the underlying operations of the operators introduced above, see 31.12.

4.14 Statements

Assignments (see 4.15), Procedure calls (see 4.16), `if` statements (see 4.17), `while` (see 4.18), `repeat` (see 4.19) and `for` loops (see 4.20), and the `return` statement (see 4.24) are called *statements*. They can be entered interactively or be part of a function definition. Every statement must be terminated by a semicolon.

Statements, unlike expressions, have no value. They are executed only to produce an effect. For example an assignment has the effect of assigning a value to a variable, a `for` loop has the effect of

executing a statement sequence for all elements in a list and so on. We will talk about *evaluation* of expressions but about *execution* of statements to emphasize this difference.

Using expressions as statements is treated as syntax error.

Example

```
gap> i := 7;;
gap> if i <> 0 then k = 16/i; fi;
Syntax error: := expected
if i <> 0 then k = 16/i; fi;
                 ^
gap>
```

As you can see from the example this warning does in particular address those users who are used to languages where = instead of := denotes assignment.

Empty statements are permitted and have no effect.

A sequence of one or more statements is a *statement sequence*, and may occur everywhere instead of a single statement. Each construct is terminated by a keyword. The simplest statement sequence is a single semicolon, which can be used as an empty statement sequence. In fact an empty statement sequence as in `for i in [ 1 .. 2 ] do od` is also permitted and is silently translated into the sequence containing just a semicolon.

4.15 Assignments

var `:=` *expr*`;`

The *assignment* has the effect of assigning the value of the expressions *expr* to the variable *var*.

The variable *var* may be an ordinary variable (see 4.8), a list element selection *list-var*`[`*int-expr*`]` (see 21.4) or a record component selection *record-var*`.`*ident* (see 29.3). Since a list element or a record component may itself be a list or a record the left hand side of an assignment may be arbitrarily complex.

Note that variables do not have a type. Thus any value may be assigned to any variable. For example a variable with an integer value may be assigned a permutation or a list or anything else.

Example

```
gap> data:= rec( numbers:= [ 1, 2, 3 ] );
rec( numbers := [ 1, 2, 3 ] )
gap> data.string:= "string";; data;
rec( numbers := [ 1, 2, 3 ], string := "string" )
gap> data.numbers[2]:= 4;; data;
rec( numbers := [ 1, 4, 3 ], string := "string" )
```

If the expression *expr* is a function call then this function must return a value. If the function does not return a value an error is signalled and you enter a break loop (see 6.4). As usual you can leave the break loop with `quit;`. If you enter `return` *return-expr*`;` the value of the expression *return-expr* is assigned to the variable, and execution continues after the assignment.

Example

```
gap> f1:= function( x ) Print( "value: ", x, "\n" ); end;;
gap> f2:= function( x ) return f1( x ); end;;
gap> f2( 4 );
value: 4
Function Calls: <func> must return a value at
```

```
return f1( x );
 called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can supply one by 'return <value>;' to continue
brk> return "hello";
"hello"
```

In the above example, the function `f2` calls `f1` with argument `4`, and since `f1` does not return a value (but only prints a line "`value: ...`"), the `return` statement of `f2` cannot be executed. The error message says that it is possible to return an appropriate value, and the returned string `"hello"` is used by `f2` instead of the missing return value of `f1`.

4.16 Procedure Calls

procedure-var`( [`*arg-expr* `[,`*arg-expr*`, ...]] );`

The *procedure call* has the effect of calling the procedure *procedure-var*. A procedure call is done exactly like a function call (see 4.11). The distinction between functions and procedures is only for the sake of the discussion, GAP does not distinguish between them. So we state the following conventions.

A *function* does return a value but does not produce a side effect. As a convention the name of a function is a noun, denoting what the function returns, e.g., `"Length"`, `"Concatenation"` and `"Order"`.

A *procedure* is a function that does not return a value but produces some effect. Procedures are called only for this effect. As a convention the name of a procedure is a verb, denoting what the procedure does, e.g., `"Print"`, `"Append"` and `"Sort"`.

Example

```
gap> Read( "myfile.g" );   # a call to the procedure Read
gap> l := [ 1, 2 ];;
gap> Append( l, [3,4,5] );  # a call to the procedure Append
```

There are a few exceptions of GAP functions that do both return a value and produce some effect. An example is `Sortex` (21.18.3) which sorts a list and returns the corresponding permutation of the entries.

4.17 If

`if` *bool-expr1* `then` *statements1* `{ elif` *bool-expr2* `then` *statements2* `}[ else` *statements3* `] fi;`

The `if` statement allows one to execute statements depending on the value of some boolean expression. The execution is done as follows.

First the expression *bool-expr1* following the `if` is evaluated. If it evaluates to `true` the statement sequence *statements1* after the first `then` is executed, and the execution of the `if` statement is complete.

Otherwise the expressions *bool-expr2* following the `elif` are evaluated in turn. There may be any number of `elif` parts, possibly none at all. As soon as an expression evaluates to `true` the

corresponding statement sequence *statements2* is executed and execution of the `if` statement is complete.

If the `if` expression and all, if any, `elif` expressions evaluate to `false` and there is an `else` part, which is optional, its statement sequence *statements3* is executed and the execution of the `if` statement is complete. If there is no `else` part the `if` statement is complete without executing any statement sequence.

Since the `if` statement is terminated by the `fi` keyword there is no question where an `else` part belongs, i.e., GAP has no "dangling else". In

```
if expr1 then if expr2 then stats1 else stats2 fi; fi;
```

the `else` part belongs to the second `if` statement, whereas in

```
if expr1 then if expr2 then stats1 fi; else stats2 fi;
```

the `else` part belongs to the first `if` statement.

Since an `if` statement is not an expression it is not possible to write

```
abs := if x > 0 then x; else -x; fi;
```

which would, even if legal syntax, be meaningless, since the `if` statement does not produce a value that could be assigned to `abs`.

If one of the expressions *bool-expr1*, *bool-expr2* is evaluated and its value is neither `true` nor `false` an error is signalled and a break loop (see 6.4) is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `true`. Likewise, if you enter `return false;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `false`.

Example

```
gap> i := 10;;
gap> if 0 < i then
>    s := 1;
>  elif i < 0 then
>    s := -1;
>  else
>    s := 0;
>  fi;
gap> s;  # the sign of i
1
```

4.18 While

`while` *bool-expr* `do` *statements* `od;`

The `while` loop executes the statement sequence *statements* while the condition *bool-expr* evaluates to `true`.

First *bool-expr* is evaluated. If it evaluates to `false` execution of the `while` loop terminates and the statement immediately following the `while` loop is executed next. Otherwise if it evaluates to `true` the *statements* are executed and the whole process begins again.

The difference between the `while` loop and the `repeat until` loop (see 4.19) is that the *`statements`* in the `repeat until` loop are executed at least once, while the *`statements`* in the `while` loop are not executed at all if *`bool-expr`* is `false` at the first iteration.

If *`bool-expr`* does not evaluate to `true` or `false` an error is signalled and a break loop (see 6.4) is entered. As usual you can leave the break loop with `quit;`. If you enter `return false;`, execution continues with the next statement immediately following the `while` loop. If you enter `return true;`, execution continues at *`statements`*, after which the next evaluation of *`bool-expr`* may cause another error.

The following example shows a `while` loop that sums up the squares $1^2, 2^2, \ldots$ until the sum exceeds 200.

Example

```
gap> i := 0;; s := 0;;
gap> while s <= 200 do
>    i := i + 1; s := s + i^2;
>  od;
gap> s;
204
```

A `while` loop may be left prematurely using `break`, see 4.21.

4.19 Repeat

`repeat` *`statements`* `until` *`bool-expr`*`;`

The `repeat` loop executes the statement sequence *`statements`* until the condition *`bool-expr`* evaluates to `true`.

First *`statements`* are executed. Then *`bool-expr`* is evaluated. If it evaluates to `true` the `repeat` loop terminates and the statement immediately following the `repeat` loop is executed next. Otherwise if it evaluates to `false` the whole process begins again with the execution of the *`statements`*.

The difference between the `while` loop (see 4.18) and the `repeat until` loop is that the *`statements`* in the `repeat until` loop are executed at least once, while the *`statements`* in the `while` loop are not executed at all if *`bool-expr`* is `false` at the first iteration.

If *`bool-expr`* does not evaluate to `true` or `false` an error is signalled and a break loop (see 6.4) is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution continues with the next statement immediately following the `repeat` loop. If you enter `return false;`, execution continues at *`statements`*, after which the next evaluation of *`bool-expr`* may cause another error.

The `repeat` loop in the following example has the same purpose as the `while` loop in the preceding example, namely to sum up the squares $1^2, 2^2, \ldots$ until the sum exceeds 200.

Example

```
gap> i := 0;; s := 0;;
gap> repeat
>    i := i + 1; s := s + i^2;
>  until s > 200;
gap> s;
204
```

A `repeat` loop may be left prematurely using `break`, see 4.21.

4.20 For

`for` *simple-var* `in` *list-expr* `do` *statements* `od;`

The `for` loop executes the statement sequence *statements* for every element of the list *list-expr*.

The statement sequence *statements* is first executed with *simple-var* bound to the first element of the list *list-expr*, then with *simple-var* bound to the second element of *list-expr* and so on. *simple-var* must be a simple variable, it must not be a list element selection *list-var*`[`*int-expr*`]` or a record component selection *record-var*`.`*ident*.

The execution of the `for` loop over a list is exactly equivalent to the following `while` loop.

```
loop_list := list;
loop_index := 1;
while loop_index <= Length(loop_list) do
  variable := loop_list[loop_index];
  statements
  loop_index := loop_index + 1;
od;
```

with the exception that "loop_list" and "loop_index" are different variables for each `for` loop, i.e., these variables of different `for` loops do not interfere with each other.

The list *list-expr* is very often a range (see 21.22).

`for` *variable* `in [`*from*`..`*to*`] do` *statements* `od;`

corresponds to the more common

`for` *variable* `from` *from* `to` *to* `do` *statements* `od;`

in other programming languages.

Example

```
gap> s := 0;;
gap> for i in [1..100] do
>    s := s + i;
> od;
gap> s;
5050
```

Note in the following example how the modification of the *list* in the loop body causes the loop body also to be executed for the new values.

Example

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i in l do
>    Print( i, " " );
>    if i mod 2 = 0 then Add( l, 3 * i / 2 ); fi;
> od; Print( "\n" );
1 2 3 4 5 6 3 6 9 9
gap> l;
[ 1, 2, 3, 4, 5, 6, 3, 6, 9, 9 ]
```

Note in the following example that the modification of the *variable* that holds the list has no influence on the loop.

Example

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i in l do
>    Print( i, " " );
>    l := [];
> od; Print( "\n" );
1 2 3 4 5 6
gap> l;
[  ]
```

`for` *variable* `in` *iterator* `do` *statements* `od;`

It is also possible to have a `for`-loop run over an iterator (see 30.8). In this case the `for`-loop is equivalent to

```
while not IsDoneIterator(iterator) do
  variable := NextIterator(iterator)
  statements
od;
```

`for` *variable* `in` *object* `do` *statements* `od;`

Finally, if an object *object* which is not a list or an iterator appears in a `for`-loop, then GAP will attempt to evaluate the function call `Iterator(`*object*`)`. If this is successful then the loop is taken to run over the iterator returned.

Example

```
gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
Group([ (1,2,3,4,5), (1,2)(3,4)(5,6) ])
gap> count := 0;; sumord := 0;;
gap> for x in g do
> count := count + 1; sumord := sumord + Order(x); od;
gap> count;
120
gap> sumord;
471
```

The effect of

`for` *variable* `in` *domain* `do`

should thus normally be the same as

`for` *variable* `in AsList(`*domain*`) do`

but may use much less storage, as the iterator may be more compact than a list of all the elements. See 30.8 for details about iterators.

A `for` loop may be left prematurely using `break`, see 4.21. This combines especially well with a loop over an iterator, as a way of searching through a domain for an element with some useful property.

4.21 Break

`break;`

The statement `break;` causes an immediate exit from the innermost loop enclosing it.

Example

```
gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
Group([ (1,2,3,4,5), (1,2)(3,4)(5,6) ])
gap> for x in g do
> if Order(x) = 3 then
> break;
> fi; od;
gap> x;
(1,4,3)(2,6,5)
```

It is an error to use this statement other than inside a loop.

Example

```
gap> break;
Error, A break statement can only appear inside a loop
not in any function
```

4.22 Continue

`continue;`

The statement `continue;` causes the rest of the current iteration of the innermost loop enclosing it to be skipped.

Example

```
gap> g := Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> for x in g do
> if Order(x) = 3 then
> continue;
> fi; Print(x,"\n"); od;
()
(2,3)
(1,3)
(1,2)
```

It is an error to use this statement other than inside a loop.

Example

```
gap> continue;
Error, A continue statement can only appear inside a loop
not in any function
```

4.23 Function

`function( [` *arg-ident* `{, ` *arg-ident*`} ] )`
`  [local` *loc-ident* `{, ` *loc-ident*`} ; ]`
statements
`end`

A function is in fact a literal and not a statement. Such a function literal can be assigned to a variable or to a list element or a record component. Later this function can be called as described in 4.11.

The following is an example of a function definition. It is a function to compute values of the Fibonacci sequence (see Fibonacci (16.3.1)).

Example

```
gap> fib := function ( n )
>     local f1, f2, f3, i;
>     f1 := 1; f2 := 1;
>     for i in [3..n] do
>       f3 := f1 + f2;
>       f1 := f2;
>       f2 := f3;
>     od;
>     return f2;
>   end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Because for each of the formal arguments *arg-ident* and for each of the formal locals *loc-ident* a new variable is allocated when the function is called (see 4.11), it is possible that a function calls itself. This is usually called *recursion*. The following is a recursive function that computes values of the Fibonacci sequence.

Example

```
gap> fib := function ( n )
>     if n < 3 then
>       return 1;
>     else
>       return fib(n-1) + fib(n-2);
>     fi;
>   end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Note that the recursive version needs 2 * fib(*n*)-1 steps to compute fib(*n*), while the iterative version of fib needs only *n*-2 steps. Both are not optimal however, the library function Fibonacci (16.3.1) only needs about Log(*n*) steps.

As noted in Section 4.11, the case where a function is defined with exactly one formal argument with the name arg, is special. It provides a way of defining a function with a variable number of arguments; the values of all the actual arguments are stored in a list and this list is assigned to the new variable corresponding to the formal argument arg. There are two typical scenarios for wanting such a possibility: having optional arguments and having any number of arguments.

The following example shows one way that the function Position (21.16.1) might be encoded and demonstrates the "optional argument" scenario.

Example

```
gap> position := function ( arg )
>     local list, obj, pos;
>     list := arg[1];
>     obj  := arg[2];
>     if 2 = Length(arg) then
>       pos := 0;
>     else
>       pos := arg[3];
```

```
>     fi;
>     repeat
>       pos := pos + 1;
>       if pos > Length(list) then
>         return fail;
>       fi;
>     until list[pos] = obj;
>     return pos;
>    end;
function( arg ) ... end
gap> position([1, 4, 2], 4);
2
gap> position([1, 4, 2], 3);
fail
gap> position([1, 4, 2], 4, 2);
fail
```

The following example demonstrates the "any number of arguments" scenario.

Example

```
gap> sum := function ( arg )
>     local total, x;
>     total := 0;
>     for x in arg do
>       total := total + x;
>     od;
>     return total;
>    end;
function( arg ) ... end
gap> sum(1, 2, 3);
6
gap> sum(1, 2, 3, 4);
10
gap> sum();
0
```

The user should compare the above with the GAP function `Sum` (21.20.26) which, for example, may take a list argument and optionally an initial element (which zero should the sum of an empty list return?).

Note that if a function *f* is defined as above with the single formal argument `arg` then `NumberArgumentsFunction(`*f*`)` returns -1 (see `NumberArgumentsFunction` (5.1.2)).

The argument `arg` when used as the single argument name of some function *f* tells GAP that when it encounters *f* that it should form a list out of the arguments of *f*. What if one wishes to do the "opposite": tell GAP that a list should be "unwrapped" and passed as several arguments to a function. The function `CallFuncList` (5.2.1) is provided for this purpose.

Also see Chapter 5.

arg-ident -> expr

This is a shorthand for

`function ( `*arg-ident*` ) return `*expr*`; end.`

arg-ident must be a single identifier, i.e., it is not possible to write functions of several arguments this way. Also `arg` is not treated specially, so it is also impossible to write functions that take a variable number of arguments this way.

The following is an example of a typical use of such a function

Example

```
gap> Sum( List( [1..100], x -> x^2 ) );
338350
```

When the definition of a function *fun1* is evaluated inside another function *fun2*, GAP binds all the identifiers inside the function *fun1* that are identifiers of an argument or a local of *fun2* to the corresponding variable. This set of bindings is called the environment of the function *fun1*. When *fun1* is called, its body is executed in this environment. The following implementation of a simple stack uses this. Values can be pushed onto the stack and then later be popped off again. The interesting thing here is that the functions `push` and `pop` in the record returned by `Stack` access the local variable `stack` of `Stack`. When `Stack` is called, a new variable for the identifier `stack` is created. When the function definitions of `push` and `pop` are then evaluated (as part of the `return` statement) each reference to `stack` is bound to this new variable. Note also that the two stacks `A` and `B` do not interfere, because each call of `Stack` creates a new variable for `stack`.

Example

```
gap> Stack := function ()
>     local  stack;
>     stack := [];
>     return rec(
>       push := function ( value )
>         Add( stack, value );
>       end,
>       pop := function ()
>         local  value;
>         value := stack[Length(stack)];
>         Unbind( stack[Length(stack)] );
>         return value;
>       end
>     );
>  end;;
gap> A := Stack();;
gap> B := Stack();;
gap> A.push( 1 ); A.push( 2 ); A.push( 3 );
gap> B.push( 4 ); B.push( 5 ); B.push( 6 );
gap> A.pop(); A.pop(); A.pop();
3
2
1
gap> B.pop(); B.pop(); B.pop();
6
5
4
```

This feature should be used rarely, since its implementation in GAP is not very efficient.

4.24 Return (With or without Value)

`return;`

In this form `return` terminates the call of the innermost function that is currently executing, and control returns to the calling function. An error is signalled if no function is currently executing. No value is returned by the function.

`return` *expr*`;`

In this form `return` terminates the call of the innermost function that is currently executing, and returns the value of the expression *expr*. Control returns to the calling function. An error is signalled if no function is currently executing.

Both statements can also be used in break loops (see 6.4). `return;` has the effect that the computation continues where it was interrupted by an error or the user hitting CTRL-C. `return` *expr*`;` can be used to continue execution after an error. What happens with the value *expr* depends on the particular error.

For examples of `return` statements, see the functions `fib` and `Stack` in Section 4.23.

Chapter 5

Functions

The section 4.23 describes how to define a function. In this chapter we describe functions that give information about functions, and various utility functions used either when defining functions or calling functions.

5.1 Information about a function

5.1.1 NameFunction

▷ `NameFunction(`*`func`*`)` (function)

returns the name of a function. For operations, this is the name used in their declaration. For functions, this is the variable name they were first assigned to. (For some internal functions, this might be a name *different* from the name that is documented.) If no such name exists, the string `"unknown"` is returned.

Example

```
gap> NameFunction(SylowSubgroup);
"SylowSubgroup"
gap> Blubberflutsch:=x->x;;
gap> NameFunction(Blubberflutsch);
"Blubberflutsch"
gap> a:=Blubberflutsch;;
gap> NameFunction(a);
"Blubberflutsch"
gap> NameFunction(x->x);
"unknown"
gap> NameFunction(NameFunction);
"NameFunction"
```

5.1.2 NumberArgumentsFunction

▷ `NumberArgumentsFunction(`*`func`*`)` (function)

returns the number of arguments the function *`func`* accepts. For functions that use `arg` to take a variable number of arguments, as well as for operations, -1 is returned. For attributes, 1 is returned.

Example

```
gap> NumberArgumentsFunction(function(a,b,c,d,e,f,g,h,i,j,k)return 1;end);
11
gap> NumberArgumentsFunction(Size);
1
gap> NumberArgumentsFunction(IsCollsCollsElms);
3
gap> NumberArgumentsFunction(Sum);
-1
```

5.1.3 NamesLocalVariablesFunction

▷ NamesLocalVariablesFunction(*func*) (function)

returns a mutable list of strings; the first entries are the names of the arguments of the function *func*, in the same order as they were entered in the definition of *func*, and the remaining ones are the local variables as given in the `local` statement in *func*. (The number of arguments can be computed with `NumberArgumentsFunction` (5.1.2).)

Example

```
gap> NamesLocalVariablesFunction(function( a, b ) local c; return 1; end);
[ "a", "b", "c" ]
gap> NamesLocalVariablesFunction(function( arg ) local a; return 1; end);
[ "arg", "a" ]
gap> NamesLocalVariablesFunction( Size );
fail
```

5.1.4 FilenameFunc

▷ FilenameFunc(*func*) (function)

For a function *func*, `FilenameFunc` returns either `fail` or the absolute path of the file from which *func* has been read. The return value `fail` occurs if *func* is a compiled function or an operation. For functions that have been entered interactively, the string `"*stdin*"` is returned, see Section 9.5.

Example

```
gap> FilenameFunc( LEN_LIST );  # a kernel function
fail
gap> FilenameFunc( Size );      # an operation
fail
gap> FilenameFunc( x -> x^2 );  # an interactively entered function
"*stdin*"
gap> meth:= ApplicableMethod( Size, [ Group( () ) ] );;
gap> FilenameFunc( meth );
"... some path .../grpperm.gi"
```

5.1.5 StartlineFunc

▷ StartlineFunc(*func*) (function)

▷ EndlineFunc(*func*) (function)

Let *func* be a function. If FilenameFunc (5.1.4) returns fail for *func* then also StartlineFunc returns fail. If FilenameFunc (5.1.4) returns a filename for *func* then StartlineFunc returns the line number in this file where the definition of *func* starts.

EndlineFunc behaves similarly and returns the line number in this file where the definition of *func* ends.

Example

```
gap> meth:= ApplicableMethod( Size, [ Group( () ) ] );;
gap> FilenameFunc( meth );
"... some path ... gap4r5/lib/grpperm.gi"
gap> StartlineFunc( meth );
487
gap> EndlineFunc( meth );
487
```

5.1.6 PageSource

▷ PageSource(*func*) (function)

This shows the file containing the source code of the function or method *func* in a pager (see Pager (2.4.1)). The display starts at a line shortly before the code of *func*.

This function works if FilenameFunc(*func*) returns the name of a proper file. In that case this filename and the position of the function definition are also printed. Otherwise the function indicates that the source is not available (for example this happens for functions which are implemented in the GAP C-kernel).

Usage examples:

```
met := ApplicableMethod(\^, [(1,2),2743527]); PageSource(met);
PageSource(Combinations);
ct:=CharacterTable(Group((1,2,3)));
met := ApplicableMethod(Size,[ct]); PageSource(met);
```

5.2 Calling a function with a list argument that is interpreted as several arguments

5.2.1 CallFuncList

▷ CallFuncList(*func, args*) (operation)

returns the result, when calling function *func* with the arguments given in the list *args*, i.e. *args* is "unwrapped" so that *args* appears as several arguments to *func*.

Example

```
gap> CallFuncList(\+, [6, 7]);
13
gap> #is equivalent to:
gap> \+(6, 7);
13
```

A more useful application of `CallFuncList` is for a function `g` that is called in the body of a function `f` with (a sublist of) the arguments of `f`, where `f` has been defined with a single formal argument `arg` (see 4.23), as in the following code fragment.

Example

```
f := function ( arg )
       CallFuncList(g, arg);
       ...
     end;
```

In the body of `f` the several arguments passed to `f` become a list `arg`. If `g` were called instead via `g( arg )` then `g` would see a single list argument, so that `g` would, in general, have to "unwrap" the passed list. The following (not particularly useful) example demonstrates both described possibilities for the call to `g`.

Example

```
gap> PrintNumberFromDigits := function ( arg )
>     CallFuncList( Print, arg );
>     Print( "\n" );
>    end;
function( arg ) ... end
gap> PrintNumberFromDigits( 1, 9, 7, 3, 2 );
19732
gap> PrintDigits := function ( arg )
>     Print( arg );
>     Print( "\n" );
>    end;
function( arg ) ... end
gap> PrintDigits( 1, 9, 7, 3, 2 );
[ 1, 9, 7, 3, 2 ]
```

5.3 Functions that do nothing

The following functions return fixed results (or just their own argument). They can be useful in places when the syntax requires a function, but actually no functionality is required. So `ReturnTrue` (5.3.1) is often used as family predicate in `InstallMethod` (78.2.1).

5.3.1 ReturnTrue

▷ `ReturnTrue(...)` (function)

This function takes any number of arguments, and always returns `true`.

Example

```
gap> f:=ReturnTrue;
function( arg ) ... end
gap> f();
true
gap> f(42);
true
```

5.3.2 ReturnFalse

▷ `ReturnFalse(...)` (function)

This function takes any number of arguments, and always returns `false`.

Example

```
gap> f:=ReturnFalse;
function( arg ) ... end
gap> f();
false
gap> f("any_string");
false
```

5.3.3 ReturnFail

▷ `ReturnFail(...)` (function)

This function takes any number of arguments, and always returns `fail`.

Example

```
gap> oops:=ReturnFail;
function( arg ) ... end
gap> oops();
fail
gap> oops(-42);
fail
```

5.3.4 IdFunc

▷ `IdFunc(`*obj*`)` (function)

returns *obj*.

Example

```
gap> id:=IdFunc;
function( object ) ... end
gap> id(42);
42
gap> f:=id(SymmetricGroup(3));
Sym( [ 1 .. 3 ] )
gap> s:=One(AutomorphismGroup(SymmetricGroup(3)));
IdentityMapping( Sym( [ 1 .. 3 ] ) )
gap> f=s;
false
```

5.4 Function Types

Functions are GAP objects and thus have categories and a family.

5.4.1 IsFunction

▷ IsFunction(*obj*) (Category)

is the category of functions.

Example

```
gap> IsFunction(x->x^2);
true
gap> IsFunction(Factorial);
true
gap> f:=One(AutomorphismGroup(SymmetricGroup(3)));
IdentityMapping( Sym( [ 1 .. 3 ] ) )
gap> IsFunction(f);
false
```

5.4.2 IsOperation

▷ IsOperation(*obj*) (Category)

is the category of operations. Every operation is a function, but not vice versa.

Example

```
gap> MinimalPolynomial;
<Operation "MinimalPolynomial">
gap> IsOperation(MinimalPolynomial);
true
gap> IsFunction(MinimalPolynomial);
true
gap> Factorial;
function( n ) ... end
gap> IsOperation(Factorial);
false
```

5.4.3 FunctionsFamily

▷ FunctionsFamily (global variable)

is the family of all functions.

5.5 Naming Conventions

The way functions are named in GAP might help to memorize or even guess names of library functions.

If a variable name consists of several words then the first letter of each word is capitalized.

If the first part of the name of a function is a verb then the function may modify its argument(s) but does not return anything, for example Append (21.4.5) appends the list given as second argument to the list given as first argument. Otherwise the function returns an object without changing the arguments, for example Concatenation (21.20.1) returns the concatenation of the lists given as arguments.

If the name of a function contains the word "Of" then the return value is thought of as information deduced from the arguments. Usually such functions are attributes (see 13.5). Examples are

GeneratorsOfGroup (39.2.4), which returns a list of generators for the group entered as argument, or DiagonalOfMat (24.12.1).

For the setter and tester functions of an attribute Attr the names SetAttr resp. HasAttr are available (see 13.5).

If the name of a function contains the word "By" then the return value is thought of as built in a certain way from the parts given as arguments. For example, creating a group as a factor group of a given group by a normal subgroup can be done by taking the image of NaturalHomomorphismByNormalSubgroup (39.18.1). Other examples of "By" functions are GroupHomomorphismByImages (40.1.1) and LaurentPolynomialByCoefficients (66.13.1).

Often such functions construct an algebraic structure given by its generators (for example, RingByGenerators (56.1.4)). In some cases, "By" may be replaced by "With" (like e.g. GroupWithGenerators (39.2.3)) or even both versions of the name may be used. The difference between StructByGenerators and StructWithGenerators is that the latter guarantees that the GeneratorsOfStruct value of the result is equal to the given set of generators (see 31.3).

If the name of a function has the form "AsSomething" then the return value is an object (usually a collection which has the same family of elements), which may, for example:

- know more about its own structure (and so support more operations) than its input (e.g. if the elements of the collection form a group, then this group can be constructed using AsGroup (39.2.5));
- discard its additional structure (e.g. AsList (30.3.8) applied to a group will return a list of its elements);
- contain all elements of the original object without duplicates (like e.g. AsSet (30.3.10) does if its argument is a list of elements from the same family);
- remain unchanged (like e.g. AsSemigroup (51.1.6) does if its argument is a group).

If Something and the argument of AsSomething are domains, some further rules apply as explained in **Tutorial: Changing the Structure**.

If the name of a function fun1 ends with "NC" then there is another function fun2 with the same name except that the NC is missing. NC stands for "no check". When fun2 is called then it checks whether its arguments are valid, and if so then it calls fun1. The functions SubgroupNC (39.3.1) and Subgroup (39.3.1) are a typical example.

The idea is that the possibly time consuming check of the arguments can be omitted if one is sure that they are unnecessary. For example, if an algorithm produces generators of the derived subgroup of a group then it is guaranteed that they lie in the original group; Subgroup (39.3.1) would check this, and SubgroupNC (39.3.1) omits the check.

Needless to say, all these rules are not followed slavishly, for example there is one operation Zero (31.10.3) instead of two operations ZeroOfElement and ZeroOfAdditiveGroup.

Chapter 6

Main Loop and Break Loop

This chapter is a first of a series of chapters that describe the interactive environment in which you use GAP.

6.1 Main Loop

The normal interaction with GAP happens in the so-called *read-eval-print* loop. This means that you type an input, GAP first reads it, evaluates it, and then shows the result. Note that the term *print* may be confusing since there is a GAP function called `Print` (6.3.4) (see 6.3) which is in fact *not* used in the read-eval-print loop, but traditions are hard to break. In the following, whenever we want to express that GAP places some characters on the standard output, we will say that GAP *shows* something.

The exact sequence in the read-eval-print loop is as follows.

To signal that it is ready to accept your input, GAP shows the *prompt* `gap>`. When you see this, you know that GAP is waiting for your input.

Note that every statement must be terminated by a semicolon. You must also enter RETURN (i.e., strike the RETURN key) before GAP starts to read and evaluate your input. (The RETURN key may actually be marked with the word ENTER and a returning arrow on your terminal.) Because GAP does not do anything until you enter RETURN, you can edit your input to fix typos and only when everything is correct enter RETURN and have GAP take a look at it (see 6.8). It is also possible to enter several statements as input on a single line. Of course each statement must be terminated by a semicolon.

It is absolutely acceptable to enter a single statement on several lines. When you have entered the beginning of a statement, but the statement is not yet complete, and you enter RETURN, GAP will show the *partial prompt* `>`. When you see this, you know that GAP is waiting for the rest of the statement. This happens also when you forget the semicolon `;` that terminates every GAP statement. Note that when RETURN has been entered and the current statement is not yet complete, GAP will already evaluate those parts of the input that are complete, for example function calls that appear as arguments in another function call which needs several input lines. So it may happen that one has to wait some time for the partial prompt.

When you enter RETURN, GAP first checks your input to see if it is syntactically correct (see Chapter 4 for the definition of syntactically correct). If it is not, GAP prints an error message of the following form

Example

```
gap> 1 * ;
Syntax error: expression expected
1 * ;
    ^
```

The first line tells you what is wrong about the input, in this case the * operator takes two expressions as operands, so obviously the right one is missing. If the input came from a file (see Read (9.7.1)), this line will also contain the filename and the line number. The second line is a copy of the input. And the third line contains a caret pointing to the place in the previous line where GAP realized that something is wrong. This need not be the exact place where the error is, but it is usually quite close.

Sometimes, you will also see a partial prompt after you have entered an input that is syntactically incorrect. This is because GAP is so confused by your input, that it thinks that there is still something to follow. In this case you should enter ;RETURN repeatedly, ignoring further error messages, until you see the full prompt again. When you see the full prompt, you know that GAP forgave you and is now ready to accept your next –hopefully correct– input.

If your input is syntactically correct, GAP evaluates or executes it, i.e., performs the required computations (see Chapter 4 for the definition of the evaluation).

If you do not see a prompt, you know that GAP is still working on your last input. Of course, you can *type ahead*, i.e., already start entering new input, but it will not be accepted by GAP until GAP has completed the ongoing computation.

When GAP is ready it will usually show the result of the computation, i.e., the value computed. Note that not all statements produce a value, for example, if you enter a `for` loop, nothing will be printed, because the `for` loop does not produce a value that could be shown.

Also sometimes you do not want to see the result. For example if you have computed a value and now want to assign the result to a variable, you probably do not want to see the value again. You can terminate statements by *two semicolons* to suppress showing the result.

If you have entered several statements on a single line GAP will first read, evaluate, and show the first one, then read, evaluate, and show the second one, and so on. This means that the second statement will not even be checked for syntactical correctness until GAP has completed the first computation.

After the result has been shown GAP will display another prompt, and wait for your next input. And the whole process starts all over again. Note that if you have entered several statements on a single line, a new prompt will only be printed after GAP has read, evaluated, and shown the last statement.

In each statement that you enter, the result of the previous statement that produced a value is available in the variable `last`. The next to previous result is available in `last2` and the result produced before that is available in `last3`.

Example

```
gap> 1;2;3;
1
2
3
gap> last3 + last2 * last;
7
```

Also in each statement the time spent by the last statement, whether it produced a value or not, is available in the variable `time` (7.6.3). This is an integer that holds the number of milliseconds.

6.2 Special Rules for Input Lines

The input for some GAP objects may not fit on one line, in particular big integers, long strings or long identifiers. In these cases you can still type or paste them in long single lines. For nicer display you can also specify the input on several lines. This is achieved by ending a line by a backslash or by a backslash and a carriage return character, then continue the input on the beginning of the next line. When reading this GAP will ignore such continuation backslashes, carriage return characters and newline characters. GAP also prints long strings and integers this way.

Example

```
gap> n := 1234\
> 567890;
1234567890
gap> "This is a very long string that does not fit on a line \
> and is therefore continued on the next line.";
"This is a very long string that does not fit on a line and is therefo\
re continued on the next line."
gap> bla\
> bla := 5;; blabla;
5
```

There is a special rule about GAP prompts in input lines: In line editing mode (usual user input and GAP started without `-n`) in lines starting with whitespace following `gap> `, `> ` or `brk> ` this beginning part is removed. This rule is very convenient because it allows to cut and paste input from other GAP sessions or manual examples easily into your current session.

6.3 View and Print

GAP has three different operations to display or print objects: `Display` (6.3.6), `ViewObj` (6.3.5) and `PrintObj` (6.3.5), and these three have different purposes as follows. The first, `Display` (6.3.6), should print the object to the standard output in a human-readable relatively complete and verbose form. The second, `ViewObj` (6.3.5), should print the object to the standard output in a short and concise form, it is used in the main read-eval-print loop to display the resulting object of a computation. The third, `PrintObj` (6.3.5), should print the object to the standard output in a complete form which is GAP-readable if at all possible, such that reading the output into GAP produces an object which is equal to the original one.

All three operations have corresponding operations which do not print anything to standard output but return the output as a string. These are `DisplayString` (27.6.1), `ViewString` (27.6.3) and `PrintString` (27.6.5) (corresponding to `PrintObj` (6.3.5)). Additionally, there is `String` (27.6.6) which is very similar to `PrintString` (27.6.5) but does not insert control characters for line breaks.

For implementation convenience it is allowed that some of these operations have methods which delegate to some other of these operations. However, the rules for this are that a method may only delegate to another operation which appears further down in the following table:

Display (6.3.6)
ViewObj (6.3.5)
PrintObj (6.3.5)
DisplayString (27.6.1)
ViewString (27.6.3)
PrintString (27.6.5)
String (27.6.6)

This is to avoid circular delegations.

Note in particular that none of the methods of the string producing operations may delegate to the corresponding printing operations. Note also that the above mentioned purposes of the different operations suggest that delegations between different operations will be sub-optimal in most scenarios.

6.3.1 Default delegations in the library

The library contains the following low ranked default methods:

- A method for DisplayString (27.6.1) which returns the constant value of the global variable DEFAULTDISPLAYSTRING (27.6.2).
- A method for ViewString (27.6.3) which returns the constant value of the global variable DEFAULTVIEWSTRING (27.6.4).
- A method for Display (6.3.6) which first calls DisplayString (27.6.1) and prints the result, if it is a different object than DEFAULTDISPLAYSTRING (27.6.2). Otherwise the method delegates to PrintObj (6.3.5).
- A method for ViewObj (6.3.5) which first calls ViewString (27.6.3) and prints the result, if it is a different object than DEFAULTVIEWSTRING (27.6.4). Otherwise the method delegates to PrintObj (6.3.5).
- A method for PrintObj (6.3.5) which prints the result of PrintString (27.6.5).
- A method for PrintString (27.6.5) which returns the result of String (27.6.6)

6.3.2 Recommendations for the implementation

This subsection describes what methods for printing and viewing one should implement for new GAP objects.

One should at the very least install a String (27.6.6) method to allow printing. Using the standard delegations this enables a limited form of viewing, displaying and printing.

If, for larger objects, nicer line breaks are needed, one should install a separate PrintString (27.6.5) method which puts in positions for good line breaks using the control characters \< (ASCII 1) and \> (ASCII 2).

If, for even larger objects, output performance and memory usage matters, one should install a separate PrintObj (6.3.5) method.

One should usually install a ViewString (27.6.3) method, unless the above String (27.6.6) method is good enough for ViewObj (6.3.5) purposes. Performance and memory should never matter here, so it is usually unnecessary to install a separate ViewObj (6.3.5) method.

If the type of object calls for it one should install a DisplayString (27.6.1) method. This is the case if a human readable verbose form is required.

If the performance and memory usage for Display (6.3.6) matters, one should install a separate Display (6.3.6) method.

Note that if only a String (27.6.6) method is installed, then ViewObj (6.3.5) works and ViewString (27.6.3) returns DEFAULTVIEWSTRING (27.6.4). Likewise, Display (6.3.6) works and DisplayString (27.6.1) returns DEFAULTDISPLAYSTRING (27.6.2). If you want to avoid this then install methods for these operations as well.

6.3.3 View

▷ View(*obj1, obj2...*) (function)

View shows the objects *obj1*, *obj2*... etc. *in a short form* on the standard output by calling the ViewObj (6.3.5) operation on each of them. View is called in the read-eval-print loop, thus the output looks exactly like the representation of the objects shown by the main loop. Note that no space or newline is printed between the objects.

6.3.4 Print

▷ Print(*obj1, obj2, ...*) (function)

Also Print shows the objects *obj1*, *obj2*... etc. on the standard output. The difference compared to View (6.3.3) is in general that the shown form is not required to be short, and that in many cases the form shown by Print is GAP readable.

Example

```
gap> z:= Z(2);
Z(2)^0
gap> v:= [ z, z, z, z, z, z, z ];
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> ConvertToVectorRep(v);; v;
<a GF2 vector of length 7>
gap> Print( v, "\n" );
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]
```

Another difference is that Print shows strings without the enclosing quotes, so Print can be used to produce formatted text on the standard output (see also chapter 27). Some characters preceded by a backslash, such as \n, are processed specially (see chapter 27.2). PrintTo (9.7.3) can be used to print to a file.

Example

```
gap> for i in [1..5] do
>      Print( i, " ", i^2, " ", i^3, "\n" );
>    od;
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
gap> g:= SmallGroup(12,5);
```

```
<pc group of size 12 with 3 generators>
gap> Print( g, "\n" );
Group( [ f1, f2, f3 ] )
gap> View( g );  Print( "\n" );
<pc group of size 12 with 3 generators>
```

6.3.5 ViewObj

▷ ViewObj(*obj*) (operation)
▷ PrintObj(*obj*) (operation)

The functions View (6.3.3) and Print (6.3.4) actually call the operations ViewObj and PrintObj, respectively, for each argument. By installing special methods for these operations, it is possible to achieve special printing behavior for certain objects (see chapter 78). The only exceptions are strings (see Chapter 27), for which the default PrintObj and ViewObj methods as well as the function View (6.3.3) print also the enclosing doublequotes, whereas Print (6.3.4) strips the doublequotes.

The default method for ViewObj is to call PrintObj. So it is sufficient to have a PrintObj method for an object in order to View (6.3.3) it. If one wants to supply a "short form" for View (6.3.3), one can install additionally a method for ViewObj.

6.3.6 Display

▷ Display(*obj*) (operation)

Displays the object *obj* in a nice, formatted way which is easy to read (but might be difficult for machines to understand). The actual format used for this depends on the type of *obj*. Each method should print a newline character as last character.

Example

```
gap> Display( [ [ 1, 2, 3 ], [ 4, 5, 6 ] ] * Z(5) );
 2 4 1
 3 . 2
```

One can assign a string to an object that Print (6.3.4) will use instead of the default used by Print (6.3.4), via SetName (12.8.1). Also, Name (12.8.2) returns the string previously assigned to the object for printing, via SetName (12.8.1). The following is an example in the context of domains.

Example

```
gap> g:= Group( (1,2,3,4) );
Group([ (1,2,3,4) ])
gap> SetName( g, "C4" ); g;
C4
gap> Name( g );
"C4"
```

When setting up examples, in particular if for beginning users, it sometimes can be convenient to hide the structure behind a printing name. For many objects, such as groups, this can be done using SetName (12.8.1). If the objects however is represented internally, for example permutations representing group elements, this function is not applicable. Instead the function SetNameObject (6.3.7) can be used to interface with the display routines on a lower level.

6.3.7 SetNameObject

▷ `SetNameObject(o, s)` (function)

`SetNameObject` sets the string `s` as display name for object `o` in an interactive session. When applying `View` (6.3.3) to object `o`, for example in the system's main loop, GAP will print the string `s`. Calling `SetNameObject` for the same object `o` with `s` set to `fail` deletes the special viewing setup. since use of this features potentially slows down the whole print process, this function should be used sparingly.

Example

```
gap> SetNameObject(3,"three");
gap> Filtered([1..10],IsPrimeInt);
[ 2, three, 5, 7 ]
gap> SetNameObject(3,fail);
gap> Filtered([1..10],IsPrimeInt);
[ 2, 3, 5, 7 ]
```

6.4 Break Loops

When an error has occurred or when you interrupt GAP (usually by hitting CTRL-C) GAP enters a break loop, that is in most respects like the main read eval print loop (see 6.1). That is, you can enter statements, GAP reads them, evaluates them, and shows the result if any. However those evaluations happen within the context in which the error occurred. So you can look at the arguments and local variables of the functions that were active when the error happened and even change them. The prompt is changed from `gap>` to `brk>` to indicate that you are in a break loop.

Example

```
gap> 1/0;
Rational operations: <divisor> must not be zero
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <divisor> via 'return <divisor>;' to continue
```

If errors occur within a break loop GAP enters another break loop at a *deeper level*. This is indicated by a number appended to `brk`:

Example

```
brk> 1/0;
Rational operations: <divisor> must not be zero
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <divisor> via 'return <divisor>;' to continue
brk_02>
```

There are two ways to leave a break loop, see 6.4.1 and 6.4.2.

6.4.1 quit from a break loop

The first way to leave a break loop is to *quit* the break loop. To do this you enter `quit;` or type the *eof* (*e*nd *o*f *f*ile) character, which is usually CTRL-D except when using the `-e` option (see Section 3.1). Note that GAP code between `quit;` and the end of the input line is ignored.

Example

```
brk_02> quit;
brk>
```

In this case control returns to the break loop one level above or to the main loop, respectively. So iterated break loops must be left iteratively. Note also that if you type `quit;` from a `gap>` prompt, GAP will exit (see 6.7).

Note: If you leave a break loop with `quit` without completing a command it is possible (though not very likely) that data structures will be corrupted or incomplete data have been stored in objects. Therefore no guarantee can be given that calculations afterwards will return correct results! If you have been using options `quit`ting a break loop generally leaves the options stack with options you no longer want. The function `ResetOptionsStack` (8.1.3) removes all options on the options stack, and this is the sole intended purpose of this function.

6.4.2 return from a break loop

The other way to leave a break loop is to *return* from a break loop. To do this you type `return;` or `return` *`obj`*`;`. If the break loop was entered because you interrupted GAP, then you can continue by typing `return;`. If the break loop was entered due to an error, you may have to modify the value of a variable before typing `return;` (see the example for `IsDenseList` (21.1.2)) or you may have to return an object *`obj`* (by typing: `return` *`obj`*`;`) to continue the computation; in any case, the message printed on entering the break loop will tell you which of these alternatives is possible. For example, if the break loop was entered because a variable had no assigned value, the value to be returned is often a value that this variable should have to continue the computation.

Example

```
brk> return 9;  # we had tried to enter the divisor 9 but typed 0 ...
1/9
gap>
```

6.4.3 OnBreak

▷ `OnBreak` (global variable)

By default, when a break loop is entered, GAP prints a trace of the innermost 5 commands currently being executed. This behaviour can be configured by changing the value of the global variable `OnBreak`. When a break loop is entered, the value of `OnBreak` is checked. If it is a function, then it is called with no arguments. By default, the value of `OnBreak` is `Where` (6.4.5).

Example

```
gap> OnBreak := function() Print("Hello\n"); end;
function(  ) ... end
```

Example

```
gap> Error("!\n");
Error, !
Hello
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

In cases where a break loop is entered during a function that was called with options (see Chapter 8), a `quit;` will also cause the options stack to be reset and an `Info`-ed warning stating this is emitted at `InfoWarning` (7.4.7) level 1 (see Chapter 7.4).

Note that for break loops entered by a call to `Error` (6.6.1), the lines after "`Entering break read-eval-print loop ...`" and before the `brk>` prompt can also be customised, namely by redefining `OnBreakMessage` (6.4.4).

Also, note that one can achieve the effect of changing `OnBreak` *locally*. As mentioned above, the default value of `OnBreak` is `Where` (6.4.5). Thus, a call to `Error` (6.6.1) generally gives a trace back up to five levels of calling functions. Conceivably, we might like to have a function like `Error` (6.6.1) that does not trace back without globally changing `OnBreak`. Such a function we might call `ErrorNoTraceBack` and here is how we might define it. (Note `ErrorNoTraceBack` is *not* a GAP function.)

Example

```
gap> ErrorNoTraceBack := function(arg) # arg is special variable that GAP
>                                      # knows to treat as list of arg's
>      local SavedOnBreak, ENTBOnBreak;
>      SavedOnBreak := OnBreak;        # save current value of OnBreak
>
>      ENTBOnBreak := function()       # our 'local' OnBreak
>      local s;
>        for s in arg do
>          Print(s);
>        od;
>        OnBreak := SavedOnBreak;      # restore OnBreak afterwards
>      end;
>
>      OnBreak := ENTBOnBreak;
>      Error();
>    end;
function( arg ) ... end
```

Here is a somewhat trivial demonstration of the use of `ErrorNoTraceBack`.

Example

```
gap> ErrorNoTraceBack("Gidday!", " How's", " it", " going?\n");
Error, Gidday! How's it going?
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Now we call `Error` (6.6.1) with the same arguments to show the difference.

Example

```
gap> Error("Gidday!", " How's", " it", " going?\n");
Error, Gidday! How's it going?
Hello
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Observe that the value of OnBreak before the ErrorNoTraceBack call was restored. However, we had changed OnBreak from its default value; to restore OnBreak to its default value, we should do the following.

Example

```
gap> OnBreak := Where;;
```

6.4.4 OnBreakMessage

▷ OnBreakMessage (global variable)

When a break loop is entered by a call to Error (6.6.1) the message after the "Entering break read-eval-print loop ..." line is produced by the function OnBreakMessage, which just like OnBreak (6.4.3) is a user-configurable global variable that is a *function* with *no arguments*.

Example

```
gap> OnBreakMessage(); # By default, OnBreakMessage prints the following
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
```

Perhaps you are familiar with what's possible in a break loop, and so don't need to be reminded. In this case, you might wish to do the following (the first line just makes it easy to restore the default value later).

Example

```
gap> NormalOnBreakMessage := OnBreakMessage;; # save the default value
gap> OnBreakMessage := function() end;        # do-nothing function
function(  ) ... end
gap> OnBreakMessage();
gap> OnBreakMessage := NormalOnBreakMessage;; # reset
```

With OnBreak (6.4.3) still set away from its default value, calling Error (6.6.1) as we did above, now produces:

Example

```
gap> Error("!\n");
Error, !
Hello
Entering break read-eval-print loop ...
brk> quit; # to get back to outer loop
```

However, suppose you are writing a function which detects an error condition and OnBreakMessage needs to be changed only *locally*, i.e., the instructions on how to recover from the break loop need to be specific to that function. The same idea used to define ErrorNoTraceBack (see OnBreak (6.4.3)) can be adapted to achieve this. The function CosetTableFromGensAndRels (47.6.5) is an example in the GAP code where the idea is actually used.

6.4.5 Where

▷ `Where(`*nr*`)` (function)

shows the last *nr* commands on the execution stack during whose execution the error occurred. If not given, *nr* defaults to 5. (Assume, for the following example, that after the last example `OnBreak` (6.4.3) has been set back to its default value.)

Example

```
gap> StabChain(SymmetricGroup(100)); # After this we typed ^C
user interrupt at
bpt := S.orbit[1];
 called from
SiftedPermutation( S, (g * rep) ^ -1 ) called from
StabChainStrong( S.stabilizer, [ sch ], options ); called from
StabChainStrong( S.stabilizer, [ sch ], options ); called from
StabChainStrong( S, GeneratorsOfGroup( G ), options ); called from
StabChainOp( G, rec(
     ) ) called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> Where(2);
 called from
SiftedPermutation( S, (g * rep) ^ -1 ) called from
StabChainStrong( S.stabilizer, [ sch ], options ); called from
...
```

Note that the variables displayed even in the first line of the `Where` list (after the `called from` line) may be already one environment level higher and `DownEnv` (6.5.1) may be necessary to access them.

At the moment this backtrace does not work from within compiled code (this includes the method selection which by default is compiled into the kernel). If this creates problems for debugging, call GAP with the `-M` option (see 3.1) to avoid loading compiled code.

(Function calls to `Info` (7.4.5) and methods installed for binary operations are handled in a special way. In rare circumstances it is possible therefore that they do not show up in a `Where` log but the log refers to the *last* proper function call that happened before.)

The command line option `-T` to GAP disables the break loop. This is mainly intended for testing purposes and for special applications. If this option is given then errors simply cause GAP to return to the main loop.

6.5 Variable Access in a Break Loop

In a break loop access to variables of the current break level and higher levels is possible, but if the same variable name is used for different objects or if a function calls itself recursively, of course only the variable at the lowest level can be accessed.

6.5.1 DownEnv and UpEnv

▷ DownEnv(*nr*) (function)
▷ UpEnv(*nr*) (function)

DownEnv moves down *nr* steps in the environment and allows one to inspect variables on this level; if *nr* is negative it steps up in the environment again; *nr* defaults to 1 if not given. UpEnv acts similarly to DownEnv but in the reverse direction (the mnemonic rule to remember the difference between DownEnv and UpEnv is the order in which commands on the execution stack are displayed by Where (6.4.5)).

Example

```
gap> OnBreak := function() Where(0); end;; # eliminate back-tracing on
gap>                                       # entry to break loop
gap> test:= function( n )
>    if n > 3 then Error( "!\n" ); fi; test( n+1 ); end;;
gap> test( 1 );
Error, !
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> n;
4
brk> DownEnv();
brk> n;
3
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( 2 );
brk> n;
1
brk> Where();
 called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( -2 );
brk> n;
3
brk> quit;
gap> OnBreak := Where;; # restore OnBreak to its default value
```

Note that the change of the environment caused by DownEnv only affects variable access in the break loop. If you use return to continue a calculation GAP automatically jumps to the right environment level again.

Note also that search for variables looks first in the chain of outer functions which enclosed the definition of a currently executing function, before it looks at the chain of calling functions which led to the current invocation of the function.

Example

```
gap> foo := function()
> local x; x := 1;
> return function() local y; y := x*x; Error("!!\n"); end;
> end;
function(  ) ... end
gap> bar := foo();
function(  ) ... end
gap> fun := function() local x; x := 3; bar(); end;
function(  ) ... end
gap> fun();
Error, !!
 called from
bar(  ); called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> x;
1
brk> DownEnv(1);
brk> x;
3
```

Here the `x` of `foo` which contained the definition of `bar` is found before that of `fun` which caused its execution. Using `DownEnv` we can access the `x` from `fun`.

6.6 Error and ErrorCount

6.6.1 Error

▷ `Error(`*`messages, ...`*`)` (function)

`Error` signals an error from within a function. First the messages *`messages`* are printed, this is done exactly as if `Print` (6.3.4) (see 6.3) were called with these arguments. Then a break loop (see 6.4) is entered, unless the standard error output is not connected to a terminal. You can leave this break loop with `return;` to continue execution with the statement following the call to `Error`.

6.6.2 ErrorCount

▷ `ErrorCount()` (function)

`ErrorCount` returns a count of the number of errors (including user interruptions) which have occurred in the GAP session so far. This count is reduced modulo 2^{28} on 32 bit systems, 2^{60} on 64 bit systems. The count is incremented by each error, even if GAP was started with the `-T` option to disable the break loop.

6.7 Leaving GAP

The normal way to terminate a GAP session is to enter either `quit;` (note the semicolon) or an end-of-file character (usually CTRL-D) at the `gap>` prompt in the main read eval print loop.

6.7.1 QUIT

▷ `QUIT` (global variable)

An emergency way to leave GAP is to enter `QUIT` at any `gap>` or `brk>` or `brk_nn>` prompt.

6.7.2 InstallAtExit

▷ `InstallAtExit(func)` (function)
▷ `QUITTING` (global variable)

Before actually terminating, GAP will call (with no arguments) all of the functions that have been installed using `InstallAtExit`. These typically perform tasks such as cleaning up temporary files created during the session, and closing open files. If an error occurs during the execution of one of these functions, that function is simply abandoned, no break loop is entered.

Example

```
gap> InstallAtExit(function() Print("bye\n"); end);
gap> quit;
bye
```

During execution of these functions, the global variable `QUITTING` will be set to `true` if GAP is exiting because the user typed `QUIT` and `false` otherwise. Since `QUIT` is considered as an emergency measure, different action may be appropriate.

6.7.3 SaveOnExitFile

▷ `SaveOnExitFile` (global variable)

If, when GAP is exiting due to a `quit` or end-of-file (i.e. not due to a `QUIT`) the variable `SaveOnExitFile` is bound to a string value, then the system will try to save the workspace to that file.

6.8 Line Editing

In most installations GAP will be compiled to use the Gnu readline library (see the line `Libs used:` on GAP startup). In that case skip to the next section 6.9. (The line editing commands described in the rest of this section were available in previous versions of GAP, they will work almost the same in the standard configuration of the Gnu readline library.)

GAP allows one you to edit the current input line with a number of editing commands. Those commands are accessible either as *control keys* or as *escape keys*. You enter a control key by pressing the CTRL key, and, while still holding the CTRL key down, hitting another key `key`. You enter an

escape key by hitting ESC and then hitting another key `key`. Below we denote control keys by CTRL-`key` and escape keys by ESC-`key`. The case of `key` does not matter, i.e., CTRL-A and CTRL-a are equivalent.

Normally, line editing will be enabled if the input is connected to a terminal. Line editing can be enabled or disabled using the command line options `-f` and `-n` respectively (see 3.1), however this is a machine dependent feature of GAP.

Typing CTRL-KEY or ESC-KEY for characters not mentioned below always inserts CTRL-`key` resp. ESC-`key` at the current cursor position.

The first few commands allow you to move the cursor on the current line.

CTRL-A
move the cursor to the beginning of the line.

ESC-B
move the cursor to the beginning of the previous word.

CTRL-B
move the cursor backward one character.

CTRL-F
move the cursor forward one character.

ESC-F
move the cursor to the end of the next word.

CTRL-E
move the cursor to the end of the line.

The next commands delete or kill text. The last killed text can be reinserted, possibly at a different position, with the "yank" command CTRL-Y.

CTRL-H or *del*
delete the character left of the cursor.

CTRL-D
delete the character under the cursor.

CTRL-K
kill up to the end of the line.

ESC-D
kill forward to the end of the next word.

ESC-DEL
kill backward to the beginning of the last word.

CTRL-X
kill entire input line, and discard all pending input.

CTRL-Y
insert (yank) a just killed text.

The next commands allow you to change the input.

CTRL-T
exchange (twiddle) current and previous character.

ESC-U
uppercase next word.

ESC-L
lowercase next word.

ESC-C
capitalize next word.

The TAB character, which is in fact the control key CTRL-I, looks at the characters before the cursor, interprets them as the beginning of an identifier and tries to complete this identifier. If there is more than one possible completion, it completes to the longest common prefix of all those completions. If the characters to the left of the cursor are already the longest common prefix of all completions hitting TAB a second time will display all possible completions.

TAB complete the identifier before the cursor.

The next commands allow you to fetch previous lines, e.g., to correct typos, etc.

CTRL-L
insert last input line before current character.

CTRL-P
redisplay the last input line, another CTRL-P will redisplay the line before that, etc. If the cursor is not in the first column only the lines starting with the string to the left of the cursor are taken.

CTRL-N
Like CTRL-P but goes the other way round through the history.

ESC-<
goes to the beginning of the history.

ESC->
goes to the end of the history.

CTRL-O
accepts this line and perform a CTRL-N.

Finally there are a few miscellaneous commands.

CTRL-V
enter next character literally, i.e., enter it even if it is one of the control keys.

CTRL-U
execute the next line editing command 4 times.

ESC-`num`
execute the next line editing command `num` times.

ESC-CTRL-L
redisplay input line.

The four arrow keys (cursor keys) can be used instead of CTRL-B, CTRL-F, CTRL-P, and CTRL-N, respectively.

6.9 Editing using the `readline` library

The descriptions in this section are valid only if your GAP installation uses the `readline` library for command line editing. You can check by `IsBound(GAPInfo.UseReadline);` if this is the case.

You can use all the features of `readline`, as for example explained in `http://tiswww.case.edu/php/chet/readline/rluserman.html`. Therefore the command line editing in GAP is similar to the `bash` shell and many other programs. On a Unix/Linux system you may also have a manpage, try `man readline`.

Compared to the command line editing which was used in GAP up to version 4.4 (or compared to not using the `readline` library) using `readline` has several advantages:

- Most keys still do the same as explained in 6.8 (in the default configuration).
- There are many additional commands, e.g. undoing (CTRL-_, keyboard macros (CTRL-X(, CTRL-X) and CTRL-XE), file name completion (hit ESC two or four times), showing matching parentheses, `vi`-style key bindings, deleting and yanking text, ...
- Lines which are longer than a physical terminal row can be edited more conveniently.
- Arbitrary unicode characters can be typed into string literals.
- The key bindings can be configured, either via your `~/.inputrc` file or by GAP commands, see 6.9.1.
- The command line history can be saved to and read from a file, see 6.9.2.
- Adventurous users can even implement completely new command line editing functions on GAP level, see 6.9.4.

6.9.1 Readline customization

You can use your readline init file (by default `~/.inputrc` on Unix/Linux) to customize key bindings. If you want settings be used only within GAP you can write them between lines containing `$if GAP` and `$endif`. For a detailed documentation of the available settings and functions see here.

From readline init file

```
$if GAP
  set blink-matching-paren on
  "\C-n": dump-functions
  "\ep": kill-region
$endif
```

Alternatively, from within GAP the command `ReadlineInitLine(line);` can be used, where *line* is a string containing a line as in the init file.

Note that after pressing CTRL-V the next special character is input verbatim. This is very useful to bind keys or key sequences. For example, binding the function key F3 to the command `kill-whole-line` by using the sequence CTRL-V F3 looks on many terminals like this: `ReadlineInitLine("\"^[OR\":kill-whole-line");`. (You can get the line back later with CTRL-Y.)

The CTRL-G key can be used to type any unicode character by its code point. The number of the character can either be given as a count, or if the count is one the input characters before the cursor are taken (as decimal number or as hex number which starts with `0x`. For example, the double stroke character $\mathbb{Z}$ can be input by any of the three key sequences ESC 8484 CTRL-G, 8484 CTRL-G or 0X2124 CTRL-G.

Some terminals bind the CTRL-S and CTRL-Q keys to stop and restart terminal output. Furthermore, sometimes CTRL-\ quits a program. To disable this behaviour (and maybe use these keys for command line editing) you can use `Exec("stty stop undef; stty start undef; stty quit undef");` in your GAP session or your `gaprc` file (see 3.2).

6.9.2 The command line history

GAP can save your input lines for later reuse. The keys CTRL-P (or UP), CTRL-N (or DOWN), ESC< and ESC> work as documented in 6.8, that is they scroll backward and forward in the history or go to its beginning or end. Also, CTRL-O works as documented, it is useful for repeating a sequence of previous lines. (But CTRL-L clears the screen as in other programs.)

The command line history can be used across several instances of GAP via the following two commands.

6.9.3 SaveCommandLineHistory

▷ `SaveCommandLineHistory(`*`[fname, ][app]`*`)` (function)

Returns: `fail` or number of saved lines

▷ `ReadCommandLineHistory(`*`[fname]`*`)` (function)

Returns: `fail` or number of added lines

The first command saves the lines in the command line history to the file given by the string *fname*. The default for *fname* is `history` in the user's GAP root path `GAPInfo.UserGapRoot` or `"~/.gap_hist"` if this directory does not exist. If the optional argument *app* is `true` then the lines are appended to that file otherwise the file is overwritten.

The second command is the converse, it reads the lines from file *fname* and *prepends* them to the current command line history.

By default an arbitrary number of input lines is stored in the command line history. For very long GAP sessions or if `SaveCommandLineHistory` and `ReadCommandLineHistory` are used repeatedly it can be sensible to restrict the number of saved lines via `SetUserPreference("HistoryMaxLines", num);` to a non negative number `num` (the default is `infinity`). An automatic storing and restoring of the command line history can be configured via `SetUserPreference("SaveAndRestoreHistory", true);`.

Note that these functions are only available if your GAP is configured to use the `readline` library.

6.9.4 Writing your own command line editing functions

It is possible to write new command line editing functions in GAP as follows.

The functions have one argument `l` which is a list with five entries of the form `[count, key, line, cursorpos, markpos]` where `count` and `key` are the last pressed key and its count (these are not so useful here because users probably do not want to overwrite the binding of a single key), then `line` is a string containing the line typed so far, `cursorpos` is the current position of the cursor (point), and `markpos` the current position of the mark.

The result of such a function must be a list which can have various forms:

`[str]`
: with a string `str`. In this case the text `str` is inserted at the cursor position.

`[kill, begin, end]`
: where `kill` is `true` or `false` and `begin` and `end` are positions on the input line. This removes the text from the lower position to before the higher position. If `kill` is `true` the text is killed, i.e. put in the kill ring for later yanking.

`[begin, end, str]`
: where `begin` and `end` are positions on the input line and `str` is a string. Then the text from position `begin` to before `end` is substituted by `str`.

`[1, lstr]`
: where `lstr` is a list of strings. Then these strings are displayed like a list of possible completions. The input line is not changed.

`[2, chars]`
: where `chars` is a string. The characters from `chars` are used as the next characters from the input. (At most 512 characters are possible.)

`[100]`
: This rings the bell as configured in the terminal.

In the first three cases the result list can contain a position as a further entry, this becomes the new cursor position. Or it can contain two positions as further entries, these become the new cursor position and the new position of the mark.

Such a function can be installed as a macro for `readline` via `InstallReadlineMacro(name, fun);` where `name` is a string used as name of the macro and `fun` is a function as above. This macro can be called by a key sequence which is returned by `InvocationReadlineMacro(name);`.

As an example we define a function which puts double quotes around the word under or before the cursor position. The space character, the characters in `"(,)"`, and the beginning and end of the line are considered as word boundaries. The function is then installed as a macro and bound to the key sequence ESC Q.

Example

```
gap> EditAddQuotes := function(l)
>   local str, pos, i, j, new;
>   str := l[3];
>   pos := l[4];
>   i := pos;
>   while i > 1 and (not str[i-1] in ",( ") do
```

```
>     i := i-1;
>   od;
>   j := pos;
>   while IsBound(str[j]) and not str[j] in ",) " do
>     j := j+1;
>   od;
>   new := "\"";
>   Append(new, str{[i..j-1]});
>   Append(new, "\"");
>   return [i, j, new];
> end;;
gap> InstallReadlineMacro("addquotes", EditAddQuotes);
gap> invl := InvocationReadlineMacro("addquotes");;
gap> ReadlineInitLine(Concatenation("\"\\eQ\":\"",invl,"\""));;
```

6.10 Editing Files

In most cases, it is preferable to create longer input (in particular GAP programs) separately in an editor, and to read in the result via `Read` (9.7.1). Note that `Read` (9.7.1) by default reads from the directory in which GAP was started (respectively under Windows the directory containing the GAP binary), so you might have to give an absolute path to the file.

If you cannot create several windows, the `Edit` (6.10.1) command may be used to leave GAP, start an editor, and read in the edited file automatically.

6.10.1 Edit

▷ `Edit(`*filename*`)` (function)

`Edit` starts an editor with the file whose filename is given by the string *filename*, and reads the file back into GAP when you exit the editor again.

GAP will call your preferred editor if you call `SetUserPreference("Editor", `*path*`);` where *path* is the path to your editor, e.g., `/usr/bin/vim`. On Windows you can use `edit.com`.

Under Mac OS X, you should use `SetUserPreference("Editor", "open");`, this will open the file in the default editor. If you call `SetUserPreference("EditorOptions", ["-t"]);`, the file will open in `TextEdit`, and `SetUserPreference("EditorOptions", ["-a", "<appl>"]);` will open the file using the application `<appl>`.

This can for example be done in your `gap.ini` file, see Section 3.2.1.

6.11 Editor Support

In the `etc` subdirectory of the GAP installation we provide some setup files for the editors `vim` and `emacs/xemacs`.

`vim` is a powerful editor that understands the basic `vi` commands but provides much more functionality. You can find more information about it (and download it) from `http://www.vim.org`.

To get support for GAP syntax in vim, create in your home directory a directory `.vim` with subdirectories `.vim/syntax` and `.vim/indent` (If you are not using Unix, refer to the `vim` documentation

on where to place syntax files). Then copy the file `etc/gap.vim` to `.vim/syntax/gap.vim` and the file `etc/gap_indent.vim` to `.vim/indent/gap.vim`.

Then edit the `.vimrc` file in your home directory. Add lines as in the following example:

Example

```
if has("syntax")
  syntax on             " Default to no syntax highlightning
endif

" For GAP files
augroup gap
  " Remove all gap autocommands
  au!
autocmd BufRead,BufNewFile *.g,*.gi,*.gd set filetype=gap comments=s:##\ \ ,m:##\ \ ,e:##\ \ b:#

" I'm using the external program 'par' for formating comment lines starting
" with '##  '. Include these lines only when you have par installed.
  autocmd BufRead,BufNewFile *.g,*.gi,*.gd set formatprg="par w76p4s0j"
  autocmd BufWritePost,FileWritePost *.g,*.gi,*.gd set formatprg="par w76p0s0j"
augroup END
```

See the headers of the two mentioned files for additional comments and adjust details according to your personal taste. Send comments and suggestions to `support@gap-system.org`. Setup files for `emacs`/`xemacs` are contained in the `etc/emacs` subdirectory.

6.12 Changing the Screen Size

6.12.1 SizeScreen

▷ `SizeScreen(`*`[sz]`*`)` (function)

Called with no arguments, `SizeScreen` returns the size of the screen as a list with two entries. The first is the length of each line, the second is the number of lines.

Called with one argument that is a list *sz*, `SizeScreen` sets the size of the screen; The first entry of *sz*, if bound, is the length of each line, and the second entry of *sz*, if bound, is the number of lines. The values for unbound entries of *sz* are left unaffected. The function returns the new values.

Note that those parameters can also be set with the command line options `-x` for the line length and `-y` for the number of lines (see Section 3.1).

To check/change whether line breaking occurs for files and streams see `PrintFormattingStatus` (10.4.8) and `SetPrintFormattingStatus` (10.4.8).

The line length must be between 20 and 4096 characters (inclusive) and the number of lines must be at least 10. Values outside this range will be adjusted to the nearest endpoint of the range.

6.13 Teaching Mode

When using GAP in the context of (undergraduate) teaching it is often desirable to simplify some of the system output and functionality defaults (potentially at the cost of making the printing of objects more expensive). This can be achieved by turning on a teaching mode:

6.13.1 TeachingMode

▷ `TeachingMode(`*`[switch]`*`)` (function)

When called with a boolean argument *switch*, this function will turn teaching mode respectively on or off.

Example

```
gap> a:=Z(11)^3;
Z(11)^3
gap> TeachingMode(true);
#I  Teaching mode is turned ON
gap> a;
ZmodnZObj(8,11)
gap> TeachingMode(false);
#I  Teaching mode is turned OFF
gap> a;
Z(11)^3
```

At the moment, teaching mode changes the following things

Prime Field Elements
Elements of fields of prime order are printed as `ZmodnZObj` (14.5.3) instead as power of a primitive root.

Quadratic Irrationalities
Elements of a quadratic extension of the rationals are printed using the square root `ER` (18.4.2) instead of using roots of unity.

Creation of some small groups
The group creator functions `CyclicGroup` (50.1.2), `AbelianGroup` (50.1.3), `ElementaryAbelianGroup` (50.1.4), and `DihedralGroup` (50.1.6) create by default (if no other representation is specified) not a pc group, but a finitely presented group, which makes the generators easier to interpret.

Chapter 7

Debugging and Profiling Facilities

This chapter describes some functions that are useful mainly for debugging and profiling purposes.

Probably the most important debugging tool in GAP is the break loop (see Section 6.4) which can be entered by putting an `Error` (6.6.1) statement into your code or by hitting Control-C. In the break loop one can inspect variables, stack traces and issue commands as usual in an interactive GAP session. See also the `DownEnv` (6.5.1), `UpEnv` (6.5.1) and `Where` (6.4.5) functions.

Sections 7.2 and 7.3 show how to get information about the methods chosen by the method selection mechanism (see chapter 78).

The final sections describe functions for collecting statistics about computations (see `Runtime` (7.6.2), 7.7).

7.1 Recovery from NoMethodFound-Errors

When the method selection fails because there is no applicable method, an error as in the following example occurs and a break loop is entered:

Example

```
gap> IsNormal(2,2);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for `IsNormal' on 2 arguments called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk>
```

This only says, that the method selection tried to find a method for `IsNormal` on two arguments and failed. In this situation it is crucial to find out, why this happened. Therefore there are a few functions which can display further information. Note that you can leave the break loop by the `quit` command (see 6.4.1) and that the information about the incident is no longer accessible afterwards.

7.1.1 ShowArguments

▷ `ShowArguments()` (function)

This function is only available within a break loop caused by a "No Method Found"-error. It prints as a list the arguments of the operation call for which no method was found.

7.1.2 ShowArgument

▷ `ShowArgument(`*nr*`)` (function)

This function is only available within a break loop caused by a "No Method Found"-error. It prints the *nr*-th arguments of the operation call for which no method was found. `ShowArgument` needs exactly one argument which is an integer between 0 and the number of arguments the operation was called with.

7.1.3 ShowDetails

▷ `ShowDetails()` (function)

This function is only available within a break loop caused by a "No Method Found"-error. It prints the details of this error: The operation, the number of arguments, a flag which indicates whether the operation is being traced, a flag which indicates whether the operation is a constructor method, and the number of methods that refused to apply by calling `TryNextMethod` (78.4.1). The last number is called `Choice` and is printed as an ordinal. So if exactly k methods were found but called `TryNextMethod` (78.4.1) and there were no more methods it says `Choice:` k`th`.

7.1.4 ShowMethods

▷ `ShowMethods(`*[verbosity]*`)` (function)

This function is only available within a break loop caused by a "No Method Found"-error. It prints an overview about the installed methods for those arguments the operation was called with (using 7.2. The verbosity can be controlled by the optional integer parameter *verbosity*. The default is 2, which lists all applicable methods. With verbosity 1 `ShowMethods` only shows the number of installed methods and the methods matching, which can only be those that were already called but refused to work by calling `TryNextMethod` (78.4.1). With verbosity 3 not only all installed methods but also the reasons why they do not match are displayed.

7.1.5 ShowOtherMethods

▷ `ShowOtherMethods(`*[verbosity]*`)` (function)

This function is only available within a break loop caused by a "No Method Found"-error. It prints an overview about the installed methods for a different number of arguments than the number of arguments the operation was called with (using 7.2. The verbosity can be controlled by the optional integer parameter *verbosity*. The default is 1 which lists only the number of applicable methods. With verbosity 2 `ShowOtherMethods` lists all installed methods and with verbosity 3 also the reasons, why they are not applicable. Calling `ShowOtherMethods` with verbosity 3 in this function will normally not make any sense, because the different numbers of arguments are simulated by supplying the corresponding number of ones, for which normally no reasonable methods will be installed.

7.2 Inspecting Applicable Methods

7.2.1 ApplicableMethod

▷ `ApplicableMethod(`*opr, args[, printlevel[, nr]]*`)` (function)
▷ `ApplicableMethodTypes(`*opr, args[, printlevel[, nr]]*`)` (function)

Called with two arguments, `ApplicableMethod` returns the method of highest rank that is applicable for the operation *opr* with the arguments in the list *args*. The default *printlevel* is 0. If no method is applicable then `fail` is returned.

If a positive integer is given as the fourth argument *nr* then `ApplicableMethod` returns the *nr*-th applicable method for the operation *opr* with the arguments in the list *args*, where the methods are ordered according to descending rank. If less than *nr* methods are applicable then `fail` is returned.

If the fourth argument *nr* is the string `"all"` then `ApplicableMethod` returns a list of all applicable methods for *opr* with arguments *args*, ordered according to descending rank.

Depending on the integer value *printlevel*, additional information is printed. Admissible values and their meaning are as follows.

0 no information,

1 information about the applicable method,

2 also information about the not applicable methods of higher rank,

3 also for each not applicable method the first reason why it is not applicable,

4 also for each not applicable method all reasons why it is not applicable.

6 also the function body of the selected method(s)

When a method returned by `ApplicableMethod` is called then it returns either the desired result or the string `"TRY_NEXT_METHOD"`, which corresponds to a call to `TryNextMethod` (78.4.1) in the method and means that the method selection would call the next applicable method.

Note: The GAP kernel provides special treatment for the infix operations `\+`, `\-`, `\*`, `\/`, `\^`, `\mod` and `\in`. For some kernel objects (notably cyclotomic numbers, finite field elements and row vectors thereof) it calls kernel methods circumventing the method selection mechanism. Therefore for these operations `ApplicableMethod` may return a method which is not the kernel method actually used.

The function `ApplicableMethodTypes` takes the *types* or *filters* of the arguments as argument (if only filters are given of course family predicates cannot be tested).

7.3 Tracing Methods

7.3.1 TraceMethods (for operations)

▷ `TraceMethods(`*opr1, opr2, ...*`)` (function)
▷ `TraceMethods(`*oprs*`)` (function)

After the call of `TraceMethods`, whenever a method of one of the operations *opr1*, *opr2*, ... is called, the information string used in the installation of the method is printed. The second form has the same effect for each operation from the list *oprs* of operations.

7.3.2 UntraceMethods (for operations)

▷ `UntraceMethods(`*`opr1, opr2, ...`*`)` (function)
▷ `UntraceMethods(`*`oprs`*`)` (function)

turns the tracing off for all operations *opr1*, *opr2*, ... or in the second form, for all operations in the list *oprs*.

Example

```
gap> TraceMethods( [ Size ] );
gap> g:= Group( (1,2,3), (1,2) );;
gap> Size( g );
#I  Size: for a permutation group
#I  Setter(Size): system setter
#I  Size: system getter
#I  Size: system getter
6
gap> UntraceMethods( [ Size ] );
```

7.3.3 TraceImmediateMethods

▷ `TraceImmediateMethods(`*`flag`*`)` (function)

If *flag* is true, tracing for all immediate methods is turned on. If *flag* is false it is turned off. (There is no facility to trace *specific* immediate methods.)

Example

```
gap> TraceImmediateMethods( true );
gap> g:= Group( (1,2,3), (1,2) );;
#I  immediate: Size
#I  immediate: IsCyclic
#I  immediate: IsCommutative
#I  immediate: IsTrivial
gap> Size( g );
#I  immediate: IsNonTrivial
#I  immediate: Size
#I  immediate: IsFreeAbelian
#I  immediate: IsTorsionFree
#I  immediate: IsNonTrivial
#I  immediate: GeneralizedPcgs
#I  immediate: IsPerfectGroup
#I  immediate: IsEmpty
6
gap> TraceImmediateMethods( false );
gap> UntraceMethods( [ Size ] );
```

This example gives an explanation for the two calls of the "system getter" for `Size` (30.4.6). Namely, there are immediate methods that access the known size of the group. Note that the group g was known to be finitely generated already before the size was computed, the calls of the immediate method for `IsFinitelyGeneratedGroup` (39.15.17) after the call of `Size` (30.4.6) have other arguments than g.

7.4 Info Functions

The Info (7.4.5) mechanism permits operations to display intermediate results or information about the progress of the algorithms. Information is always given according to one or more *info classes*. Each of the info classes defined in the GAP library usually covers a certain range of algorithms, so for example InfoLattice covers all the cyclic extension algorithms for the computation of a subgroup lattice.

The amount of information to be displayed can be specified by the user for each info class separately by a *level*, the higher the level the more information will be displayed. Ab initio all info classes have level zero except InfoWarning (7.4.7) which initially has level 1.

7.4.1 NewInfoClass

▷ NewInfoClass(*name*) (operation)

creates a new info class with name *name*.

7.4.2 DeclareInfoClass

▷ DeclareInfoClass(*name*) (function)

creates a new info class with name *name* and binds it to the global variable *name*. The variable must previously be writable, and is made readonly by this function.

7.4.3 SetInfoLevel

▷ SetInfoLevel(*infoclass, level*) (operation)

Sets the info level for *infoclass* to *level*.

7.4.4 InfoLevel

▷ InfoLevel(*infoclass*) (operation)

returns the info level of *infoclass*.

7.4.5 Info

▷ Info(*infoclass, level, info[, moreinfo, ...]*) (function)

If the info level of *infoclass* is at least *level* the remaining arguments, *info* and possibly *moreinfo* and so on, are evaluated. (Technically, Info is a keyword and not a function.)

By default, they are viewed, preceded by the string "#I " and followed by a newline. Otherwise the third and subsequent arguments are not evaluated. (The latter can save substantial time when displaying difficult results.)

The behaviour can be customized with SetInfoHandler (7.4.6).

Example

```
gap> InfoExample:=NewInfoClass("InfoExample");;
gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
gap> SetInfoLevel(InfoExample,1);
gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
#I  one
gap> SetInfoLevel(InfoExample,2);
gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
#I  one
#I  two
gap> InfoLevel(InfoExample);
2
gap> Info(InfoExample,3,Length(Combinations([1..9999])));
```

Note that the last `Info` call is executed without problems, since the actual level 2 of `InfoExample` causes `Info` to ignore the last argument, which prevents `Length(Combinations([1..9999]))` from being evaluated; note that an evaluation would be impossible due to memory restrictions.

A set of info classes (called an *info selector*) may be passed to a single `Info` statement. As a shorthand, info classes and selectors may be combined with + rather than `Union` (30.5.3). In this case, the message is triggered if the level of *any* of the classes is high enough.

Example

```
gap> InfoExample:=NewInfoClass("InfoExample");;
gap> SetInfoLevel(InfoExample,0);
gap> Info(InfoExample + InfoWarning, 1, "hello");
#I  hello
gap> Info(InfoExample + InfoWarning, 2, "hello");
gap> SetInfoLevel(InfoExample,2);
gap> Info(InfoExample + InfoWarning, 2, "hello");
#I  hello
gap> InfoLevel(InfoWarning);
1
```

7.4.6 Customizing `Info` (7.4.5) statements

▷ `SetInfoHandler(`*`infoclass, handler`*`)` (function)

▷ `SetInfoOutput(`*`infoclass, out`*`)` (function)

▷ `SetDefaultInfoOutput(`*`out`*`)` (function)

Returns: nothing

This allows to customize what happens in an `Info(`*`infoclass`*`, `*`level`*`, ...)` statement.

In the first function *`handler`* must be a function with three arguments *`infoclass`*, *`level`*, *`list`*. Here *`list`* is the list containing the third to last argument of the `Info` (7.4.5) call.

The default handler is the function `DefaultInfoHandler`. It prints `"#I "`, then the third and further arguments of the info statement, and finally a `"\n"`.

If the first argument of an `Info` (7.4.5) statement is a sum of Info classes, the handler of the first summand is used.

The file or stream to which `Info` (7.4.5) statements for individual `Info` (7.4.5) classes print can be changed with `SetInfoOutput`. The initial default for all `Info` (7.4.5) classes is the string `"*Print*"` which means the current output file. The default can be changed with `SetDefaultInfoOutput`. The

argument *out* can be a filename or an open stream, the special names "*Print*", "*errout* and "*stdout* are also recognized.

For example, SetDefaultInfoOutput("*errout*"); would send Info (7.4.5) output to standard error, which can be interesting if GAPs output is redirected.

7.4.7 InfoWarning

▷ InfoWarning (global variable)

is an info class to which general warnings are sent at level 1, which is its default level. More specialised warnings are shown via calls of Info (7.4.5) at InfoWarning level 2, e.g. information about the autoloading of GAP packages and the initial line matched when displaying an on-line help topic.

7.5 Assertions

Assertions are used to find errors in algorithms. They test whether intermediate results conform to required conditions and issue an error if not.

7.5.1 SetAssertionLevel

▷ SetAssertionLevel(*lev*) (function)

assigns the global assertion level to *lev*. By default it is zero.

7.5.2 AssertionLevel

▷ AssertionLevel() (function)

returns the current assertion level.

7.5.3 Assert

▷ Assert(*lev, cond[, message]*) (function)

With two arguments, if the global assertion level is at least *lev*, condition *cond* is tested and if it does not return true an error is raised. Thus Assert(lev, *cond*) is equivalent to the code

Example

```
if AssertionLevel() >= lev and not <cond> then
  Error("Assertion failure");
fi;
```

With the *message* argument form of the Assert statement, if the global assertion level is at least *lev*, condition *cond* is tested and if it does not return true then *message* is evaluated and printed.

Assertions are used at various places in the library. Thus turning assertions on can slow code execution significantly.

7.6 Timing

7.6.1 Runtimes

▷ `Runtimes()` (function)

`Runtimes` returns a record with components bound to integers or `fail`. Each integer is the cpu time (processor time) in milliseconds spent by GAP in a certain status:

`user_time`
: cpu time spent with GAP functions (without child processes).

`system_time`
: cpu time spent in system calls, e.g., file access (`fail` if not available).

`user_time_children`
: cpu time spent in child processes (`fail` if not available).

`system_time_children`
: cpu time spent in system calls by child processes (`fail` if not available).

Note that this function is not fully supported on all systems. Only the `user_time` component is (and may on some systems include the system time).

The following example demonstrates tasks which contribute to the different time components:

Example

```
gap> Runtimes(); # after startup
rec( user_time := 3980, system_time := 60, user_time_children := 0,
  system_time_children := 0 )
gap> Exec("cat /usr/bin/*||wc"); # child process with a lot of file access
 893799 7551659 200928302
gap> Runtimes();
rec( user_time := 3990, system_time := 60, user_time_children := 1590,
  system_time_children := 600 )
gap> a:=0;;for i in [1..100000000] do a:=a+1; od; # GAP user time
gap> Runtimes();
rec( user_time := 12980, system_time := 70, user_time_children := 1590,
  system_time_children := 600 )
gap> ?blabla  # first call of help, a lot of file access
Help: no matching entry found
gap> Runtimes();
rec( user_time := 13500, system_time := 440, user_time_children := 1590,
  system_time_children := 600 )
```

7.6.2 Runtime

▷ `Runtime()` (function)

`Runtime` returns the time spent by GAP in milliseconds as an integer. It is the same as the value of the `user_time` component given by `Runtimes` (7.6.1), as explained above.

See `StringTime` (27.9.9) for a translation from milliseconds into hour/minute format.

7.6.3 time

▷ `time` (global variable)

In the read-eval-print loop, `time` stores the time the last command took.

7.7 Profiling

Profiling of code can be used to determine in which parts of a program how much time has been spent and how much memory has been allocated during runtime. The idea is that

- first one switches on profiling for those GAP functions the performance of which one wants to check,
- then one runs some GAP computations,
- then one looks at the profile information collected during these computations,
- then one runs more computations (perhaps clearing all profile information before, see `ClearProfile` (7.7.9)),
- and finally one switches off profiling.

For switching on and off profiling, GAP supports entering a list of functions (see `ProfileFunctions` (7.7.4), `UnprofileFunctions` (7.7.5)) or a list of operations whose methods shall be (un)profiled (`ProfileMethods` (7.7.6), `UnprofileMethods` (7.7.7)), and `DisplayProfile` (7.7.8) can be used to show profile information about functions in a given list.

Besides these functions, `ProfileGlobalFunctions` (7.7.1), `ProfileOperations` (7.7.2), and `ProfileOperationsAndMethods` (7.7.3) can be used for switching on or off profiling for *all* global functions, operations, and operations together with all their methods, respectively, and for showing profile information about these functions.

Note that GAP will perform more slowly when profiling than when not.

7.7.1 ProfileGlobalFunctions

▷ `ProfileGlobalFunctions(`*`[bool]`*`)` (function)

Called with argument `true`, `ProfileGlobalFunctions` starts profiling of all functions that have been declared via `DeclareGlobalFunction` (79.18.7). Old profile information for all these functions is cleared. A function call with the argument `false` stops profiling of all these functions. Recorded information is still kept, so you can display it even after turning the profiling off.

When `ProfileGlobalFunctions` is called without argument, profile information for all global functions is displayed, see `DisplayProfile` (7.7.8).

7.7.2 ProfileOperations

▷ `ProfileOperations(`*`[bool]`*`)` (function)

Called with argument `true`, `ProfileOperations` starts profiling of all operations. Old profile information for all operations is cleared. A function call with the argument `false` stops profiling of

all operations. Recorded information is still kept, so you can display it even after turning the profiling off.

When ProfileOperations is called without argument, profile information for all operations is displayed (see DisplayProfile (7.7.8)).

7.7.3 ProfileOperationsAndMethods

▷ ProfileOperationsAndMethods(*[bool]*) (function)

Called with argument true, ProfileOperationsAndMethods starts profiling of all operations and their methods. Old profile information for these functions is cleared. A function call with the argument false stops profiling of all operations and their methods. Recorded information is still kept, so you can display it even after turning the profiling off.

When ProfileOperationsAndMethods is called without argument, profile information for all operations and their methods is displayed, see DisplayProfile (7.7.8).

7.7.4 ProfileFunctions

▷ ProfileFunctions(*funcs*) (function)

starts profiling for all function in the list *funcs*. You can use ProfileGlobalFunctions (7.7.1) to turn profiling on for all globally declared functions simultaneously.

7.7.5 UnprofileFunctions

▷ UnprofileFunctions(*funcs*) (function)

stops profiling for all function in the list *funcs*. Recorded information is still kept, so you can display it even after turning the profiling off.

7.7.6 ProfileMethods

▷ ProfileMethods(*ops*) (function)

starts profiling of the methods for all operations in the list *ops*.

7.7.7 UnprofileMethods

▷ UnprofileMethods(*ops*) (function)

stops profiling of the methods for all operations in the list *ops*. Recorded information is still kept, so you can display it even after turning the profiling off.

7.7.8 DisplayProfile

▷ DisplayProfile(*[functions,][mincount, mintime]*) (function)
▷ GAPInfo.ProfileThreshold (global variable)

Called without arguments, DisplayProfile displays the profile information for profiled operations, methods and functions. If an argument *functions* is given, only profile information for the functions in the list *functions* is shown. If two integer values *mincount*, *mintime* are given as arguments then the output is restricted to those functions that were called at least *mincount* times or for which the total time spent (see below) was at least *mintime* milliseconds. The defaults for *mincount* and *mintime* are the entries of the list stored in the global variable GAPInfo.ProfileThreshold.

The default value of GAPInfo.ProfileThreshold is [10000, 30].

Profile information is displayed in a list of lines for all functions (including operations and methods) which are profiled. For each function, "count" gives the number of times the function has been called. "self/ms" gives the time (in milliseconds) spent in the function itself, "chld/ms" the time (in milliseconds) spent in profiled functions called from within this function, "stor/kb" the amount of storage (in kilobytes) allocated by the function itself, "chld/kb" the amount of storage (in kilobytes) allocated by profiled functions called from within this function, and "package" the name of the GAP package to which the function belongs; the entry "GAP" in this column means that the function belongs to the GAP library, the entry "(oprt.)" means that the function is an operation (which may belong to several packages), and an empty entry means that FilenameFunc (5.1.4) cannot determine in which file the function is defined.

The list is sorted according to the total time spent in the functions, that is the sum of the values in the columns "self/ms" and "chld/ms".

At the end of the list, two lines are printed that show the total time used and the total memory allocated by the profiled functions not shown in the list (label OTHER) and by all profiled functions (label TOTAL), respectively.

An interactive variant of DisplayProfile is the function BrowseProfile (**Browse: BrowseProfile**) that is provided by the GAP package Browse.

7.7.9 ClearProfile

▷ ClearProfile() (function)

clears all stored profile information.

7.7.10 An Example of Profiling

Let us suppose we want to get information about the computation of the conjugacy classes of a certain permutation group. For that, first we create the group, then we start profiling for all global functions and for all operations and their methods, then we compute the conjugacy classes, and then we stop profiling.

Example

```
gap> g:= PrimitiveGroup( 24, 1 );;
gap> ProfileGlobalFunctions( true );
gap> ProfileOperationsAndMethods( true );
gap> ConjugacyClasses( g );;
gap> ProfileGlobalFunctions( false );
gap> ProfileOperationsAndMethods( false );
```

Now the profile information is available. We can list the information for all profiled functions with DisplayProfile (7.7.8).

Example

```
gap> DisplayProfile();
  count  self/ms  chld/ms  stor/kb  chld/kb  package  function
  17647        0        0      275        0  GAP      BasePoint
  10230        0        0      226        0  (oprt.)  ShallowCopy
  10139        0        0        0        0           PositionSortedOp: for*
  10001        0        0      688        0           UniteSet: for two int*
  10001        8        0       28      688  (oprt.)  UniteSet
  14751       12        0        0        0           =: for two families: *
  10830        8        4      182      276  GAP      Concatenation
   2700       20       12      313       55  GAP      AddRefinement
   2444       28        4     3924      317  GAP      ConjugateStabChain
   4368        0       32        7      714  (oprt.)  Size
   2174       32        4     1030      116  GAP      List
    585        4       32       45      742  GAP      RRefine
   1532       32        8      194       56  GAP      AddGeneratorsExtendSc*
   1221        8       32      349      420  GAP      Partition
 185309       28       12        0        0  (oprt.)  Length
    336        4       40       95      817  GAP      ExtendSeriesPermGroup
      4       28       20      488      454  (oprt.)  Sortex
   2798        0       52       54      944  GAP      StabChainForcePoint
    560        4       48       83      628  GAP      StabChainSwap
    432       16       40      259      461  GAP      SubmagmaWithInversesNC
 185553       48        8      915       94  (oprt.)  Add
     26        0       64        0     2023  (oprt.)  CentralizerOp
     26        0       64        0     2023  GAP      CentralizerOp: perm g*
     26        0       64        0     2023  GAP      Centralizer: try to e*
    152        4       64        0     2024  (oprt.)  Centralizer
   1605        0       68        0     2032  (oprt.)  StabilizerOfExternalS*
     26        0       68        0     2024  GAP      Meth(StabilizerOfExte*
    382        0       96       69     1922  GAP      TryPcgsPermGroup
   5130        4       96      309     3165  GAP      ForAll
   7980       24      116      330     6434  GAP      ChangeStabChain
  12076       12      136      351     6478  GAP      ProcessFixpoint
    192        0      148        4     3029  GAP      StabChainMutable: cal*
   2208        4      148        3     3083  (oprt.)  StabChainMutable
    217        0      160        0     3177  (oprt.)  StabChainOp
    217       12      148       60     3117  GAP      StabChainOp: group an*
    216       36      464      334    12546  GAP      PartitionBacktrack
   1479       12      668      566    18474  GAP      RepOpElmTuplesPermGro*
   1453       12      684       56    18460  GAP      in: perm class rep
    126        0      728       13    19233  GAP      ConjugacyClassesTry
      1        0      736        0    19671  GAP      ConjugacyClassesByRan*
      2        0      736        2    19678  (oprt.)  ConjugacyClasses
      1        0      736        0    19675  GAP      ConjugacyClasses: per*
  13400     1164        0        0        0  (oprt.)  Position
            484             12052                    OTHER
           2048             23319                    TOTAL
```

We can restrict the list to global functions with `ProfileGlobalFunctions` (7.7.1).

Example

```
gap> ProfileGlobalFunctions();
  count  self/ms  chld/ms  stor/kb  chld/kb  package  function
```

```
  17647        0        0      275        0  GAP      BasePoint
  10830        8        4      182      276  GAP      Concatenation
   2700       20       12      313       55  GAP      AddRefinement
   2444       28        4     3924      317  GAP      ConjugateStabChain
   2174       32        4     1030      116  GAP      List
    585        4       32       45      742  GAP      RRefine
   1532       32        8      194       56  GAP      AddGeneratorsExtendSc*
   1221        8       32      349      420  GAP      Partition
    336        4       40       95      817  GAP      ExtendSeriesPermGroup
   2798        0       52       54      944  GAP      StabChainForcePoint
    560        4       48       83      628  GAP      StabChainSwap
    432       16       40      259      461  GAP      SubmagmaWithInversesNC
    382        0       96       69     1922  GAP      TryPcgsPermGroup
   5130        4       96      309     3165  GAP      ForAll
   7980       24      116      330     6434  GAP      ChangeStabChain
  12076       12      136      351     6478  GAP      ProcessFixpoint
    216       36      464      334    12546  GAP      PartitionBacktrack
   1479       12      668      566    18474  GAP      RepOpElmTuplesPermGro*
    126        0      728       13    19233  GAP      ConjugacyClassesTry
      1        0      736        0    19671  GAP      ConjugacyClassesByRan*
           1804            14536                      OTHER
           2048            23319                      TOTAL
```

We can restrict the list to operations with `ProfileOperations` (7.7.2).

Example

```
gap> ProfileOperations();
  count  self/ms  chld/ms  stor/kb  chld/kb  package  function
  10230        0        0      226        0  (oprt.)  ShallowCopy
  10001        8        0       28      688  (oprt.)  UniteSet
   4368        0       32        7      714  (oprt.)  Size
 185309       28       12        0        0  (oprt.)  Length
      4       28       20      488      454  (oprt.)  Sortex
 185553       48        8      915       94  (oprt.)  Add
     26        0       64        0     2023  (oprt.)  CentralizerOp
    152        4       64        0     2024  (oprt.)  Centralizer
   1605        0       68        0     2032  (oprt.)  StabilizerOfExternalS*
   2208        4      148        3     3083  (oprt.)  StabChainMutable
    217        0      160        0     3177  (oprt.)  StabChainOp
      2        0      736        2    19678  (oprt.)  ConjugacyClasses
  13400     1164        0        0        0  (oprt.)  Position
            764            21646                      OTHER
           2048            23319                      TOTAL
```

We can restrict the list to operations and their methods with `ProfileOperationsAndMethods` (7.7.3).

Example

```
gap> ProfileOperationsAndMethods();
  count  self/ms  chld/ms  stor/kb  chld/kb  package  function
  10230        0        0      226        0  (oprt.)  ShallowCopy
  10139        0        0        0        0           PositionSortedOp: for*
  10001        0        0      688        0           UniteSet: for two int*
```

```
  10001        8        0       28      688  (oprt.)  UniteSet
  14751       12        0        0        0           =: for two families: *
   4368        0       32        7      714  (oprt.)  Size
 185309       28       12        0        0  (oprt.)  Length
      4       28       20      488      454  (oprt.)  Sortex
 185553       48        8      915       94  (oprt.)  Add
     26        0       64        0     2023  (oprt.)  CentralizerOp
     26        0       64        0     2023  GAP      CentralizerOp: perm g*
     26        0       64        0     2023  GAP      Centralizer: try to e*
    152        4       64        0     2024  (oprt.)  Centralizer
   1605        0       68        0     2032  (oprt.)  StabilizerOfExternalS*
     26        0       68        0     2024  GAP      Meth(StabilizerOfExte*
    192        0      148        4     3029  GAP      StabChainMutable: cal*
   2208        4      148        3     3083  (oprt.)  StabChainMutable
    217        0      160        0     3177  (oprt.)  StabChainOp
    217       12      148       60     3117  GAP      StabChainOp: group an*
   1453       12      684       56    18460  GAP      in: perm class rep
      2        0      736        2    19678  (oprt.)  ConjugacyClasses
      1        0      736        0    19675  GAP      ConjugacyClasses: per*
  13400     1164        0        0        0  (oprt.)  Position
            728             20834                     OTHER
           2048             23319                     TOTAL
```

Finally, we can restrict the list to explicitly given functions with `DisplayProfile` (7.7.8), by entering the list of functions as an argument.

Example

```
gap> DisplayProfile( [ StabChainOp, Centralizer ] );
  count  self/ms  chld/ms  stor/kb  chld/kb  package  function
    152       4       64        0     2024  (oprt.)  Centralizer
    217       0      160        0     3177  (oprt.)  StabChainOp
           2044             23319                     OTHER
           2048             23319                     TOTAL
```

7.7.11 DisplayCacheStats

▷ `DisplayCacheStats()` (function)

displays statistics about the different caches used by the method selection.

7.7.12 ClearCacheStats

▷ `ClearCacheStats()` (function)

clears all statistics about the different caches used by the method selection.

7.8 Information about the version used

The global variable `GAPInfo.Version` (see `GAPInfo` (3.5.1)) contains the version number of the version of GAP. Its value can be checked other version number using `CompareVersionNumbers` (76.3.6).

To produce sample citations for the used version of GAP or for a package available in this GAP installation, use Cite (76.3.15).

If you wish to report a problem to GAP Support or GAP Forum, it may be useful to not only report the version used, but also to include the GAP banner displays the information about the architecture for which the GAP binary is built, used libraries and loaded packages.

7.9 Test Files

Test files are used to check that GAP produces correct results in certain computations. A selection of test files for the library can be found in the tst directory of the GAP distribution.

7.9.1 ReadTest

▷ ReadTest(*string*) (operation)

reads the test file with name *string*. The test file should contain lines of GAP input and corresponding output. The input lines start with the gap> prompt (or with the > prompt if commands exceed one line). The output is exactly as would result from typing in the input interactively to a GAP session (on a screen with 80 characters per line).

Optionally, START_TEST (7.9.2) and STOP_TEST (7.9.2) may be used in the beginning and end of test files to reinitialize the caches and the global random number generator in order to be independent of the reading order of several test files. Furthermore, START_TEST (7.9.2) increases the assertion level for the time of the test, and STOP_TEST (7.9.2) sets the proportionality factor that is used to output a "GAPstone" speed ranking after the file has been completely processed.

7.9.2 Starting and stopping test

▷ START_TEST(*id*) (function)

▷ STOP_TEST(*file, fac*) (function)

START_TEST (7.9.2) and STOP_TEST (7.9.2) may be optionally used in files that are read via ReadTest (7.9.1). If used, START_TEST (7.9.2) reinitialize the caches and the global random number generator, in order to be independent of the reading order of several test files. Furthermore, the assertion level (see Assert (7.5.3)) is set to 2 by START_TEST (7.9.2) and set back to the previous value in the subsequent STOP_TEST (7.9.2) call.

To use these options, a test file should be started with a line

Example

```
gap> START_TEST( "arbitrary identifier string" );
```

(Note that the gap> prompt is part of the line!)
and should be finished with a line

Example

```
gap> STOP_TEST( "filename", 10000 );
```

Here the string "filename" should give the name of the test file. The number is a proportionality factor that is used to output a "GAPstone" speed ranking after the file has been completely processed.

For the files provided with the distribution this scaling is roughly equalized to yield the same numbers as produced by the test file `tst/combinat.tst`.

Note that the functions in `tst/testutil.g` temporarily replace `STOP_TEST` (7.9.2) before they call `ReadTest` (7.9.1).

If you want to run a quick test of your GAP installation (though this is not required), you can read in a test script that exercises some GAP's capabilities.

Example

```
gap> Read( Filename( DirectoriesLibrary( "tst" ), "testinstall.g" ) );
```

The test requires about 750MB of memory and runs about one minute on an Intel Core 2 Duo / 2.53 GHz machine. You will get a large number of lines with output about the progress of the tests.

Example

```
test file         GAP4stones     time(msec)
-----------------------------------------
testing: ................./gap4r5/tst/zlattice.tst
zlattice.tst               0              0
testing: ................./gap4r5/tst/gaussian.tst
gaussian.tst               0             10
[ further lines deleted ]
```

If you want to run a more advanced check (this is not required and make take up to an hour), you can read `testall.g` which is an extended test script performing all tests from the `tst` directory.

Example

```
gap> Read( Filename( DirectoriesLibrary( "tst" ), "testall.g" ) );
```

The test requires about 750MB of memory and runs about one hour on an Intel Core 2 Duo / 2.53 GHz machine, and produces an output similar to the `testinstall.g` test.

7.9.3 Test

▷ `Test(`*`fname[, optrec]`*`)` (function)

Returns: `true` or `false`.

The argument *`fname`* must be the name of a file or an open input stream. The content of this file or stream should contain GAP input and output. The function `Test` runs the input lines, compares the actual output with the output stored in *`fname`* and reports differences. With an optional record as argument *`optrec`* details of this process can be adjusted.

More precisely, the content of *`fname`* must have the following format.
Lines starting with `"gap> "` are considered as GAP input, they can be followed by lines starting with `"> "` if the input is continued over several lines.
To allow for comments in *`fname`* the following lines are ignored by default: lines at the beginning of *`fname`* that start with `"#"`, and one empty line together with one or more lines starting with `"#"`.
All other lines are considered as GAP output from the preceding GAP input.

By default the actual GAP output is compared exactly with the stored output, and if these are different some information about the differences is printed.

If any differences are found then `Test` returns `false`, otherwise `true`.

If the optional argument *`optrec`* is given it must be a record. The following components of *`optrec`* are recognized and can change the default behaviour of `Test`:

`ignoreComments`
: If set to `false` then no lines in *`fname`* are ignored as explained above (default is `true`).

`width`
: The screen width used for the new output (default is `80`).

`compareFunction`
: This must be a function that gets two strings as input, the newly generated and the stored output of some GAP input. The function must return `true` or `false`, indicating if the strings should be considered equivalent or not. By default \= (31.11.1) is used.
Two strings are recognized as abbreviations in this component: `"uptowhitespace"` checks if the two strings become equal after removing all white space. And `"uptonl"` compares the string up to trailing newline characters.

`reportDiff`
: A function that gets six arguments and reports a difference in the output: the GAP input, the expected GAP output, the newly generated output, the name of tested file, the line number of the input, the time to run the input. (The default is demonstrated in the example below.)

`rewriteToFile`
: If this is bound to a string it is considered as a file name and that file is written with the same input and comment lines as *`fname`* but the output substituted by the newly generated version (default is `false`).

`writeTimings`
: If this is bound to a string it is considered as a file name, that file is written and contains timing information for each input in *`fname`*.

`compareTimings`
: If this is bound to a string it is considered as name of a file to which timing information was stored via `writeTimings` in a previous call. The new timings are compared to the stored ones. By default only commands which take more than a threshold of 100 milliseconds are considered, and only differences of more than 20% are considered significant. These defaults can be overwritten by assigning a list `[timingfile, threshold, percentage]` to this component. (The default of `compareTimings` is `false`.)

`reportTimeDiff`
: This component can be used to overwrite the default function to display timing differences. It must be a function with 5 arguments: GAP input, name of test file, line number, stored time, new time.

`ignoreSTOP_TEST`
: By default set to `true`, in that case the output of GAP input starting with `"STOP_TEST"` is not checked.

`showProgress`
: If this is `true` then GAP prints position information and the input line before it is processed (default is `false`).

`subsWindowsLineBreaks`

If this is `true` then GAP substitutes DOS/Windows style line breaks "\r\n" by UNIX style line breaks "\n" after reading the test file. (default is `true`).

Example

```
gap> tnam := Filename(DirectoriesLibrary(), "../doc/ref/demo.tst");;
gap> mask := function(str) return Concatenation("| ",
>          JoinStringsWithSeparator(SplitString(str, "\n", ""), "\n| "),
>          "\n"); end;;
gap> Print(mask(StringFile(tnam)));
| # this is a demo file for the 'Test' function
| #
| gap> g := Group((1,2), (1,2,3));
| Group([ (1,2), (1,2,3) ])
|
| # another comment following an empty line
| # the following fails:
| gap> a := 13+29;
| 41
gap> ss := InputTextString(StringFile(tnam));;
gap> Test(ss);
########> Diff in test stream, line 8:
# Input is:
a := 13+29;
# Expected output:
41
# But found:
42
########
false
gap> RewindStream(ss);
true
gap> dtmp := DirectoryTemporary();;
gap> ftmp := Filename(dtmp,"demo.tst");;
gap> Test(ss, rec(reportDiff := Ignore, rewriteToFile := ftmp));
false
gap> Test(ftmp);
true
gap> Print(mask(StringFile(ftmp)));
| # this is a demo file for the 'Test' function
| #
| gap> g := Group((1,2), (1,2,3));
| Group([ (1,2), (1,2,3) ])
|
| # another comment following an empty line
| # the following fails:
| gap> a := 13+29;
| 42
```

7.10 Debugging Recursion

The GAP interpreter monitors the level of nesting of GAP functions during execution. By default, whenever this nesting reaches a multiple of 5000, GAP enters a break loop (6.4) allowing you to terminate the calculation, or enter RETURN; to continue it.

Example

```
gap> dive:= function(depth) if depth>1 then dive(depth-1); fi; return; end;
function( depth ) ... end
gap> dive(100);
gap> OnBreak:= function() Where(1); end; # shorter traceback
function(  ) ... end
gap> dive(6000);
recursion depth trap (5000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
gap> dive(11000);
recursion depth trap (5000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
recursion depth trap (10000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
gap>
```

This behaviour can be controlled using the following procedure.

7.10.1 SetRecursionTrapInterval

▷ SetRecursionTrapInterval(*interval*) (function)

interval must be a non-negative small integer (between 0 and 2^{28}). An *interval* of 0 suppresses the monitoring of recursion altogether. In this case excessive recursion may cause GAP to crash.

Example

```
gap> dive:= function(depth) if depth>1 then dive(depth-1); fi; return; end;
function( depth ) ... end
gap> SetRecursionTrapInterval(1000);
gap> dive(2500);
recursion depth trap (1000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
recursion depth trap (2000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
gap> SetRecursionTrapInterval(-1);
SetRecursionTrapInterval( <interval> ): <interval> must be a non-negative smal\
l integer
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <interval> via 'return <interval>;' to continue
brk> return ();
SetRecursionTrapInterval( <interval> ): <interval> must be a non-negative smal\
l integer
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <interval> via 'return <interval>;' to continue
brk> return 0;
gap> dive(20000);
gap> dive(2000000);
Segmentation fault
```

7.11 Global Memory Information

The GAP environment provides automatic memory management, so that the programmer does not need to concern themselves with allocating space for objects, or recovering space when objects are

no longer needed. The component of the kernel which provides this is called GASMAN (GAP Storage MANager). Messages reporting garbage collections performed by GASMAN can be switched on by the -g command line option (see section 3.1). There are also some facilities to access information from GASMAN from GAP programs.

7.11.1 GasmanStatistics

▷ GasmanStatistics() (function)

GasmanStatistics returns a record containing some information from the garbage collection mechanism. The record may contain up to four components: full, partial, npartial, and nfull.

The full component will be present if a full garbage collection has taken place since GAP started. It contains information about the most recent full garbage collection. It is a record, with six components: livebags contains the number of bags which survived the garbage collection; livekb contains the total number of kilobytes occupied by those bags; deadbags contains the total number of bags which were reclaimed by that garbage collection and all the partial garbage collections preceding it, since the previous full garbage collection; deadkb contains the total number of kilobytes occupied by those bags; freekb reports the total number of kilobytes available in the GAP workspace for new objects and totalkb the actual size of the workspace.

These figures should be viewed with some caution. They are stored internally in fixed length integer formats, and deadkb and deadbags are liable to overflow if there are many partial collections before a full collection. Also, note that livekb and freekb will not usually add up to totalkb. The difference is essentially the space overhead of the memory management system.

The partial component will be present if there has been a partial garbage collection since the last full one. It is also a record with the same six components as full. In this case deadbags and deadkb refer only to the number and total size of the garbage bags reclaimed in this partial garbage collection and livebagsand livekb only to the numbers and total size of the young bags that were considered for garbage collection, and survived.

The npartial and nfull components will contain the number of full and partial garbage collections performed since GAP started.

7.11.2 GasmanMessageStatus

▷ GasmanMessageStatus() (function)
▷ SetGasmanMessageStatus(*stat*) (function)

GasmanMessageStatus returns one of the strings "none", "full", or "all", depending on whether the garbage collector is currently set to print messages on no collections, full collections only, or all collections, respectively.

Calling SetGasmanMessageStatus with the argument *stat*, which should be one of the three strings mentioned above, sets the garbage collector messaging level.

7.11.3 GasmanLimits

▷ GasmanLimits() (function)

`GasmanLimits` returns a record with three components: `min` is the minimum workspace size as set by the `-m` command line option in kilobytes. The workspace size will never be reduced below this by the garbage collector. `max` is the maximum workspace size, as set by the '-o' command line option, also in kilobytes. If the workspace would need to grow past this point, GAP will enter a break loop to warn the user. A value of 0 indicates no limit. `kill` is the absolute maximum, set by the `-K` command line option. The workspace will never be allowed to grow past this limit.

Chapter 8

Options Stack

GAP supports a *global options system*. This is intended as a way for the user to provide guidance to various algorithms that might be used in a computation. Such guidance should not change mathematically the specification of the computation to be performed, although it may change the algorithm used. A typical example is the selection of a strategy for the Todd-Coxeter coset enumeration procedure. An example of something not suited to the options mechanism is the imposition of exponent laws in the p-Quotient algorithm.

The basis of this system is a global stack of records. All the entries of each record are thought of as options settings, and the effective setting of an option is given by the topmost record in which the relevant field is bound.

The reason for the choice of a stack is the intended pattern of use:

```
PushOptions( rec( stuff ) );
DoSomething( args );
PopOptions();
```

This can be abbreviated, to `DoSomething( args : stuff );` with a small additional abbreviation of *stuff* permitted. See 4.11.2 for details. The full form can be used where the same options are to run across several calls, or where the `DoSomething` procedure is actually an infix operator, or other function with special syntax.

An alternative to this system is the use of additional optional arguments in procedure calls. This is not felt to be sufficient because many procedure calls might cause, for example, a coset enumeration and each would need to make provision for the possibility of extra arguments. In this system the options are pushed when the user-level procedure is called, and remain in effect (unless altered) for all procedures called by it.

Note that in some places in the system optional records containing options which are valid only for the immediate function or method call are in fact used.

8.1 Functions Dealing with the Options Stack

8.1.1 PushOptions

▷ `PushOptions(`*options, record*`)` (function)

This function pushes a record of options onto the global option stack. Note that `PushOptions( rec( `*opt*`:= fail ) )` has the effect of resetting the option *opt*, since an option that has never been

set has the value `fail` returned by `ValueOption` (8.1.5).

Note that there is no check for misspelt or undefined options.

8.1.2 PopOptions

▷ `PopOptions()` (function)

This function removes the top-most options record from the options stack if there is one.

8.1.3 ResetOptionsStack

▷ `ResetOptionsStack()` (function)

unbinds (i.e. removes) all the options records from the options stack.

Note: `ResetOptionsStack` should *not* be used within a function. Its intended use is to clean up the options stack in the event that the user has `quit` from a `break` loop, so leaving a stack of no-longer-needed options (see 6.4.1).

8.1.4 OnQuit

▷ `OnQuit()` (function)

called when a user selects to `quit;` a break loop entered via execution of `Error` (6.6.1). As GAP starts up, `OnQuit` is defined to do nothing, in case an error is encountered during GAP start-up. Later in the loading process we redefine `OnQuit` to do a variant of `ResetOptionsStack` (8.1.3) to ensure the options stack is empty after a user quits an `Error` (6.6.1)-induced break loop. (`OnQuit` differs from `ResetOptionsStack` (8.1.3) in that it warns when it does something rather than the other way round.) Currently, `OnQuit` is not advertised, since exception handling may make it obsolete.

8.1.5 ValueOption

▷ `ValueOption(`*`opt`*`)` (function)

This function is a method for accessing the options stack without changing it; *`opt`* should be the name of an option, i.e. a string. A function which makes decisions that might be affected by options should examine the result of `ValueOption`. If *`opt`* is currently not set then `fail` is returned.

8.1.6 DisplayOptionsStack

▷ `DisplayOptionsStack()` (function)

This function prints a human-readable display of the complete options stack.

8.1.7 InfoOptions

▷ `InfoOptions` (info class)

This info class can be used to enable messages about options being changed (level 1) or accessed (level 2).

8.2 Options Stack – an Example

The example below shows simple manipulation of the Options Stack, first using `PushOptions` (8.1.1) and `PopOptions` (8.1.2) and then using the special function calling syntax.

Example

```
gap> foo := function()
> Print("myopt1 = ", ValueOption("myopt1"),
>       " myopt2 = ",ValueOption("myopt2"),"\n");
> end;
function(  ) ... end
gap> foo();
myopt1 = fail myopt2 = fail
gap> PushOptions(rec(myopt1 := 17));
gap> foo();
myopt1 = 17 myopt2 = fail
gap> DisplayOptionsStack();
[ rec(
      myopt1 := 17 ) ]
gap> PopOptions();
gap> foo();
myopt1 = fail myopt2 = fail
gap> foo( : myopt1, myopt2 := [Z(3),"aardvark"]);
myopt1 = true myopt2 = [ Z(3), "aardvark" ]
gap> DisplayOptionsStack();
[  ]
gap>
```

Chapter 9

Files and Filenames

Files are identified by filenames, which are represented in GAP as strings. Filenames can be created directly by the user or a program, but of course this is operating system dependent.

Filenames for some files can be constructed in a system independent way using the following functions. This is done by first getting a directory object for the directory the file shall reside in, and then constructing the filename. However, it is sometimes necessary to construct filenames of files in subdirectories relative to a given directory object. In this case the directory separator is *always* / even under DOS or MacOS.

Section 9.3 describes how to construct directory objects for the common GAP and system directories. Using the command `Filename` (9.4.1) it is possible to construct a filename pointing to a file in these directories. There are also functions to test for accessibility of files, see 9.6.

9.1 Portability

For portability filenames and directory names should be restricted to at most 8 alphanumerical characters optionally followed by a dot `.` and between 1 and 3 alphanumerical characters. Upper case letters should be avoided because some operating systems do not make any distinction between case, so that `NaMe`, `Name` and `name` all refer to the same file whereas some operating systems are case sensitive. To avoid problems only lower case characters should be used.

Another function which is system-dependent is `LastSystemError` (9.1.1).

9.1.1 LastSystemError

▷ `LastSystemError()` (function)

`LastSystemError` returns a record describing the last system error that has occurred. This record contains at least the component `message` which is a string. This message is, however, highly operating system dependent and should only be used as an informational message for the user.

9.2 GAP Root Directories

When GAP is started it determines a list of directories which we call the *GAP root directories*. In a running GAP session this list can be found in `GAPInfo.RootPaths`.

The core part of GAP knows which files to read relative to its root directories. For example when GAP wants to read its library file `lib/group.gd`, it appends this path to each path in `GAPInfo.RootPaths` until it finds the path of an existing file. The first file found this way is read.

Furthermore, GAP looks for available packages by examining the subdirectories `pkg/` in each of the directories in `GAPInfo.RootPaths`.

The root directories are specified via one or several of the `-l paths` command line options, see 3.1. Furthermore, by default GAP automatically prepends a user specific GAP root directory to the list; this can be avoided by calling GAP with the `-r` option. The name of this user specific directory depends on your operating system, it can be found in `GAPInfo.UserGapRoot`. This directory can be used to tell GAP about personal preferences, to always load some additional code, to install additional packages, or to overwrite some GAP files. See 3.2 for more information how to do this.

9.3 Directories

9.3.1 IsDirectory

▷ `IsDirectory(`*obj*`)` (Category)

`IsDirectory` is a category of directories.

9.3.2 Directory

▷ `Directory(`*string*`)` (operation)

returns a directory object for the string *string*. `Directory` understands `"."` for "current directory", that is, the directory in which GAP was started. It also understands absolute paths.

If the variable `GAPInfo.UserHome` is defined (this may depend on the operating system) then `Directory` understands a string with a leading `~` (tilde) character for a path relative to the user's home directory (but a string beginning with `"~other_user"` is *not* interpreted as a path relative to `other_user`'s home directory, as in a UNIX shell).

Paths are otherwise taken relative to the current directory.

9.3.3 DirectoryTemporary

▷ `DirectoryTemporary()` (function)

returns a directory object in the category `IsDirectory` (9.3.1) for a *new* temporary directory. This is guaranteed to be newly created and empty immediately after the call to `DirectoryTemporary`. GAP will make a reasonable effort to remove this directory upon termination of the GAP job that created the directory.

If `DirectoryTemporary` is unable to create a new directory, `fail` is returned. In this case `LastSystemError` (9.1.1) can be used to get information about the error.

9.3.4 DirectoryCurrent

▷ `DirectoryCurrent()` (function)

returns the directory object for the current directory.

9.3.5 DirectoriesLibrary

▷ DirectoriesLibrary([name]) (function)

DirectoriesLibrary returns the directory objects for the GAP library *name* as a list. *name* must be one of "lib" (the default), "doc", "tst", and so on.

The string "" is also legal and with this argument DirectoriesLibrary returns the list of GAP root directories. The return value of this call differs from GAPInfo.RootPaths in that the former is a list of directory objects and the latter a list of strings.

The directory *name* must exist in at least one of the root directories, otherwise fail is returned.

As the files in the GAP root directories (see 9.2) can be distributed into different directories in the filespace a list of directories is returned. In order to find an existing file in a GAP root directory you should pass that list to Filename (9.4.1) as the first argument. In order to create a filename for a new file inside a GAP root directory you should pass the first entry of that list. However, creating files inside the GAP root directory is not recommended, you should use DirectoryTemporary (9.3.3) instead.

9.3.6 DirectoriesSystemPrograms

▷ DirectoriesSystemPrograms() (function)

DirectoriesSystemPrograms returns the directory objects for the list of directories where the system programs reside, as a list. Under UNIX this would usually represent $PATH.

9.3.7 DirectoryContents

▷ DirectoryContents(*dir*) (function)

This function returns a list of filenames/directory names that reside in the directory *dir*. The argument *dir* can either be given as a string indicating the name of the directory or as a directory object (see IsDirectory (9.3.1)). It is an error, if such a directory does not exist.

The ordering of the list entries can depend on the operating system.

An interactive way to show the contents of a directory is provided by the function BrowseDirectory (**Browse: BrowseDirectory**) from the GAP package Browse.

9.3.8 DirectoryDesktop

▷ DirectoryDesktop() (operation)

returns a directory object for the users desktop directory as defined on many modern operating systems. The function is intended to provide a cross-platform interface to a directory that is easily accessible by the user. Under Unix systems (including Mac OS X) this will be the Desktop directory in the users home directory if it exists, and the users home directory otherwise. Under Windows it will the the users Desktop folder (or the appropriate name under different languages).

9.3.9 DirectoryHome

▷ `DirectoryHome()` (operation)

returns a directory object for the users home directory, defined as a directory in which the user will typically have full read and write access. The function is intended to provide a cross-platform interface to a directory that is easily accessible by the user. Under Unix systems (including Mac OS X) this will be the usual user home directory. Under Windows it will the the users `My Documents` folder (or the appropriate name under different languages).

9.4 File Names

9.4.1 Filename

▷ `Filename(`*`dir, name`*`)` (operation)
▷ `Filename(`*`list-of-dirs, name`*`)` (operation)

If the first argument is a directory object *`dir`*, `Filename` returns the (system dependent) filename as a string for the file with name *`name`* in the directory *`dir`*. `Filename` returns the filename regardless of whether the directory contains a file with name *`name`* or not.

If the first argument is a list *`list-of-dirs`* (possibly of length 1) of directory objects, then `Filename` searches the directories in order, and returns the filename for the file *`name`* in the first directory which contains a file *`name`* or `fail` if no directory contains a file *`name`*.

For example, in order to locate the system program `date` use `DirectoriesSystemPrograms` (9.3.6) together with the second form of `Filename`.

Example

```
gap> path := DirectoriesSystemPrograms();;
gap> date := Filename( path, "date" );
"/bin/date"
```

In order to locate the library file `files.gd` use `DirectoriesLibrary` (9.3.5) together with the second form of `Filename`.

Example

```
gap> path := DirectoriesLibrary();;
gap> Filename( path, "files.gd" );
"./lib/files.gd"
```

In order to construct filenames for new files in a temporary directory use `DirectoryTemporary` (9.3.3) together with the first form of `Filename`.

Example

```
gap> tmpdir := DirectoryTemporary();;
gap> Filename( [ tmpdir ], "file.new" );
fail
gap> Filename( tmpdir, "file.new" );
"/var/tmp/tmp.0.021738.0001/file.new"
```

9.5 Special Filenames

The special filename `"*stdin*"` denotes the standard input, i.e., the stream through which the user enters commands to GAP. The exact behaviour of reading from `"*stdin*"` is operating system dependent, but usually the following happens. If GAP was started with no input redirection, statements are read from the terminal stream until the user enters the end of file character, which is usually CTRL-D. Note that terminal streams are special, in that they may yield ordinary input *after* an end of file. Thus when control returns to the main read-eval-print loop the user can continue with GAP. If GAP was started with an input redirection, statements are read from the current position in the input file up to the end of the file. When control returns to the main read eval view loop the input stream will still return end of file, and GAP will terminate.

The special filename `"*errin*"` denotes the stream connected to the UNIX `stderr` output. This stream is usually connected to the terminal, even if the standard input was redirected, unless the standard error stream was also redirected, in which case opening of `"*errin*"` fails.

The special filename `"*stdout*"` can be used to print to the standard output.

The special filename `"*errout*"` can be used to print to the standard error output file, which is usually connected to the terminal, even if the standard output was redirected.

9.6 File Access

When the following functions return `false` one can use `LastSystemError` (9.1.1) to find out the reason (as provided by the operating system), see the examples.

9.6.1 IsExistingFile

▷ `IsExistingFile(`*`name-file`*`)` (function)

`IsExistingFile` returns `true` if a file with the filename *`name-file`* exists and can be seen by the GAP process. Otherwise `false` is returned.

Example

```
gap> IsExistingFile( "/bin/date" );     # file ‘/bin/date’ exists
true
gap> IsExistingFile( "/bin/date.new" ); # non existing ‘/bin/date.new’
false
gap> IsExistingFile( "/bin/date/new" ); # ‘/bin/date’ is not a directory
false
gap> LastSystemError().message;
"Not a directory"
```

9.6.2 IsReadableFile

▷ `IsReadableFile(`*`name-file`*`)` (function)

`IsReadableFile` returns `true` if a file with the filename *`name-file`* exists *and* the GAP process has read permissions for the file, or `false` if this is not the case.

Example

```
gap> IsReadableFile( "/bin/date" );     # file ‘/bin/date’ is readable
true
```

```
gap> IsReadableFile( "/bin/date.new" ); # non-existing ‘/bin/date.new’
false
gap> LastSystemError().message;
"No such file or directory"
```

9.6.3 IsWritableFile

▷ IsWritableFile(*name-file*) (function)

IsWritableFile returns true if a file with the filename *name-file* exists *and* the GAP process has write permissions for the file, or false if this is not the case.

Example

```
gap> IsWritableFile( "/bin/date" );  # file ‘/bin/date’ is not writable
false
```

9.6.4 IsExecutableFile

▷ IsExecutableFile(*name-file*) (function)

IsExecutableFile returns true if a file with the filename *name-file* exists *and* the GAP process has execute permissions for the file, or false if this is not the case. Note that execute permissions do not imply that it is possible to execute the file, e.g., it may only be executable on a different machine.

Example

```
gap> IsExecutableFile( "/bin/date" );   # ... but executable
true
```

9.6.5 IsDirectoryPath

▷ IsDirectoryPath(*name-file*) (function)

IsDirectoryPath returns true if the file with the filename *name-file* exists *and* is a directory, and false otherwise. Note that this function does not check if the GAP process actually has write or execute permissions for the directory. You can use IsWritableFile (9.6.3), resp. IsExecutableFile (9.6.4) to check such permissions.

9.7 File Operations

9.7.1 Read

▷ Read(*name-file*) (operation)

reads the input from the file with the filename *name-file*, which must be given as a string.

Read first opens the file *name-file*. If the file does not exist, or if GAP cannot open it, e.g., because of access restrictions, an error is signalled.

Then the contents of the file are read and evaluated, but the results are not printed. The reading and evaluations happens exactly as described for the main loop (see 6.1).

If a statement in the file causes an error a break loop is entered (see 6.4). The input for this break loop is not taken from the file, but from the input connected to the `stderr` output of GAP. If `stderr` is not connected to a terminal, no break loop is entered. If this break loop is left with `quit` (or CTRL-D), GAP exits from the `Read` command, and from all enclosing `Read` commands, so that control is normally returned to an interactive prompt. The `QUIT` statement (see 6.7) can also be used in the break loop to exit GAP immediately.

Note that a statement must not begin in one file and end in another. I.e., *eof* (*e*nd-*o*f-*f*ile) is not treated as whitespace, but as a special symbol that must not appear inside any statement.

Note that one file may very well contain a read statement causing another file to be read, before input is again taken from the first file. There is an upper limit of 15 on the number of files that may be open simultaneously.

9.7.2 ReadAsFunction

▷ `ReadAsFunction(`*`name-file`*`)` (operation)

reads the file with filename *`name-file`* as a function and returns this function.

Example

Suppose that the file `/tmp/example.g` contains the following

Example

```
local a;

a := 10;
return a*10;
```

Reading the file as a function will not affect a global variable `a`.

Example

```
gap> a := 1;
1
gap> ReadAsFunction("/tmp/example.g")();
100
gap> a;
1
```

9.7.3 PrintTo and AppendTo

▷ `PrintTo(`*`name-file[, obj1, ...]`*`)` (function)
▷ `AppendTo(`*`name-file[, obj1, ...]`*`)` (function)

`PrintTo` works like `Print` (6.3.4), except that the arguments *`obj1`*, ... (if present) are printed to the file with the name *`name-file`* instead of the standard output. This file must of course be writable by GAP. Otherwise an error is signalled. Note that `PrintTo` will *overwrite* the previous contents of this file if it already existed; in particular, `PrintTo` with just the *`name-file`* argument empties that file.

`AppendTo` works like `PrintTo`, except that the output does not overwrite the previous contents of the file, but is appended to the file.

There is an upper limit of 15 on the number of output files that may be open simultaneously.

Note that one should be careful not to write to a logfile (see `LogTo` (9.7.4)) with `PrintTo` or `AppendTo`.

9.7.4 LogTo

▷ LogTo(*name-file*) (operation)
▷ LogTo() (operation)

Calling LogTo with a string *name-file* causes the subsequent interaction to be logged to the file with the name *name-file*, i.e., everything you see on your terminal will also appear in this file. (LogTo (10.4.5) may also be used to log to a stream.) This file must of course be writable by GAP, otherwise an error is signalled. Note that LogTo will overwrite the previous contents of this file if it already existed.

Called without arguments, LogTo stops logging to a file or stream.

9.7.5 InputLogTo

▷ InputLogTo(*name-file*) (operation)
▷ InputLogTo() (operation)

Calling InputLogTo with a string *name-file* causes the subsequent input to be logged to the file with the name *name-file*, i.e., everything you type on your terminal will also appear in this file. Note that InputLogTo and LogTo (9.7.4) cannot be used at the same time while InputLogTo and OutputLogTo (9.7.6) can. Note that InputLogTo will overwrite the previous contents of this file if it already existed.

Called without arguments, InputLogTo stops logging to a file or stream.

9.7.6 OutputLogTo

▷ OutputLogTo(*name-file*) (operation)
▷ OutputLogTo() (operation)

Calling OutputLogTo with a string *name-file* causes the subsequent output to be logged to the file with the name *name-file*, i.e., everything GAP prints on your terminal will also appear in this file. Note that OutputLogTo and LogTo (9.7.4) cannot be used at the same time while InputLogTo (9.7.5) and OutputLogTo can. Note that OutputLogTo will overwrite the previous contents of this file if it already existed.

Called without arguments, OutputLogTo stops logging to a file or stream.

9.7.7 CrcFile

▷ CrcFile(*name-file*) (function)

CRC (cyclic redundancy check) numbers provide a certain method of doing checksums. They are used by GAP to check whether files have changed.

CrcFile computes a checksum value for the file with filename *name-file* and returns this value as an integer. The function returns fail if a system error occurred, say, for example, if *name-file* does not exist. In this case the function LastSystemError (9.1.1) can be used to get information about the error.

Example

```
gap> CrcFile( "lib/morpheus.gi" );
2705743645
```

9.7.8 RemoveFile

▷ RemoveFile(*name-file*) (function)

will remove the file with filename *name-file* and returns true in case of success. The function returns fail if a system error occurred, for example, if your permissions do not allow the removal of *name-file*. In this case the function LastSystemError (9.1.1) can be used to get information about the error.

9.7.9 Reread

▷ Reread(*name-file*) (function)
▷ REREADING (global variable)

In general, it is not possible to read the same GAP library file twice, or to read a compiled version after reading a GAP version, because crucial global variables are made read-only (see 4.9) and filters and methods are added to global tables.

A partial solution to this problem is provided by the function Reread (and related functions RereadLib etc.). Reread(*name-file*) sets the global variable REREADING to true, reads the file named by *name-file* and then resets REREADING. Various system functions behave differently when REREADING is set to true. In particular, assignment to read-only global variables is permitted, calls to NewRepresentation (79.2.1) and NewInfoClass (7.4.1) with parameters identical to those of an existing representation or info class will return the existing object, and methods installed with InstallMethod (78.2.1) may sometimes displace existing methods.

This function may not entirely produce the intended results, especially if what has changed is the super-representation of a representation or the requirements of a method. In these cases, it is necessary to restart GAP to read the modified file.

An additional use of Reread is to load the compiled version of a file for which the GAP language version had previously been read (or perhaps was included in a saved workspace). See 76.3.9 and 3.3 for more information.

It is not advisable to use Reread programmatically. For example, if a file that contains calls to Reread is read with Reread then REREADING may be reset too early.

Chapter 10

Streams

Streams provide flexible access to GAP's input and output processing. An *input stream* takes characters from some source and delivers them to GAP which *reads* them from the stream. When an input stream has delivered all characters it is at `end-of-stream`. An *output stream* receives characters from GAP which *writes* them to the stream, and delivers them to some destination.

A major use of streams is to provide efficient and flexible access to files. Files can be read and written using `Read` (9.7.1) and `AppendTo` (9.7.3), however the former only allows a complete file to be read as GAP input and the latter imposes a high time penalty if many small pieces of output are written to a large file. Streams allow input files in other formats to be read and processed, and files to be built up efficiently from small pieces of output. Streams may also be used for other purposes, for example to read from and print to GAP strings, or to read input directly from the user.

Any stream is either a *text stream*, which translates the `end-of-line` character (`\n`) to or from the system's representation of `end-of-line` (e.g., *new-line* under UNIX and *carriage-return-new-line* under DOS), or a *binary stream*, which does not translate the `end-of-line` character. The processing of other unprintable characters by text streams is undefined. Binary streams pass them unchanged.

Whereas it is cheap to append to a stream, streams do consume system resources, and only a limited number can be open at any time, therefore it is necessary to close a stream as soon as possible using `CloseStream` (10.2.1). If creating a stream failed then `LastSystemError` (9.1.1) can be used to get information about the failure.

10.1 Categories for Streams and the StreamsFamily

10.1.1 IsStream

▷ `IsStream(`*obj*`)` (Category)

Streams are GAP objects and all open streams, input, output, text and binary, lie in this category.

10.1.2 IsClosedStream

▷ `IsClosedStream(`*obj*`)` (Category)

When a stream is closed, its type changes to lie in `IsClosedStream`. This category is used to install methods that trap accesses to closed streams.

10.1.3 IsInputStream

▷ `IsInputStream(`*obj*`)` (Category)

All input streams lie in this category, and support input operations such as `ReadByte` (10.3.4) (see 10.3)

10.1.4 IsInputTextStream

▷ `IsInputTextStream(`*obj*`)` (Category)

All *text* input streams lie in this category. They translate new-line characters read.

10.1.5 IsInputTextNone

▷ `IsInputTextNone(`*obj*`)` (Category)

It is convenient to use a category to distinguish dummy streams (see 10.9) from others. Other distinctions are usually made using representations

10.1.6 IsOutputStream

▷ `IsOutputStream(`*obj*`)` (Category)

All output streams lie in this category and support basic operations such as `WriteByte` (10.4.1) (see Section 10.4).

10.1.7 IsOutputTextStream

▷ `IsOutputTextStream(`*obj*`)` (Category)

All *text* output streams lie in this category and translate new-line characters on output.

10.1.8 IsOutputTextNone

▷ `IsOutputTextNone(`*obj*`)` (Category)

It is convenient to use a category to distinguish dummy streams (see 10.9) from others. Other distinctions are usually made using representations

10.1.9 StreamsFamily

▷ `StreamsFamily` (family)

All streams lie in the `StreamsFamily`.

10.2 Operations applicable to All Streams

10.2.1 CloseStream

▷ `CloseStream(`*stream*`)` (operation)

In order to preserve system resources and to flush output streams every stream should be closed as soon as it is no longer used using `CloseStream`.

It is an error to try to read characters from or write characters to a closed stream. Closing a stream tells the GAP kernel and/or the operating system kernel that the file is no longer needed. This may be necessary because the GAP kernel and/or the operating system may impose a limit on how many streams may be open simultaneously.

10.2.2 FileDescriptorOfStream

▷ `FileDescriptorOfStream(`*stream*`)` (operation)

returns the UNIX file descriptor of the underlying file. This is mainly useful for the `UNIXSelect` (10.2.3) function call. This is as of now only available on UNIX-like operating systems and only for streams to local processes and local files.

10.2.3 UNIXSelect

▷ `UNIXSelect(`*inlist, outlist, exclist, timeoutsec, timeoutusec*`)` (function)

makes the UNIX C-library function `select` accessible from GAP for streams. The functionality is as described in the man page (see UNIX file descriptors (integers) for streams. They can be obtained via `FileDescriptorOfStream` (10.2.2) for streams to local processes and to local files. The argument *timeoutsec* is a timeout in seconds as in the `struct timeval` on the C level. The argument *timeoutusec* is analogously in microseconds. The total timeout is the sum of both. If one of those timeout arguments is not a small integer then no timeout is applicable (`fail` is allowed for the timeout arguments).

The return value is the number of streams that are ready, this may be 0 if a timeout was specified. All file descriptors in the three lists that are not yet ready are replaced by `fail` in this function. So the lists are changed!

This function is not available on the Macintosh architecture and is only available if your operating system has `select`, which is detected during compilation of GAP.

10.3 Operations for Input Streams

Three operations normally used to read files: `Read` (9.7.1), `ReadAsFunction` (9.7.2) and `ReadTest` (7.9.1) can also be used to read GAP input from a stream. The input is immediately parsed and executed. When reading from a stream *str*, the GAP kernel generates calls to `ReadLine(`*str*`)` to supply text to the parser.

Three further operations: `ReadByte` (10.3.4), `ReadLine` (10.3.5) and `ReadAll` (10.3.6), support reading characters from an input stream without parsing them. This can be used to read data in any format and process it in GAP.

Additional operations for input streams support detection of end of stream, and (for those streams for which it is appropriate) random access to the data.

10.3.1 Read (for streams)

▷ `Read(`*input-text-stream*`)` (operation)

reads the input-text-stream as input until `end-of-stream` occurs. See 9.7 for details.

10.3.2 ReadAsFunction (for streams)

▷ `ReadAsFunction(`*input-text-stream*`)` (operation)

reads the input-text-stream as function and returns this function. See 9.7 for details.

Example

```
gap> # a function with local ‘a’ does not change the global one
gap> a := 1;;
gap> i := InputTextString( "local a; a := 10; return a*10;" );;
gap> ReadAsFunction(i)();
100
gap> a;
1
gap> # reading it via ‘Read’ does
gap> i := InputTextString( "a := 10;" );;
gap> Read(i);
gap> a;
10
```

10.3.3 ReadTest (for streams)

▷ `ReadTest(`*input-text-stream*`)` (operation)

reads the input-text-stream as test input until `end-of-stream` occurs. See 9.7 for details.

10.3.4 ReadByte

▷ `ReadByte(`*input-stream*`)` (operation)

`ReadByte` returns one character (returned as integer) from the input stream *input-stream*. `ReadByte` returns `fail` if there is no character available, in particular if it is at the end of a file.

If *input-stream* is the input stream of a input/output process, `ReadByte` may also return `fail` if no byte is currently available.

`ReadByte` is the basic operation for input streams. If a `ReadByte` method is installed for a user-defined type of stream which does not block, then all the other input stream operations will work (although possibly not at peak efficiency).

`ReadByte` will wait (block) until a byte is available. For instance if the stream is a connection to another process, it will wait for the process to output a byte.

10.3.5 ReadLine

▷ ReadLine(*input-stream*) (operation)

ReadLine returns one line (returned as string *with* the newline) from the input stream *input-stream*. ReadLine reads in the input until a newline is read or the end-of-stream is encountered.

If *input-stream* is the input stream of a input/output process, ReadLine may also return fail or return an incomplete line if the other process has not yet written any more. It will always wait (block) for at least one byte to be available, but will then return as much input as is available, up to a limit of one line

A default method is supplied for ReadLine which simply calls ReadByte (10.3.4) repeatedly. This is only safe for streams that cannot block. The kernel uses calls to ReadLine to supply input to the parser when reading from a stream.

10.3.6 ReadAll

▷ ReadAll(*input-stream[, limit]*) (operation)

ReadAll returns all characters as string from the input stream *stream-in*. It waits (blocks) until at least one character is available from the stream, or until there is evidence that no characters will ever be available again. This last indicates that the stream is at end-of-stream. Otherwise, it reads as much input as it can from the stream without blocking further and returns it to the user. If the stream is already at end of file, so that no bytes are available, fail is returned. In the case of a file stream connected to a normal file (not a pseudo-tty or named pipe or similar), all the bytes should be immediately available and this function will read the remainder of the file.

With a second argument, at most *limit* bytes will be returned. Depending on the stream a bounded number of additional bytes may have been read into an internal buffer.

A default method is supplied for ReadAll which simply calls ReadLine (10.3.5) repeatedly. This is only really safe for streams which cannot block. Other streams should install a method for ReadAll

Example

```
gap> i := InputTextString( "1Hallo\nYou\n1" );;
gap> ReadByte(i);
49
gap> CHAR_INT(last);
'1'
gap> ReadLine(i);
"Hallo\n"
gap> ReadLine(i);
"You\n"
gap> ReadLine(i);
"1"
gap> ReadLine(i);
fail
gap> ReadAll(i);
""
gap> RewindStream(i);;
gap> ReadAll(i);
"1Hallo\nYou\n1"
```

10.3.7 IsEndOfStream

▷ `IsEndOfStream(`*input-stream*`)` (operation)

`IsEndOfStream` returns `true` if the input stream is at *end-of-stream*, and `false` otherwise. Note that `IsEndOfStream` might return `false` even if the next `ReadByte` (10.3.4) fails.

10.3.8 PositionStream

▷ `PositionStream(`*input-stream*`)` (operation)

Some input streams, such as string streams and file streams attached to disk files, support a form of random access by way of the operations `PositionStream`, `SeekPositionStream` (10.3.10) and `RewindStream` (10.3.9). `PositionStream` returns a non-negative integer denoting the current position in the stream (usually the number of characters *before* the next one to be read.

If this is not possible, for example for an input stream attached to standard input (normally the keyboard), then `fail` is returned

10.3.9 RewindStream

▷ `RewindStream(`*input-stream*`)` (operation)

`RewindStream` attempts to return an input stream to its starting condition, so that all the same characters can be read again. It returns `true` if the rewind succeeds and `fail` otherwise

A default method implements RewindStream using `SeekPositionStream` (10.3.10).

10.3.10 SeekPositionStream

▷ `SeekPositionStream(`*input-stream, pos*`)` (operation)

`SeekPositionStream` attempts to rewind or wind forward an input stream to the specified position. This is not possible for all streams. It returns `true` if the seek is successful and `fail` otherwise.

10.4 Operations for Output Streams

10.4.1 WriteByte

▷ `WriteByte(`*output-stream, byte*`)` (operation)

writes the next character (given as *integer*) to the output stream *output-stream*. The function returns `true` if the write succeeds and `fail` otherwise.

`WriteByte` is the basic operation for output streams. If a `WriteByte` method is installed for a user-defined type of stream, then all the other output stream operations will work (although possibly not at peak efficiency).

10.4.2 WriteLine

▷ WriteLine(*output-stream, string*) (operation)

appends *string* to *output-stream*. A final newline is written. The function returns `true` if the write succeeds and `fail` otherwise.

A default method is installed which implements `WriteLine` by repeated calls to `WriteByte` (10.4.1).

10.4.3 WriteAll

▷ WriteAll(*output-stream, string*) (operation)

appends *string* to *output-stream*. No final newline is written. The function returns `true` if the write succeeds and `fail` otherwise. It will block as long as necessary for the write operation to complete (for example for a child process to clear its input buffer)

A default method is installed which implements `WriteAll` by repeated calls to `WriteByte` (10.4.1).

When printing or appending to a stream (using `PrintTo` (9.7.3), or `AppendTo` (9.7.3) or when logging to a stream), the kernel generates a call to `WriteAll` for each line output.

Example

```
gap> str := "";; a := OutputTextString(str,true);;
gap> WriteByte(a,INT_CHAR('H'));
true
gap> WriteLine(a,"allo");
true
gap> WriteAll(a,"You\n");
true
gap> CloseStream(a);
gap> Print(str);
Hallo
You
```

10.4.4 PrintTo and AppendTo (for streams)

▷ PrintTo(*output-stream, arg1, ...*) (function)

▷ AppendTo(*output-stream, arg1, ...*) (function)

These functions work like `Print` (6.3.4), except that the output is appended to the output stream *output-stream*.

Example

```
gap> str := "";; a := OutputTextString(str,true);;
gap> AppendTo( a, (1,2,3), ":", Z(3) );
gap> CloseStream(a);
gap> Print( str, "\n" );
(1,2,3):Z(3)
```

10.4.5 LogTo (for streams)

▷ LogTo(*stream*) (operation)

causes the subsequent interaction to be logged to the output stream *stream*. It works in precisely the same way as it does for files (see LogTo (9.7.4)).

10.4.6 InputLogTo (for streams)

▷ InputLogTo(*stream*) (operation)

causes the subsequent input to be logged to the output stream *stream*. It works just like it does for files (see InputLogTo (9.7.5)).

10.4.7 OutputLogTo (for streams)

▷ OutputLogTo(*stream*) (operation)

causes the subsequent output to be logged to the output stream *stream*. It works just like it does for files (see OutputLogTo (9.7.6)).

10.4.8 SetPrintFormattingStatus

▷ SetPrintFormattingStatus(*stream, newstatus*) (operation)
▷ PrintFormattingStatus(*stream*) (operation)

When text is being sent to an output text stream via PrintTo (9.7.3), AppendTo (9.7.3), LogTo (10.4.5), etc., it is by default formatted just as it would be were it being printed to the screen. Thus, it is broken into lines of reasonable length at (where possible) sensible places, lines containing elements of lists or records are indented, and so forth. This is appropriate if the output is eventually to be viewed by a human, and harmless if it to passed as input to GAP, but may be unhelpful if the output is to be passed as input to another program. It is possible to turn off this behaviour for a stream using the SetPrintFormattingStatus operation, and to test whether it is on or off using PrintFormattingStatus.

SetPrintFormattingStatus sets whether output sent to the output stream *stream* via PrintTo (9.7.3), AppendTo (9.7.3), etc. will be formatted with line breaks and indentation. If the second argument *newstatus* is true then output will be so formatted, and if false then it will not. If the stream is not a text stream, only false is allowed.

PrintFormattingStatus returns true if output sent to the output text stream *stream* via PrintTo (9.7.3), AppendTo (9.7.3), etc. will be formatted with line breaks and indentation, and false otherwise. For non-text streams, it returns false. If as argument *stream* the string "*stdout*" is given, these functions refer to the formatting status of the standard output (so usually the users terminal screen).

These functions do not influence the behaviour of the low level functions WriteByte (10.4.1), WriteLine (10.4.2) or WriteAll (10.4.3) which always write without formatting.

Example

```
gap> s := "";; str := OutputTextString(s,false);;
gap> PrintTo(str,Primes{[1..30]});
```

```
gap> s;
"[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,\
 \n  67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]"
gap> Print(s,"\n");
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
  67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
gap> SetPrintFormattingStatus(str, false);
gap> PrintTo(str,Primes{[1..30]});
gap> s;
"[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,\
 \n  67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ][ 2, 3, 5, 7\
, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, \
79, 83, 89, 97, 101, 103, 107, 109, 113 ]"
gap> Print(s,"\n");
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
  67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ][ 2, 3, 5, 7, 1\
1, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,\
 83, 89, 97, 101, 103, 107, 109, 113 ]
```

10.5 File Streams

File streams are streams associated with files. An input file stream reads the characters it delivers from a file, an output file stream prints the characters it receives to a file. The following functions can be used to create such streams. They return `fail` if an error occurred, in this case `LastSystemError` (9.1.1) can be used to get information about the error.

10.5.1 InputTextFile

▷ `InputTextFile(`*`name-file`*`)` (operation)

`InputTextFile(` *`name-file`* `)` returns an input stream in the category `IsInputTextStream` (10.1.4) that delivers the characters from the file *`name-file`*.

10.5.2 OutputTextFile

▷ `OutputTextFile(`*`name-file, append`*`)` (operation)

`OutputTextFile(` *`name-file`*`,` *`append`* `)` returns an output stream in the category `IsOutputTextFile` that writes received characters to the file *`name-file`*. If *`append`* is `false`, then the file is emptied first, otherwise received characters are added at the end of the file.

Example

```
gap> # use a temporary directory
gap> name := Filename( DirectoryTemporary(), "test" );;
gap> # create an output stream, append output, and close again
gap> output := OutputTextFile( name, true );;
gap> AppendTo( output, "Hallo\n", "You\n" );
gap> CloseStream(output);
gap> # create an input, print complete contents of file, and close
gap> input := InputTextFile(name);;
```

```
gap> Print( ReadAll(input) );
Hallo
You
gap> CloseStream(input);
gap> # append a single line
gap> output := OutputTextFile( name, true );;
gap> AppendTo( output, "AppendLine\n" );
gap> # close output stream to flush the output
gap> CloseStream(output);
gap> # create an input, print complete contents of file, and close
gap> input := InputTextFile(name);;
gap> Print( ReadAll(input) );
Hallo
You
AppendLine
gap> CloseStream(input);
```

10.6 User Streams

The commands described in this section create streams which accept characters from, or deliver characters to, the user, via the keyboard or the GAP session display.

10.6.1 InputTextUser

▷ `InputTextUser()` (function)

returns an input text stream which delivers characters typed by the user (or from the standard input device if it has been redirected). In normal circumstances, characters are delivered one by one as they are typed, without waiting until the end of a line. No prompts are printed.

10.6.2 OutputTextUser

▷ `OutputTextUser()` (function)

returns an output stream which delivers characters to the user's display (or the standard output device if it has been redirected). Each character is delivered immediately it is written, without waiting for a full line of output. Text written in this way is *not* written to the session log (see `LogTo` (9.7.4)).

10.6.3 InputFromUser

▷ `InputFromUser(`*arg*`)` (function)

prints the *arg* as a prompt, then waits until a text is typed by the user (or from the standard input device if it has been redirected). This text must be a *single* expression, followed by one *enter*. This is evaluated (see `EvalString` (27.8.3)) and the result is returned.

10.7 String Streams

String streams are streams associated with strings. An input string stream reads the characters it delivers from a string, an output string stream appends the characters it receives to a string. The following functions can be used to create such streams.

10.7.1 InputTextString

▷ InputTextString(*string*) (operation)

InputTextString(*string*) returns an input stream that delivers the characters from the string *string*. The *string* is not changed when reading characters from it and changing the *string* after the call to InputTextString has no influence on the input stream.

10.7.2 OutputTextString

▷ OutputTextString(*list, append*) (operation)

returns an output stream that puts all received characters into the list *list*. If *append* is false, then the list is emptied first, otherwise received characters are added at the end of the list.

Example

```
gap> # read input from a string
gap> input := InputTextString( "Hallo\nYou\n" );;
gap> ReadLine(input);
"Hallo\n"
gap> ReadLine(input);
"You\n"
gap> # print to a string
gap> str := "";;
gap> out := OutputTextString( str, true );;
gap> PrintTo( out, 1, "\n", (1,2,3,4)(5,6), "\n" );
gap> CloseStream(out);
gap> Print( str );
1
(1,2,3,4)(5,6)
```

10.8 Input-Output Streams

Input-output streams capture bidirectional communications between GAP and another process, either locally or (@as yet unimplemented@) remotely.

Such streams support the basic operations of both input and output streams. They should provide some buffering, allowing output data to be written to the stream, even when input data is waiting to be read, but the amount of this buffering is operating system dependent, and the user should take care not to get too far ahead in writing, or behind in reading, or deadlock may occur.

At present the only type of Input-Output streams that are implemented provide communication with a local child process, using a pseudo-tty.

Like other streams, write operations are blocking, read operations will block to get the first character, but not thereafter.

As far as possible, no translation is done on characters written to, or read from the stream, and no control characters have special effects, but the details of particular pseudo-tty implementations may effect this.

10.8.1 IsInputOutputStream

▷ `IsInputOutputStream(`*obj*`)` (Category)

`IsInputOutputStream` is the Category of Input-Output Streams; it returns `true` if the *obj* is an input-output stream and `false` otherwise.

10.8.2 InputOutputLocalProcess

▷ `InputOutputLocalProcess(`*dir, executable, args*`)` (function)

starts up a slave process, whose executable file is *executable*, with "command line" arguments *args* in the directory *dir*. (Suitable choices for *dir* are `DirectoryCurrent()` or `DirectoryTemporary()` (see Section 9.3); `DirectoryTemporary()` may be a good choice when *executable* generates output files that it doesn't itself remove afterwards.) `InputOutputLocalProcess` returns an InputOutputStream object. Bytes written to this stream are received by the slave process as if typed at a terminal on standard input. Bytes written to standard output by the slave process can be read from the stream.

When the stream is closed, the signal SIGTERM is delivered to the child process, which is expected to exit.

Example

```
gap> d := DirectoryCurrent();
dir("./")
gap> f := Filename(DirectoriesSystemPrograms(), "rev");
"/usr/bin/rev"
gap> s := InputOutputLocalProcess(d,f,[]);
< input/output stream to rev >
gap> WriteLine(s,"The cat sat on the mat");
true
gap> Print(ReadLine(s));
tam eht no tas tac ehT
gap> x := ListWithIdenticalEntries(10000,'x');;
gap> ConvertToStringRep(x);
gap> WriteLine(s,x);
true
gap> WriteByte(s,INT_CHAR('\n'));
true
gap> y := ReadAll(s);;
gap> Length(y);
4095
gap> CloseStream(s);
gap> s;
< closed input/output stream to rev >
```

10.8.3 ReadAllLine

▷ ReadAllLine(*iostream[, nofail][, IsAllLine]*) (operation)

For an input/output stream *iostream* ReadAllLine reads until a newline character if any input is found or returns fail if no input is found, i.e. if any input is found ReadAllLine is non-blocking.

If the argument *nofail* (which must be false or true) is provided and it is set to true then ReadAllLine will wait, if necessary, for input and never return fail.

If the argument *IsAllLine* (which must be a function that takes a string argument and returns either true or false) then it is used to determine what constitutes a whole line. The default behaviour is equivalent to passing the function

Example

```
line -> 0 < Length(line) and line[Length(line)] = '\n'
```

for the *IsAllLine* argument. The purpose of the *IsAllLine* argument is to cater for the case where the input being read is from an external process that writes a "prompt" for data that does not terminate with a newline.

If the first argument is an input stream but not an input/output stream then ReadAllLine behaves as if ReadLine (10.3.5) was called with just the first argument and any additional arguments are ignored.

10.9 Dummy Streams

The following two commands create dummy streams which will consume all characters and never deliver one.

10.9.1 InputTextNone

▷ InputTextNone() (function)

returns a dummy input text stream, which delivers no characters, i.e., it is always at end of stream. Its main use is for calls to Process (11.1.1) when the started program does not read anything.

10.9.2 OutputTextNone

▷ OutputTextNone() (function)

returns a dummy output stream, which discards all received characters. Its main use is for calls to Process (11.1.1) when the started program does not write anything.

10.10 Handling of Streams in the Background

This section describes a feature of the GAP kernel that can be used to handle pending streams somehow "in the background". This is currently not available on the Macintosh architecture and only on operating systems that have select.

Right before GAP reads a keypress from the keyboard it calls a little subroutine that can handle streams that are ready to be read or ready to be written. This means that GAP can handle these streams

during user input on the command line. Note that this does not work when GAP is in the middle of some calculation.

This feature is used in the following way. One can install handler functions for reading or writing streams via `InstallCharReadHookFunc` (10.10.1). Handlers can be removed via `UnInstallCharReadHookFunc` (10.10.2)

Note that handler functions must not return anything and get one integer argument, which refers to an index in one of the following arrays (according to whether the function was installed for input, output or exceptions on the stream). Handler functions usually should not output anything on the standard output because this ruins the command line during command line editing.

10.10.1 InstallCharReadHookFunc

▷ `InstallCharReadHookFunc(`*`stream, mode, func`*`)` (function)

installs the function *`func`* as a handler function for the stream *`stream`*. The argument *`mode`* decides, for what operations on the stream this function is installed. *`mode`* must be a string, in which a letter `r` means "read", `w` means "write" and `x` means "exception", according to the `select` function call in the UNIX C-library (see `man select` and `UNIXSelect` (10.2.3)). More than one letter is allowed in *`mode`*. As described above the function is called in a situation when GAP is reading a character from the keyboard. Handler functions should not use much time to complete.

This functionality does not work on the Macintosh architecture and only works if the operating system has a `select` function.

10.10.2 UnInstallCharReadHookFunc

▷ `UnInstallCharReadHookFunc(`*`stream, func`*`)` (function)

uninstalls the function *`func`* as a handler function for the stream *`stream`*. All instances are deinstalled, regardless of the mode of operation (read, write, exception).

This functionality does not work on the Macintosh architecture and only works if the operating system has a `select` function.

10.11 Comma separated files

In some situations it can be desirable to process data given in the form of a spreadsheet (such as Excel). GAP can do this using the CSV (comma separated values) format, which spreadsheet programs can usually read in or write out.

The first line of the spreadsheet is used as labels of record components, each subsequent line then corresponds to a record. Entries enclosed in double quotes are considered as strings and are permitted to contain the separation character (usually a comma).

10.11.1 ReadCSV

▷ `ReadCSV(`*`filename[, nohead][, separator]`*`)` (function)

This function reads in a spreadsheet, saved in CSV format (*c*omma *s*eparated *v*alues) and returns its entries as a list of records. The entries of the first line of the spreadsheet are used to denote the

names of the record components. Blanks will be translated into underscore characters. If the parameter *nohead* is given as `true`, instead the record components will be called `fieldn`. Each subsequent line will create one record. If given, *separator* is the character used to separate fields. Otherwise it defaults to a comma.

10.11.2 PrintCSV

▷ `PrintCSV(`*filename, list[, fields]*`)` (function)

This function prints a list of records as a spreadsheet in CSV format (which can be read in for example into Excel). The names of the record components will be printed as entries in the first line. If the argument *fields* is given only the record fields listed in this list will be printed and they will be printed in the same arrangement as given in this list. If the option noheader is set to true the line with the record field names will not be printed.

Chapter 11

Processes

GAP can call other programs, such programs are called *processes*. There are two kinds of processes: first there are processes that are started, run and return a result, while GAP is suspended until the process terminates. Then there are processes that will run in parallel to GAP as subprocesses and GAP can communicate and control the processes using streams (see `InputOutputLocalProcess` (10.8.2)).

11.1 Process and Exec

11.1.1 Process

▷ `Process(`*dir, prg, stream-in, stream-out, options*`)` (operation)

`Process` runs a new process and returns when the process terminates. It returns the return value of the process if the operating system supports such a concept.

The first argument *dir* is a directory object (see 9.3) which will be the current directory (in the usual UNIX or MSDOS sense) when the program is run. This will only matter if the program accesses files (including running other programs) via relative path names. In particular, it has nothing to do with finding the binary to run.

In general the directory will either be the current directory, which is returned by `DirectoryCurrent` (9.3.4) –this was the behaviour of GAP 3– or a temporary directory returned by `DirectoryTemporary` (9.3.3). If one expects that the process creates temporary or log files the latter should be used because GAP will attempt to remove these directories together with all the files in them when quitting.

If a program of a GAP package which does not only consist of GAP code needs to be launched in a directory relative to certain data libraries, then the first entry of `DirectoriesPackageLibrary` (76.3.4) should be used. The argument of `DirectoriesPackageLibrary` (76.3.4) should be the path to the data library relative to the package directory.

If a program calls other programs and needs to be launched in a directory containing the executables for such a GAP package then the first entry of `DirectoriesPackagePrograms` (76.3.5) should be used.

The latter two alternatives should only be used if absolutely necessary because otherwise one risks accumulating log or core files in the package directory.

Example

```
gap> path := DirectoriesSystemPrograms();;
```

```
gap> ls := Filename( path, "ls" );;
gap> stdin := InputTextUser();;
gap> stdout := OutputTextUser();;
gap> Process( path[1], ls, stdin, stdout, ["-c"] );;
awk    ls     mkdir
gap> # current directory, here the root directory
gap> Process( DirectoryCurrent(), ls, stdin, stdout, ["-c"] );;
bin    lib    trans  tst    CVS    grp    prim   thr    two
src    dev    etc    tbl    doc    pkg    small  tom
gap> # create a temporary directory
gap> tmpdir := DirectoryTemporary();;
gap> Process( tmpdir, ls, stdin, stdout, ["-c"] );;
gap> PrintTo( Filename( tmpdir, "emil" ) );
gap> Process( tmpdir, ls, stdin, stdout, ["-c"] );;
emil
```

prg is the filename of the program to launch, for portability it should be the result of `Filename` (9.4.1) and should pass `IsExecutableFile` (9.6.4). Note that `Process` does *no* searching through a list of directories, this is done by `Filename` (9.4.1).

stream-in is the input stream that delivers the characters to the process. For portability it should either be `InputTextNone` (10.9.1) (if the process reads no characters), `InputTextUser` (10.6.1), the result of a call to `InputTextFile` (10.5.1) from which no characters have been read, or the result of a call to `InputTextString` (10.7.1).

`Process` is free to consume *all* the input even if the program itself does not require any input at all.

stream-out is the output stream which receives the characters from the process. For portability it should either be `OutputTextNone` (10.9.2) (if the process writes no characters), `OutputTextUser` (10.6.2), the result of a call to `OutputTextFile` (10.5.2) to which no characters have been written, or the result of a call to `OutputTextString` (10.7.2).

options is a list of strings which are passed to the process as command line argument. Note that no substitutions are performed on the strings, i.e., they are passed immediately to the process and are not processed by a command interpreter (shell). Further note that each string is passed as one argument, even if it contains *space* characters. Note that input/output redirection commands are *not* allowed as *options*.

In order to find a system program use `DirectoriesSystemPrograms` (9.3.6) together with `Filename` (9.4.1).

Example

```
gap> path := DirectoriesSystemPrograms();;
gap> date := Filename( path, "date" );
"/bin/date"
```

The next example shows how to execute `date` with no argument and no input, and collect the output into a string stream.

Example

```
gap> str := "";; a := OutputTextString(str,true);;
gap> Process( DirectoryCurrent(), date, InputTextNone(), a, [] );
0
gap> CloseStream(a);
gap> Print(str);
Fri Jul 11 09:04:23 MET DST 1997
```

11.1.2 Exec

▷ Exec(*cmd, option1, ..., optionN*) (function)

Exec runs a shell in the current directory to execute the command given by the string *cmd* with options *option1*, ..., *optionN*.

Example

```
gap> Exec( "date" );
Thu Jul 24 10:04:13 BST 1997
```

cmd is interpreted by the shell and therefore we can make use of the various features that a shell offers as in following example.

Example

```
gap> Exec( "echo \"GAP is great!\" > foo" );
gap> Exec( "cat foo" );
GAP is great!
gap> Exec( "rm foo" );
```

Exec calls the more general operation Process (11.1.1). The function Edit (6.10.1) should be used to call an editor from within GAP.

Chapter 12

Objects and Elements

An *object* is anything in GAP that can be assigned to a variable, so nearly everything in GAP is an object.

Different objects can be regarded as equal with respect to the equivalence relation "=", in this case we say that the objects describe the same *element*.

12.1 Objects

Nearly all things one deals with in GAP are *objects*. For example, an integer is an object, as is a list of integers, a matrix, a permutation, a function, a list of functions, a record, a group, a coset or a conjugacy class in a group.

Examples of things that are not objects are comments which are only lexical constructs, `while` loops which are only syntactical constructs, and expressions, such as `1 + 1`; but note that the value of an expression, in this case the integer 2, is an object.

Objects can be assigned to variables, and everything that can be assigned to a variable is an object. Analogously, objects can be used as arguments of functions, and can be returned by functions.

12.1.1 IsObject

▷ `IsObject(`*obj*`)` (Category)

`IsObject` returns `true` if the object *obj* is an object. Obviously it can never return `false`.

It can be used as a filter in `InstallMethod` (78.2.1) when one of the arguments can be anything.

12.2 Elements as equivalence classes

The equality operation "=" defines an equivalence relation on all GAP objects. The equivalence classes are called *elements*.

There are basically three reasons to regard different objects as equal. Firstly the same information may be stored in different places. Secondly the same information may be stored in different ways; for example, a polynomial can be stored sparsely or densely. Thirdly different information may be equal modulo a mathematical equivalence relation. For example, in a finitely presented group with the relation $a^2 = 1$ the different objects a and a^3 describe the same element.

As an example of all three reasons, consider the possibility of storing an integer in several places of the memory, of representing it as a fraction with denominator 1, or of representing it as a fraction with any denominator, and numerator a suitable multiple of the denominator.

12.3 Sets

In GAP there is no category whose definition corresponds to the mathematical property of being a set, however in the manual we will often refer to an object as a *set* in order to convey the fact that mathematically, we are thinking of it as a set. In particular, two sets A and B are equal if and only if, $x \in A \iff x \in B$.

There are two types of object in GAP which exhibit this kind of behaviour with respect to equality, namely domains (see Section 12.4) and lists whose elements are strictly sorted see `IsSSortedList` (21.17.4). In general, *set* in this manual will mean an object of one of these types.

More precisely: two domains can be compared with "`{=}`", the answer being `true` if and only if the sets of elements are equal (regardless of any additional structure) and; a domain and a list can be compared with "=", the answer being `true` if and only if the list is equal to the strictly sorted list of elements of the domain.

A discussion about sorted lists and sets can be found in Section 21.19.

12.4 Domains

An especially important class of objects in GAP are those whose underlying mathematical abstraction is that of a structured set, for example a group, a conjugacy class, or a vector space. Such objects are called *domains*. The equality relation between domains is always equality *as sets*, so that two domains are equal if and only if they contain the same elements.

Domains play a central role in GAP. In a sense, the only reason that GAP supports objects such as integers and permutations is the wish to form domains of them and compute the properties of those domains.

Domains are described in Chapter 31.

12.5 Identical Objects

Two objects that are equal *as objects* (that is they actually refer to the same area of computer memory) and not only w.r.t. the equality relation "=" are called *identical*. Identical objects do of course describe the same element.

12.5.1 IsIdenticalObj

▷ `IsIdenticalObj(`*`obj1, obj2`*`)` (function)

`IsIdenticalObj` tests whether the objects *`obj1`* and *`obj2`* are identical (that is they are either equal immediate objects or are both stored at the same location in memory.

If two copies of a simple constant object (see section 12.6) are created, it is not defined whether GAP will actually store two equal but non-identical objects, or just a single object. For mutable objects, however, it is important to know whether two values refer to identical or non-identical objects,

and the documentation of operations that return mutable values should make clear whether the values returned are new, or may be identical to values stored elsewhere.

Example

```
gap> IsIdenticalObj( 10^6, 10^6);
true
gap> IsIdenticalObj( 10^30, 10^30);
false
gap> IsIdenticalObj( true, true);
true
```

Generally, one may compute with objects but think of the results in terms of the underlying elements because one is not interested in locations in memory, data formats or information beyond underlying equivalence relations. But there are cases where it is important to distinguish the relations identity and equality. This is best illustrated with an example. (The reader who is not familiar with lists in GAP, in particular element access and assignment, is referred to Chapter 21.)

Example

```
gap> l1:= [ 1, 2, 3 ];; l2:= [ 1, 2, 3 ];;
gap> l1 = l2;
true
gap> IsIdenticalObj( l1, l2 );
false
gap> l1[3]:= 4;; l1; l2;
[ 1, 2, 4 ]
[ 1, 2, 3 ]
gap> l1 = l2;
false
```

The two lists `l1` and `l2` are equal but not identical. Thus a change in `l1` does not affect `l2`.

Example

```
gap> l1:= [ 1, 2, 3 ];; l2:= l1;;
gap> l1 = l2;
true
gap> IsIdenticalObj( l1, l2 );
true
gap> l1[3]:= 4;; l1; l2;
[ 1, 2, 4 ]
[ 1, 2, 4 ]
gap> l1 = l2;
true
```

Here, `l1` and `l2` are identical objects, so changing `l1` means a change to `l2` as well.

12.5.2 IsNotIdenticalObj

▷ `IsNotIdenticalObj(`*obj1, obj2*`)` (function)

tests whether the objects *obj1* and *obj2* are not identical.

12.6 Mutability and Copyability

An object in GAP is said to be *immutable* if its mathematical value (as defined by $=$) does not change under any operation. More explicitly, suppose a is immutable and O is some operation on a, then if $a = b$ evaluates to `true` before executing $O(a)$, $a = b$ also evaluates to `true` afterwards. (Examples for operations O that change mutable objects are `Add` (21.4.2) and `Unbind` (21.5.2) which are used to change list objects, see Chapter 21.) An immutable object *may* change, for example to store new information, or to adopt a more efficient representation, but this does not affect its behaviour under $=$.

There are two points here to note. Firstly, "operation" above refers to the functions and methods which can legitimately be applied to the object, and not the `!.` operation whereby virtually any aspect of any GAP level object may be changed. The second point which follows from this, is that when implementing new types of objects, it is the programmer's responsibility to ensure that the functions and methods they write never change immutable objects mathematically.

In fact, most objects with which one deals in GAP are immutable. For instance, the permutation `(1,2)` will never become a different permutation or a non-permutation (although a variable which previously had `(1,2)` stored in it may subsequently have some other value).

For many purposes, however, *mutable* objects are useful. These objects may be changed to represent different mathematical objects during their life. For example, mutable lists can be changed by assigning values to positions or by unbinding values at certain positions. Similarly, one can assign values to components of a mutable record, or unbind them.

12.6.1 IsCopyable

▷ `IsCopyable(`*`obj`*`)` (Category)

If a mutable form of an object *`obj`* can be made in GAP, the object is called *copyable*. Examples of copyable objects are of course lists and records. A new mutable version of the object can always be obtained by the operation `ShallowCopy` (12.7.1).

Objects for which only an immutable form exists in GAP are called *constants*. Examples of constants are integers, permutations, and domains. Called with a constant as argument, `Immutable` (12.6.3) and `ShallowCopy` (12.7.1) return this argument.

12.6.2 IsMutable

▷ `IsMutable(`*`obj`*`)` (Category)

tests whether *`obj`* is mutable.

If an object is mutable then it is also copyable (see `IsCopyable` (12.6.1)), and a `ShallowCopy` (12.7.1) method should be supplied for it. Note that `IsMutable` must not be implied by another filter, since otherwise `Immutable` (12.6.3) would be able to create paradoxical objects in the sense that `IsMutable` for such an object is `false` but the filter that implies `IsMutable` is `true`.

In many situations, however, one wants to ensure that objects are *immutable*. For example, take the identity of a matrix group. Since this matrix may be referred to as the identity of the group in several places, it would be fatal to modify its entries, or add or unbind rows. We can obtain an immutable copy of an object with `Immutable` (12.6.3).

12.6.3 Immutable

▷ `Immutable(`*obj*`)` (operation)

returns an immutable structural copy (see `StructuralCopy` (12.7.2)) of *obj* in which the subobjects are immutable *copies* of the subobjects of *obj*. If *obj* is immutable then `Immutable` returns *obj* itself.

GAP will complain with an error if one tries to change an immutable object.

12.6.4 MakeImmutable

▷ `MakeImmutable(`*obj*`)` (function)

One can turn the (mutable or immutable) object *obj* into an immutable one with `MakeImmutable`; note that this also makes all subobjects of *obj* immutable, so one should call `MakeImmutable` only if *obj* and its mutable subobjects are newly created. If one is not sure about this, `Immutable` (12.6.3) should be used.

Note that it is *not* possible to turn an immutable object into a mutable one; only mutable copies can be made (see 12.7).

Using `Immutable` (12.6.3), it is possible to store an immutable identity matrix or an immutable list of generators, and to pass around references to this immutable object safely. Only when a mutable copy is really needed does the actual object have to be duplicated. Compared to the situation without immutable objects, much unnecessary copying is avoided this way. Another advantage of immutability is that lists of immutable objects may remember whether they are sorted (see 21.19), which is not possible for lists of mutable objects.

Since the operation `Immutable` (12.6.3) must work for any object in GAP, it follows that an immutable form of every object must be possible, even if it is not sensible, and user-defined objects must allow for the possibility of becoming immutable without notice.

12.6.5 Mutability of Iterators

An interesting example of mutable (and thus copyable) objects is provided by *iterators*, see 30.8. (Of course an immutable form of an iterator is not very useful, but clearly `Immutable` (12.6.3) will yield such an object.) Every call of `NextIterator` (30.8.5) changes a mutable iterator until it is exhausted, and this is the only way to change an iterator. `ShallowCopy` (12.7.1) for an iterator *iter* is defined so as to return a mutable iterator that has no mutable data in common with *iter*, and that behaves equally to *iter* w.r.t. `IsDoneIterator` (30.8.4) and (if *iter* is mutable) `NextIterator` (30.8.5). Note that this meaning of the "shallow copy" of an iterator that is returned by `ShallowCopy` (12.7.1) is not as obvious as for lists and records, and must be explicitly defined.

12.6.6 Mutability of Results of Arithmetic Operations

Many operations return immutable results, among those in particular attributes (see 13.5). Examples of attributes are `Size` (30.4.6), `Zero` (31.10.3), `AdditiveInverse` (31.10.9), `One` (31.10.2), and `Inverse` (31.10.8). Arithmetic operations, such as the binary infix operations `+`, `-`, `*`, `/`, `^`, `mod`, the unary `-`, and operations such as `Comm` (31.12.3) and `LeftQuotient` (31.12.2), return *mutable* results, *except* if all arguments are immutable. So the product of two matrices or of a vector and a matrix is immutable if and only if the two matrices or both the vector and the matrix are immutable (see

also 21.11). There is one exception to this rule, which arises where the result is less deeply nested than at least one of the argument, where mutable arguments may sometimes lead to an immutable result. For instance, a mutable matrix with immutable rows, multiplied by an immutable vector gives an immutable vector result. The exact rules are given in 21.11.

It should be noted that `0 * obj` is equivalent to `ZeroSM( obj )`, `-obj` is equivalent to `AdditiveInverseSM( obj )`, `obj^0` is equivalent to `OneSM( obj)`, and `obj^-1` is equivalent to `InverseSM( obj )`. The "SM" stands for "same mutability", and indicates that the result is mutable if and only if the argument is mutable.

The operations `ZeroOp` (31.10.3), `AdditiveInverseOp` (31.10.9), `OneOp` (31.10.2), and `InverseOp` (31.10.8) return *mutable* results whenever a mutable version of the result exists, contrary to the attributes `Zero` (31.10.3), `AdditiveInverse` (31.10.9), `One` (31.10.2), and `Inverse` (31.10.8).

If one introduces new arithmetic objects then one need not install methods for the attributes `One` (31.10.2), `Zero` (31.10.3), etc. The methods for the associated operations `OneOp` (31.10.2) and `ZeroOp` (31.10.3) will be called, and then the results made immutable.

All methods installed for the arithmetic operations must obey the rule about the mutability of the result. This means that one may try to avoid the perhaps expensive creation of a new object if both operands are immutable, and of course no problems of this kind arise at all in the (usual) case that the objects in question do not admit a mutable form, i.e., that these objects are not copyable.

In a few, relatively low-level algorithms, one wishes to treat a matrix partly as a data structure, and manipulate and change its entries. For this, the matrix needs to be mutable, and the rule that attribute values are immutable is an obstacle. For these situations, a number of additional operations are provided, for example `TransposedMatMutable` (24.5.6) constructs a mutable matrix (contrary to the attribute `TransposedMat` (24.5.6)), while `TriangulizeMat` (24.7.3) modifies a mutable matrix (in place) into upper triangular form.

Note that being immutable does not forbid an object to store knowledge. For example, if it is found out that an immutable list is strictly sorted then the list may store this information. More precisely, an immutable object may change in any way, provided that it continues to represent the same mathematical object.

12.7 Duplication of Objects

12.7.1 ShallowCopy

▷ `ShallowCopy(obj)` (operation)

`ShallowCopy` returns a *new mutable* object *equal* to its argument, if this is possible. The subobjects of `ShallowCopy( obj )` are *identical* to the subobjects of `obj`.

If GAP does not support a mutable form of the immutable object `obj` (see 12.6) then `ShallowCopy` returns `obj` itself.

Since `ShallowCopy` is an operation, the concrete meaning of "subobject" depends on the type of `obj`. But for any copyable object `obj`, the definition should reflect the idea of "first level copying".

The definition of `ShallowCopy` for lists (in particular for matrices) can be found in 21.7.

12.7.2 StructuralCopy

▷ `StructuralCopy(obj)` (function)

In a few situations, one wants to make a *structural copy* scp of an object *obj*. This is defined as follows. scp and *obj* are identical if *obj* is immutable. Otherwise, scp is a mutable copy of *obj* such that each subobject of scp is a structural copy of the corresponding subobject of *obj*. Furthermore, if two subobjects of *obj* are identical then also the corresponding subobjects of scp are identical.

Example

```
gap> obj:= [ [ 0, 1 ] ];;
gap> obj[2]:= obj[1];;
gap> obj[3]:= Immutable( obj[1] );;
gap> scp:= StructuralCopy( obj );;
gap> scp = obj; IsIdenticalObj( scp, obj );
true
false
gap> IsIdenticalObj( scp[1], obj[1] );
false
gap> IsIdenticalObj( scp[3], obj[3] );
true
gap> IsIdenticalObj( scp[1], scp[2] );
true
```

That both ShallowCopy (12.7.1) and StructuralCopy return the argument *obj* itself if it is not copyable is consistent with this definition, since there is no way to change *obj* by modifying the result of any of the two functions, because in fact there is no way to change this result at all.

12.8 Other Operations Applicable to any Object

There are a number of general operations which can be applied, in principle, to any object in GAP. Some of these are documented elsewhere –see String (27.6.6), PrintObj (6.3.5) and Display (6.3.6). Others are mainly somewhat technical.

12.8.1 SetName

▷ SetName(*obj, name*) (function)

for a suitable object *obj* sets that object to have name *name* (a string).

12.8.2 Name

▷ Name(*obj*) (attribute)

returns the name, a string, previously assigned to *obj* via a call to SetName (12.8.1). The name of an object is used *only* for viewing the object via this name.

There are no methods installed for computing names of objects, but the name may be set for suitable objects, using SetName (12.8.1).

Example

```
gap> R := PolynomialRing(Integers,2);
Integers[x_1,x_2]
gap> SetName(R,"Z[x,y]");
gap> R;
Z[x,y]
```

```
gap> Name(R);
"Z[x,y]"
```

12.8.3 IsInternallyConsistent

▷ `IsInternallyConsistent(`*obj*`)` (operation)

For debugging purposes, it may be useful to check the consistency of an object *obj* that is composed from other (composed) objects.

There is a default method of `IsInternallyConsistent`, with rank zero, that returns `true`. So it is possible (and recommended) to check the consistency of subobjects of *obj* recursively by `IsInternallyConsistent`.

(Note that `IsInternallyConsistent` is not an attribute.)

12.8.4 MemoryUsage

▷ `MemoryUsage(`*obj*`)` (operation)

returns the amount of memory in bytes used by the object *obj* and its subobjects. Note that in general, objects can reference each other in very difficult ways such that determining the memory usage is a recursive procedure. In particular, computing the memory usage of a complicated structure itself uses some additional memory, which is however no longer used after completion of this operation. This procedure descends into lists and records, positional and component objects, however it does not take into account the type and family objects! For functions, it only takes the memory usage of the function body, not of the local context the function was created in, although the function keeps a reference to that as well!

Chapter 13

Types of Objects

Every GAP object has a *type*. The type of an object is the information which is used to decide whether an operation is admissible or possible with that object as an argument, and if so, how it is to be performed (see Chapter 78).

For example, the types determine whether two objects can be multiplied and what function is called to compute the product. Analogously, the type of an object determines whether and how the size of the object can be computed. It is sometimes useful in discussing the type system, to identify types with the set of objects that have this type. Partial types can then also be regarded as sets, such that any type is the intersection of its parts.

The type of an object consists of two main parts, which describe different aspects of the object.

The *family* determines the relation of the object to other objects. For example, all permutations form a family. Another family consists of all collections of permutations, this family contains the set of permutation groups as a subset. A third family consists of all rational functions with coefficients in a certain family.

The other part of a type is a collection of *filters* (actually stored as a bit-list indicating, from the complete set of possible filters, which are included in this particular type). These filters are all treated equally by the method selection, but, from the viewpoint of their creation and use, they can be divided (with a small number of unimportant exceptions) into categories, representations, attribute testers and properties. Each of these is described in more detail below.

This chapter does not describe how types and their constituent parts can be created. Information about this topic can be found in Chapter 79.

Note: Detailed understanding of the type system is not required to use GAP. It can be helpful, however, to understand how things work and why GAP behaves the way it does.

A discussion of the type system can be found in [BL98].

13.1 Families

The family of an object determines its relationship to other objects.

More precisely, the families form a partition of all GAP objects such that the following two conditions hold: objects that are equal w.r.t. = lie in the same family; and the family of the result of an operation depends only on the families of its operands.

The first condition means that a family can be regarded as a set of elements instead of a set of objects. Note that this does not hold for categories and representations (see below), two objects that are equal w.r.t. = need not lie in the same categories and representations. For example, a sparsely

represented matrix can be equal to a densely represented matrix. Similarly, each domain is equal w.r.t. = to the sorted list of its elements, but a domain is not a list, and a list is not a domain.

13.1.1 FamilyObj

▷ `FamilyObj(`*obj*`)` (function)

returns the family of the object *obj*.

The family of the object *obj* is itself an object, its family is `FamilyOfFamilies`.

It should be emphasized that families may be created when they are needed. For example, the family of elements of a finitely presented group is created only after the presentation has been constructed. Thus families are the dynamic part of the type system, that is, the part that is not fixed after the initialisation of GAP.

Families can be parametrized. For example, the elements of each finitely presented group form a family of their own. Here the family of elements and the finitely presented group coincide when viewed as sets. Note that elements in different finitely presented groups lie in different families. This distinction allows GAP to forbid multiplications of elements in different finitely presented groups.

As a special case, families can be parametrized by other families. An important example is the family of *collections* that can be formed for each family. A collection consists of objects that lie in the same family, it is either a nonempty dense list of objects from the same family or a domain.

Note that every domain is a collection, that is, it is not possible to construct domains whose elements lie in different families. For example, a polynomial ring over the rationals cannot contain the integer `0` because the family that contains the integers does not contain polynomials. So one has to distinguish the integer zero from each zero polynomial.

Let us look at this example from a different viewpoint. A polynomial ring and its coefficients ring lie in different families, hence the coefficients ring cannot be embedded "naturally" into the polynomial ring in the sense that it is a subset. But it is possible to allow, e.g., the multiplication of an integer and a polynomial over the integers. The relation between the arguments, namely that one is a coefficient and the other a polynomial, can be detected from the relation of their families. Moreover, this analysis is easier than in a situation where the rationals would lie in one family together with all polynomials over the rationals, because then the relation of families would not distinguish the multiplication of two polynomials, the multiplication of two coefficients, and the multiplication of a coefficient with a polynomial. So the wish to describe relations between elements can be taken as a motivation for the introduction of families.

13.2 Filters

A *filter* is a special unary GAP function that returns either `true` or `false`, depending on whether or not the argument lies in the set defined by the filter. Filters are used to express different aspects of information about a GAP object, which are described below (see 13.3, 13.4, 13.5, 13.6, 13.7, 13.8).

Presently any filter in GAP is implemented as a function which corresponds to a set of positions in the bitlist which forms part of the type of each GAP object, and returns `true` if and only if the bitlist of the type of the argument has the value `true` at all of these positions.

The intersection (or meet) of two filters *filt1*, *filt2* is again a filter, it can be formed as

filt1 `and` *filt2*

See 20.4 for more details.

For example, IsList and IsEmpty is a filter that returns true if its argument is an empty list, and false otherwise. The filter IsGroup (39.2.7) is defined as the intersection of the category IsMagmaWithInverses (35.1.4) and the property IsAssociative (35.4.7).

A filter that is not the meet of other filters is called a *simple filter*. For example, each attribute tester (see 13.6) is a simple filter. Each simple filter corresponds to a position in the bitlist currently used as part of the data structure representing a type.

Every filter has a *rank*, which is used to define a ranking of the methods installed for an operation, see Section 78.2. The rank of a filter can be accessed with RankFilter (13.2.1).

13.2.1 RankFilter

▷ RankFilter(*filt*) (function)

For simple filters, an *incremental rank* is defined when the filter is created, see the sections about the creation of filters: 79.1, 79.2, 79.3, 79.4. For an arbitrary filter, its rank is given by the sum of the incremental ranks of the *involved* simple filters; in addition to the implied filters, these are also the required filters of attributes (again see the sections about the creation of filters). In other words, for the purpose of computing the rank and *only* for this purpose, attribute testers are treated as if they would imply the requirements of their attributes.

13.2.2 NamesFilter

▷ NamesFilter(*filt*) (function)

NamesFilter returns a list of names of the *implied* simple filters of the filter *filt*, these are all those simple filters imp such that every object in *filt* also lies in imp. For implications between filters, see ShowImpliedFilters (13.2.3) as well as sections 78.7, 79.1, 79.2, 79.3.

13.2.3 ShowImpliedFilters

▷ ShowImpliedFilters(*filter*) (function)

Displays information about the filters that may be implied by *filter*. They are given by their names. ShowImpliedFilters first displays the names of all filters that are unconditionally implied by *filter*. It then displays implications that require further filters to be present (indicating by + the required further filters). The function displays only first-level implications, implications that follow in turn are not displayed (though GAP will do these).

Example

```
gap> ShowImpliedFilters(IsMatrix);
Implies:
   IsGeneralizedRowVector
   IsNearAdditiveElementWithInverse
   IsAdditiveElement
   IsMultiplicativeElement


May imply with:
+IsGF2MatrixRep
   IsOrdinaryMatrix
```

```
+CategoryCollections(CategoryCollections(IsAdditivelyCommutativeElement))
   IsAdditivelyCommutativeElement

+IsInternalRep
   IsOrdinaryMatrix
```

13.3 Categories

The *categories* of an object are filters (see 13.2) that determine what operations an object admits. For example, all integers form a category, all rationals form a category, and all rational functions form a category. An object which claims to lie in a certain category is accepting the requirement that it should have methods for certain operations (and perhaps that their behaviour should satisfy certain axioms). For example, an object lying in the category `IsList` (21.1.1) must have methods for `Length` (21.17.5), `IsBound\[\]` (21.2.1) and the list element access operation `\[\]` (21.2.1).

An object can lie in several categories. For example, a row vector lies in the categories `IsList` (21.1.1) and `IsVector` (31.14.14); each list lies in the category `IsCopyable` (12.6.1), and depending on whether or not it is mutable, it may lie in the category `IsMutable` (12.6.2). Every domain lies in the category `IsDomain` (31.9.1).

Of course some categories of a mutable object may change when the object is changed. For example, after assigning values to positions of a mutable non-dense list, this list may become part of the category `IsDenseList` (21.1.2).

However, if an object is immutable then the set of categories it lies in is fixed.

All categories in the library are created during initialization, in particular they are not created dynamically at runtime.

The following list gives an overview of important categories of arithmetic objects. Indented categories are to be understood as subcategories of the non indented category listed above it.

Example

```
IsObject
    IsExtLElement
    IsExtRElement
        IsMultiplicativeElement
            IsMultiplicativeElementWithOne
                IsMultiplicativeElementWithInverse
    IsExtAElement
        IsAdditiveElement
            IsAdditiveElementWithZero
                IsAdditiveElementWithInverse
```

Every object lies in the category `IsObject` (12.1.1).

The categories `IsExtLElement` (31.14.8) and `IsExtRElement` (31.14.9) contain objects that can be multiplied with other objects via `*` from the left and from the right, respectively. These categories are required for the operands of the operation `*`.

The category `IsMultiplicativeElement` (31.14.10) contains objects that can be multiplied from the left and from the right with objects from the same family. `IsMultiplicativeElementWithOne` (31.14.11) contains objects `obj` for which a multiplicatively neutral element can be obtained by taking the 0-th power `obj^0`.

IsMultiplicativeElementWithInverse (31.14.13) contains objects obj for which a multiplicative inverse can be obtained by forming obj^-1.

Likewise, the categories IsExtAElement (31.14.1), IsAdditiveElement (31.14.3), IsAdditiveElementWithZero (31.14.5) and IsAdditiveElementWithInverse (31.14.7) contain objects that can be added via + to other objects, objects that can be added to objects of the same family, objects for which an additively neutral element can be obtained by multiplication with zero, and objects for which an additive inverse can be obtained by multiplication with -1.

So a vector lies in IsExtLElement (31.14.8), IsExtRElement (31.14.9) and IsAdditiveElementWithInverse (31.14.7). A ring element must additionally lie in IsMultiplicativeElement (31.14.10).

As stated above it is not guaranteed by the categories of objects whether the result of an operation with these objects as arguments is defined. For example, the category IsMatrix (24.2.1) is a subcategory of IsMultiplicativeElementWithInverse (31.14.13). Clearly not every matrix has a multiplicative inverse. But the category IsMatrix (24.2.1) makes each matrix an admissible argument of the operation Inverse (31.10.8), which may sometimes return fail. Likewise, two matrices can be multiplied only if they are of appropriate shapes.

Analogous to the categories of arithmetic elements, there are categories of domains of these elements.

Example

```
IsObject
    IsDomain
        IsMagma
            IsMagmaWithOne
                IsMagmaWithInversesIfNonzero
                    IsMagmaWithInverses
        IsAdditiveMagma
            IsAdditiveMagmaWithZero
                IsAdditiveMagmaWithInverses
        IsExtLSet
        IsExtRSet
```

Of course IsDomain (31.9.1) is a subcategory of IsObject (12.1.1). A domain that is closed under multiplication * is called a magma and it lies in the category IsMagma (35.1.1). If a magma is closed under taking the identity, it lies in IsMagmaWithOne (35.1.2), and if it is also closed under taking inverses, it lies in IsMagmaWithInverses (35.1.4). The category IsMagmaWithInversesIfNonzero (35.1.3) denotes closure under taking inverses only for nonzero elements, every division ring lies in this category.

Note that every set of categories constitutes its own notion of generation, for example a group may be generated as a magma with inverses by some elements, but to generate it as a magma with one it may be necessary to take the union of these generators and their inverses.

13.3.1 CategoriesOfObject

▷ CategoriesOfObject(*object*) (operation)

returns a list of the names of the categories in which *object* lies.

Example

```
gap> g:=Group((1,2),(1,2,3));;
gap> CategoriesOfObject(g);
```

```
[ "IsListOrCollection", "IsCollection", "IsExtLElement",
  "CategoryCollections(IsExtLElement)", "IsExtRElement",
  "CategoryCollections(IsExtRElement)",
  "CategoryCollections(IsMultiplicativeElement)",
  "CategoryCollections(IsMultiplicativeElementWithOne)",
  "CategoryCollections(IsMultiplicativeElementWithInverse)",
  "CategoryCollections(IsAssociativeElement)",
  "CategoryCollections(IsFiniteOrderElement)", "IsGeneralizedDomain",
  "CategoryCollections(IsPerm)", "IsMagma", "IsMagmaWithOne",
  "IsMagmaWithInversesIfNonzero", "IsMagmaWithInverses" ]
```

13.4 Representation

The *representation* of an object is a set of filters (see 13.2) that determines how an object is actually represented. For example, a matrix or a polynomial can be stored sparsely or densely; all dense polynomials form a representation. An object which claims to lie in a certain representation is accepting the requirement that certain fields in the data structure be present and have specified meanings.

GAP distinguishes four essentially different ways to represent objects. First there are the representations `IsInternalRep` for internal objects such as integers and permutations, and `IsDataObjectRep` for other objects that are created and whose data are accessible only by kernel functions. The data structures underlying such objects cannot be manipulated at the GAP level.

All other objects are either in the representation `IsComponentObjectRep` or in the representation `IsPositionalObjectRep`, see 79.10 and 79.11.

An object can belong to several representations in the sense that it lies in several subrepresentations of `IsComponentObjectRep` or of `IsPositionalObjectRep`. The representations to which an object belongs should form a chain and either two representations are disjoint or one is contained in the other. So the subrepresentations of `IsComponentObjectRep` and `IsPositionalObjectRep` each form trees. In the language of Object Oriented Programming, we support only single inheritance for representations.

These trees are typically rather shallow, since for one representation to be contained in another implies that all the components of the data structure implied by the containing representation, are present in, and have the same meaning in, the smaller representation (whose data structure presumably contains some additional components).

Objects may change their representation, for example a mutable list of characters can be converted into a string.

All representations in the library are created during initialization, in particular they are not created dynamically at runtime.

Examples of subrepresentations of `IsPositionalObjectRep` are `IsModulusRep`, which is used for residue classes in the ring of integers, and `IsDenseCoeffVectorRep`, which is used for elements of algebras that are defined by structure constants.

An important subrepresentation of `IsComponentObjectRep` is `IsAttributeStoringRep`, which is used for many domains and some other objects. It provides automatic storing of all attribute values (see below).

13.4.1 RepresentationsOfObject

▷ RepresentationsOfObject(*object*) (operation)

returns a list of the names of the representations *object* has.

Example

```
gap> g:=Group((1,2),(1,2,3));;
gap> RepresentationsOfObject(g);
[ "IsComponentObjectRep", "IsAttributeStoringRep" ]
```

13.5 Attributes

The attributes of an object describe knowledge about it.

An attribute is a unary operation without side-effects.

An object may store values of its attributes once they have been computed, and claim that it knows these values. Note that "store" and "know" have to be understood in the sense that it is very cheap to get such a value when the attribute is called again.

The stored value of an attribute is in general immutable (see 12.6), except if the attribute had been specially constructed as "mutable attribute".

It depends on the representation of an object (see 13.4) which attribute values it stores. An object in the representation IsAttributeStoringRep stores *all* attribute values once they are computed. Moreover, for an object in this representation, subsequent calls to an attribute will return the *same* object; this is achieved via a special method for each attribute setter that stores the attribute value in an object in IsAttributeStoringRep, and a special method for the attribute itself that fetches the stored attribute value. (These methods are called the "system setter" and the "system getter" of the attribute, respectively.)

Note also that it is impossible to get rid of a stored attribute value because the system may have drawn conclusions from the old attribute value, and just removing the value might leave the data structures in an inconsistent state. If necessary, a new object can be constructed.

Several attributes have methods for more than one argument. For example IsTransitive (41.10.1) is an attribute for a G-set that can also be called for the two arguments, being a group G and its action domain. If attributes are called with more than one argument then the return value is not stored in any of the arguments.

Properties are a special form of attributes that have the value true or false, see section 13.7.

Examples of attributes for multiplicative elements are Inverse (31.10.8), One (31.10.2), and Order (31.10.10). Size (30.4.6) is an attribute for domains, Centre (35.4.5) is an attribute for magmas, and DerivedSubgroup (39.12.3) is an attribute for groups.

13.5.1 KnownAttributesOfObject

▷ KnownAttributesOfObject(*object*) (operation)

returns a list of the names of the attributes whose values are known for *object*.

Example

```
gap> g:=Group((1,2),(1,2,3));;Size(g);;
gap> KnownAttributesOfObject(g);
[ "Size", "OneImmutable", "NrMovedPoints", "MovedPoints",
  "GeneratorsOfMagmaWithInverses", "MultiplicativeNeutralElement",
```

```
    "HomePcgs", "Pcgs", "GeneralizedPcgs", "StabChainMutable",
    "StabChainOptions" ]
```

13.6 Setter and Tester for Attributes

For every attribute, the *attribute setter* and the *attribute tester* are defined.

To check whether an object belongs to an attribute *attr*, the tester of the attribute is used, see Tester (13.6.1). To store a value for the attribute *attr* in an object, the setter of the attribute is used, see Setter (13.6.2).

13.6.1 Tester

▷ Tester(*attr*) (function)

For an attribute *attr*, Tester(*attr*) is a filter (see 13.2) that returns true or false, depending on whether or not the value of *attr* for the object is known. For example, Tester(Size)(*obj*) is true if the size of the object *obj* is known.

13.6.2 Setter

▷ Setter(*attr*) (function)

For an attribute *attr*, Setter(*attr*) is called automatically when the attribute value has been computed for the first time. One can also call the setter explicitly, for example, Setter(Size)(*obj*, *val*) sets *val* as size of the object *obj* if the size was not yet known.

For each attribute *attr* that is declared with DeclareAttribute (79.18.3) resp. DeclareProperty (79.18.4) (see 79.18), tester and setter are automatically made accessible by the names Has*attr* and Set*attr*, respectively. For example, the tester for Size (30.4.6) is called HasSize, and the setter is called SetSize.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;Size(g);
24
gap> HasSize(g);
true
gap> SetSize(g,99);
gap> Size(g);
24
```

For two properties *prop1* and *prop2*, the intersection *prop1* and *prop2* (see 13.2) is again a property for which a setter and a tester exist. Setting the value of this intersection to true for a GAP object means to set the values of *prop1* and *prop2* to true for this object.

Example

```
gap> prop:= IsFinite and IsCommutative;
<Property "<<and-filter>>">
gap> g:= Group( (1,2,3), (4,5) );;
gap> Tester( prop )( g );
false
gap> Setter( prop )( g, true );
```

```
gap> Tester( prop )( g );  prop( g );
true
true
```

It is *not allowed* to set the value of such an intersection to `false` for an object.

Example

```
gap> Setter( prop )( Rationals, false );
You cannot set an "and-filter" except to true
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can type 'return true;' to set all components true
(but you might really want to reset just one component) to continue
brk>
```

13.6.3 AttributeValueNotSet

▷ `AttributeValueNotSet(`*attr, obj*`)` (function)

If the value of the attribute *attr* is already stored for *obj*, `AttributeValueNotSet` simply returns this value. Otherwise the value of *attr*`( `*obj*` )` is computed and returned *without storing it* in *obj*. This can be useful when "large" attribute values (such as element lists) are needed only once and shall not be stored in the object.

Example

```
gap> HasAsSSortedList(g);
false
gap> AttributeValueNotSet(AsSSortedList,g);
[ (), (4,5), (1,2,3), (1,2,3)(4,5), (1,3,2), (1,3,2)(4,5) ]
gap> HasAsSSortedList(g);
false
```

The normal behaviour of attributes (when called with just one argument) is that once a method has been selected and executed, and has returned a value the setter of the attribute is called, to (possibly) store the computed value. In special circumstances, this behaviour can be altered dynamically on an attribute-by-attribute basis, using the functions `DisableAttributeValueStoring` and `EnableAttributeValueStoring`.

In general, the code in the library assumes, for efficiency, but not for correctness, that attribute values *will* be stored (in suitable objects), so disabling storing may cause substantial computations to be repeated.

13.6.4 InfoAttributes

▷ `InfoAttributes` (info class)

This info class (together with `InfoWarning` (7.4.7) is used for messages about attribute storing being disabled (at level 2) or enabled (level 3). It may be used in the future for other messages concerning changes to attribute behaviour.

13.6.5 DisableAttributeValueStoring

▷ DisableAttributeValueStoring(*attr*) (function)

disables the usual call of Setter(*attr*) when a method for *attr* returns a value. In consequence the values will never be stored. Note that *attr* must be an attribute and *not* a property.

13.6.6 EnableAttributeValueStoring

▷ EnableAttributeValueStoring(*attr*) (function)

enables the usual call of Setter(*attr*) when a method for *attr* returns a value. In consequence the values may be stored. This will usually have no effect unless DisableAttributeValueStoring has previously been used for *attr*. Note that *attr* must be an attribute and *not* a property.

Example

```
gap> g := Group((1,2,3,4,5),(1,2,3));
Group([ (1,2,3,4,5), (1,2,3) ])
gap> KnownAttributesOfObject(g);
[ "LargestMovedPoint", "GeneratorsOfMagmaWithInverses",
  "MultiplicativeNeutralElement" ]
gap> SetInfoLevel(InfoAttributes,3);
gap> DisableAttributeValueStoring(Size);
#I  Disabling value storing for Size
gap> Size(g);
60
gap> KnownAttributesOfObject(g);
[ "OneImmutable", "LargestMovedPoint", "NrMovedPoints",
  "MovedPoints", "GeneratorsOfMagmaWithInverses",
  "MultiplicativeNeutralElement", "StabChainMutable",
  "StabChainOptions" ]
gap> Size(g);
60
gap> EnableAttributeValueStoring(Size);
#I  Enabling value storing for Size
gap> Size(g);
60
gap> KnownAttributesOfObject(g);
[ "Size", "OneImmutable", "LargestMovedPoint", "NrMovedPoints",
  "MovedPoints", "GeneratorsOfMagmaWithInverses",
  "MultiplicativeNeutralElement", "StabChainMutable",
  "StabChainOptions" ]
```

13.7 Properties

The properties of an object are those of its attributes (see 13.5) whose values can only be true or false.

The main difference between attributes and properties is that a property defines two sets of objects, namely the usual set of all objects for which the value is known, and the set of all objects for which the value is known to be true.

(Note that it makes no sense to consider a third set, namely the set of objects for which the value of a property is `true` whether or not it is known, since there may be objects for which the containment in this set cannot be decided.)

For a property *prop*, the containment of an object *obj* in the first set is checked again by applying `Tester(` *prop* `)` to *obj*, and *obj* lies in the second set if and only if `Tester(` *prop* `)(` *obj* `)` and *prop*`(` *obj* `)` is `true`.

If a property value is known for an immutable object then this value is also stored, as part of the type of the object. To some extent, property values of mutable objects also can be stored, for example a mutable list all of whose entries are immutable can store whether it is strictly sorted. When the object is mutated (for example by list assignment) the type may need to be adjusted.

Important properties for domains are `IsAssociative` (35.4.7), `IsCommutative` (35.4.9), `IsAnticommutative` (56.4.6), `IsLDistributive` (56.4.3) and `IsRDistributive` (56.4.4), which mean that the multiplication of elements in the domain satisfies $(a*b)*c = a*(b*c)$, $a*b = b*a$, $a*b = -(b*a)$, $a*(b+c) = a*b+a*c$, and $(a+b)*c = a*c+b*c$, respectively, for all a, b, c in the domain.

13.7.1 KnownPropertiesOfObject

▷ `KnownPropertiesOfObject(`*object*`)` (operation)

returns a list of the names of the properties whose values are known for *object*.

13.7.2 KnownTruePropertiesOfObject

▷ `KnownTruePropertiesOfObject(`*object*`)` (operation)

returns a list of the names of the properties known to be `true` for *object*.

Example

```
gap> g:=Group((1,2),(1,2,3));;
gap> KnownPropertiesOfObject(g);
[ "IsFinite", "CanEasilyCompareElements", "CanEasilySortElements",
 "IsDuplicateFree", "IsGeneratorsOfMagmaWithInverses",
 "IsAssociative", "IsGeneratorsOfSemigroup", "IsSimpleSemigroup",
 "IsRegularSemigroup", "IsCompletelyRegularSemigroup",
 "IsCompletelySimpleSemigroup", "IsFinitelyGeneratedGroup",
 "IsSubsetLocallyFiniteGroup", "KnowsHowToDecompose",
 "IsNilpotentByFinite" ]
gap> Size(g);
6
gap> KnownPropertiesOfObject(g);
[ "IsEmpty", "IsTrivial", "IsNonTrivial", "IsFinite",
 "CanEasilyCompareElements", "CanEasilySortElements",
 "IsDuplicateFree", "IsGeneratorsOfMagmaWithInverses",
 "IsAssociative", "IsGeneratorsOfSemigroup", "IsSimpleSemigroup",
 "IsRegularSemigroup", "IsCompletelyRegularSemigroup",
 "IsCompletelySimpleSemigroup", "IsFinitelyGeneratedGroup",
 "IsSubsetLocallyFiniteGroup", "KnowsHowToDecompose",
 "IsPerfectGroup", "IsSolvableGroup", "IsPolycyclicGroup",
 "IsNilpotentByFinite", "IsTorsionFree", "IsFreeAbelian" ]
gap> KnownTruePropertiesOfObject(g);
```

```
[ "IsNonTrivial", "IsFinite", "CanEasilyCompareElements",
 "CanEasilySortElements", "IsDuplicateFree",
 "IsGeneratorsOfMagmaWithInverses", "IsAssociative",
 "IsGeneratorsOfSemigroup", "IsSimpleSemigroup",
 "IsRegularSemigroup", "IsCompletelyRegularSemigroup",
 "IsCompletelySimpleSemigroup", "IsFinitelyGeneratedGroup",
 "IsSubsetLocallyFiniteGroup", "KnowsHowToDecompose",
 "IsSolvableGroup", "IsPolycyclicGroup", "IsNilpotentByFinite" ]
```

13.8 Other Filters

There are situations where one wants to express a kind of knowledge that is based on some heuristic.

For example, the filters (see 13.2) `CanEasilyTestMembership` (39.25.1) and `CanEasilyComputePcgs` (45.2.3) are defined in the GAP library. Note that such filters do not correspond to a mathematical concept, contrary to properties (see 13.7). Also it need not be defined what "easily" means for an arbitrary GAP object, and in this case one cannot compute the value for an arbitrary GAP object. In order to access this kind of knowledge as a part of the type of an object, GAP provides filters for which the value is `false` by default, and it is changed to `true` in certain situations, either explicitly (for the given object) or via a logical implication (see 78.7) from other filters.

For example, a `true` value of `CanEasilyComputePcgs` (45.2.3) for a group means that certain methods are applicable that use a pcgs (see 45.1) for the group. There are logical implications to set the filter value to `true` for permutation groups that are known to be solvable, and for groups that have already a (sufficiently nice) pcgs stored. In the case one has a solvable matrix group and wants to enable methods that use a pcgs, one can set the `CanEasilyComputePcgs` (45.2.3) value to `true` for this particular group.

A filter *`filt`* of the kind described here is different from the other filters introduced in the previous sections. In particular, *`filt`* is not a category (see 13.3) or a property (see 13.7) because its value may change for a given object, and *`filt`* is not a representation (see 13.4) because it has nothing to do with the way an object is made up from some data. *`filt`* is similar to an attribute tester (see 13.6), the only difference is that *`filt`* does not refer to an attribute value; note that *`filt`* is also used in the same way as an attribute tester; namely, the `true` value may be required for certain methods to be applicable.

13.9 Types

We stated above (see 13) that, for an object *`obj`*, its *type* is formed from its family and its filters. There is a also a third component, used in a few situations, namely defining data of the type.

13.9.1 TypeObj

▷ `TypeObj(`*`obj`*`)` (function)

returns the type of the object *`obj`*.
The type of an object is itself an object.

Two types are equal if and only if the two families are identical, the filters are equal, and, if present, also the defining data of the types are equal.

13.9.2 DataType

▷ `DataType(`*type*`)` (function)

The last part of the type, defining data, has not been mentioned before and seems to be of minor importance. It can be used, e.g., for cosets Ug of a group U, where the type of each coset may contain the group U as defining data. As a consequence, two such cosets mod U and V can have the same type only if $U = V$. The defining data of the type *type* can be accessed via `DataType`.

Chapter 14

Integers

One of the most fundamental datatypes in every programming language is the integer type. GAP is no exception.

GAP integers are entered as a sequence of decimal digits optionally preceded by a "+" sign for positive integers or a "-" sign for negative integers. The size of integers in GAP is only limited by the amount of available memory, so you can compute with integers having thousands of digits.

Example

```
gap> -1234;
-1234
gap> 123456789012345678901234567890123456789012345678901234567890;
123456789012345678901234567890123456789012345678901234567890
```

Many more functions that are mainly related to the prime residue group of integers modulo an integer are described in chapter 15, and functions dealing with combinatorics can be found in chapter 16.

14.1 Integers: Global Variables

14.1.1 Integers (global variable)

▷ `Integers` (global variable)
▷ `PositiveIntegers` (global variable)
▷ `NonnegativeIntegers` (global variable)

These global variables represent the ring of integers and the semirings of positive and nonnegative integers, respectively.

Example

```
gap> Size( Integers ); 2 in Integers;
infinity
true
```

`Integers` is a subset of `Rationals` (17.1.1), which is a subset of `Cyclotomics` (18.1.2). See Chapter 18 for arithmetic operations and comparison of integers.

14.1.2 IsIntegers

▷ IsIntegers(*obj*) (Category)
▷ IsPositiveIntegers(*obj*) (Category)
▷ IsNonnegativeIntegers(*obj*) (Category)

are the defining categories for the domains Integers (14.1.1), PositiveIntegers (14.1.1), and NonnegativeIntegers (14.1.1).

Example

```
gap> IsIntegers( Integers );  IsIntegers( Rationals );  IsIntegers( 7 );
true
false
false
```

14.2 Elementary Operations for Integers

14.2.1 IsInt

▷ IsInt(*obj*) (Category)

Every rational integer lies in the category IsInt, which is a subcategory of IsRat (17.2.1).

14.2.2 IsPosInt

▷ IsPosInt(*obj*) (Category)

Every positive integer lies in the category IsPosInt.

14.2.3 Int

▷ Int(*elm*) (attribute)

Int returns an integer int whose meaning depends on the type of *elm*.

If *elm* is a rational number (see Chapter 17) then int is the integer part of the quotient of numerator and denominator of *elm* (see QuoInt (14.3.1)).

If *elm* is an element of a finite prime field (see Chapter 59) then int is the smallest nonnegative integer such that *elm* = int * One(*elm*).

If *elm* is a string (see Chapter 27) consisting of digits '0', '1', ..., '9' and '-' (at the first position) then int is the integer described by this string. The operation String (27.6.6) can be used to compute a string for rational integers, in fact for all cyclotomics.

Example

```
gap> Int( 4/3 );  Int( -2/3 );
1
0
gap> int:= Int( Z(5) );  int * One( Z(5) );
2
Z(5)
gap> Int( "12345" );  Int( "-27" );  Int( "-27/3" );
12345
```

```
-27
fail
```

14.2.4 IsEvenInt

▷ IsEvenInt(*n*) (function)

tests if the integer *n* is divisible by 2.

14.2.5 IsOddInt

▷ IsOddInt(*n*) (function)

tests if the integer *n* is not divisible by 2.

14.2.6 AbsInt

▷ AbsInt(*n*) (function)

AbsInt returns the absolute value of the integer *n*, i.e., *n* if *n* is positive, -*n* if *n* is negative and 0 if *n* is 0.

AbsInt is a special case of the general operation EuclideanDegree (56.6.2).

See also AbsoluteValue (18.1.8).

Example

```
gap> AbsInt( 33 );
33
gap> AbsInt( -214378 );
214378
gap> AbsInt( 0 );
0
```

14.2.7 SignInt

▷ SignInt(*n*) (function)

SignInt returns the sign of the integer *n*, i.e., 1 if *n* is positive, -1 if *n* is negative and 0 if *n* is 0.

Example

```
gap> SignInt( 33 );
1
gap> SignInt( -214378 );
-1
gap> SignInt( 0 );
0
```

14.2.8 LogInt

▷ LogInt(*n*, *base*) (function)

`LogInt` returns the integer part of the logarithm of the positive integer `n` with respect to the positive integer `base`, i.e., the largest positive integer e such that $base^e \leq n$. The function `LogInt` will signal an error if either `n` or `base` is not positive.

For $base = 2$ this is very efficient because the internal binary representation of the integer is used.

Example

```
gap> LogInt( 1030, 2 );
10
gap> 2^10;
1024
gap> LogInt( 1, 10 );
0
```

14.2.9 RootInt

▷ `RootInt(n[, k])` (function)

`RootInt` returns the integer part of the `k`th root of the integer `n`. If the optional integer argument `k` is not given it defaults to 2, i.e., `RootInt` returns the integer part of the square root in this case.

If `n` is positive, `RootInt` returns the largest positive integer r such that $r^k \leq n$. If `n` is negative and `k` is odd `RootInt` returns `-RootInt( -n, k )`. If `n` is negative and `k` is even `RootInt` will cause an error. `RootInt` will also cause an error if `k` is 0 or negative.

Example

```
gap> RootInt( 361 );
19
gap> RootInt( 2 * 10^12 );
1414213
gap> RootInt( 17000, 5 );
7
gap> 7^5;
16807
```

14.2.10 SmallestRootInt

▷ `SmallestRootInt(n)` (function)

`SmallestRootInt` returns the smallest root of the integer `n`.

The smallest root of an integer `n` is the integer r of smallest absolute value for which a positive integer k exists such that $n = r^k$.

Example

```
gap> SmallestRootInt( 2^30 );
2
gap> SmallestRootInt( -(2^30) );
-4
```

Note that $(-2)^{30} = +(2^{30})$.

Example

```
gap> SmallestRootInt( 279936 );
6
gap> LogInt( 279936, 6 );
```

```
7
gap> SmallestRootInt( 1001 );
1001
```

14.2.11 ListOfDigits

▷ `ListOfDigits(n)` (function)

For a positive integer `n` this function returns a list `l`, consisting of the digits of `n` in decimal representation.

Example

```
gap> ListOfDigits(3142);
[ 3, 1, 4, 2 ]
```

14.2.12 Random (for integers)

▷ `Random(Integers)` (function)

`Random` for integers returns pseudo random integers between -10 and 10 distributed according to a binomial distribution. To generate uniformly distributed integers from a range, use the construction `Random( [ low .. high ] )` (see `Random` (30.7.1)).

14.3 Quotients and Remainders

14.3.1 QuoInt

▷ `QuoInt(n, m)` (function)

`QuoInt` returns the integer part of the quotient of its integer operands.

If `n` and `m` are positive, `QuoInt` returns the largest positive integer q such that $q*m \leq n$. If `n` or `m` or both are negative the absolute value of the integer part of the quotient is the quotient of the absolute values of `n` and `m`, and the sign of it is the product of the signs of `n` and `m`.

`QuoInt` is used in a method for the general operation `EuclideanQuotient` (56.6.3).

Example

```
gap> QuoInt(5,3);  QuoInt(-5,3);  QuoInt(5,-3);  QuoInt(-5,-3);
1
-1
-1
1
```

14.3.2 BestQuoInt

▷ `BestQuoInt(n, m)` (function)

`BestQuoInt` returns the best quotient q of the integers `n` and `m`. This is the quotient such that $n - q*m$ has minimal absolute value. If there are two quotients whose remainders have the same absolute value, then the quotient with the smaller absolute value is chosen.

Example

```
gap> BestQuoInt( 5, 3 );  BestQuoInt( -5, 3 );
2
-2
```

14.3.3 RemInt

▷ RemInt(*n*, *m*) (function)

RemInt returns the remainder of its two integer operands.

If *m* is not equal to zero, RemInt returns *n* - *m* * QuoInt(*n*, *m*). Note that the rules given for QuoInt (14.3.1) imply that the return value of RemInt has the same sign as *n* and its absolute value is strictly less than the absolute value of *m*. Note also that the return value equals *n* mod *m* when both *n* and *m* are nonnegative. Dividing by 0 signals an error.

RemInt is used in a method for the general operation EuclideanRemainder (56.6.4).

Example

```
gap> RemInt(5,3);  RemInt(-5,3);  RemInt(5,-3);  RemInt(-5,-3);
2
-2
2
-2
```

14.3.4 GcdInt

▷ GcdInt(*m*, *n*) (function)

GcdInt returns the greatest common divisor of its two integer operands *m* and *n*, i.e., the greatest integer that divides both *m* and *n*. The greatest common divisor is never negative, even if the arguments are. We define GcdInt(*m*, 0) = GcdInt(0, *m*) = AbsInt(*m*) and GcdInt(0, 0) = 0.

GcdInt is a method used by the general function Gcd (56.7.1).

Example

```
gap> GcdInt( 123, 66 );
3
```

14.3.5 Gcdex

▷ Gcdex(*m*, *n*) (function)

returns a record g describing the extended greatest common divisor of *m* and *n*. The component gcd is this gcd, the components coeff1 and coeff2 are integer cofactors such that g.gcd = g.coeff1 * *m* + g.coeff2 * *n*, and the components coeff3 and coeff4 are integer cofactors such that 0 = g.coeff3 * *m* + g.coeff4 * *n*.

If *m* and *n* both are nonzero, AbsInt(g.coeff1) is less than or equal to AbsInt(*n*) / (2 * g.gcd), and AbsInt(g.coeff2) is less than or equal to AbsInt(*m*) / (2 * g.gcd).

If *m* or *n* or both are zero coeff3 is -*n* / g.gcd and coeff4 is *m* / g.gcd.

The coefficients always form a unimodular matrix, i.e., the determinant g.coeff1 * g.coeff4 - g.coeff3 * g.coeff2 is 1 or -1.

Example

```
gap> Gcdex( 123, 66 );
rec( coeff1 := 7, coeff2 := -13, coeff3 := -22, coeff4 := 41,
  gcd := 3 )
```

This means $3 = 7 * 123 - 13 * 66$, $0 = -22 * 123 + 41 * 66$.

Example

```
gap> Gcdex( 0, -3 );
rec( coeff1 := 0, coeff2 := -1, coeff3 := 1, coeff4 := 0, gcd := 3 )
gap> Gcdex( 0, 0 );
rec( coeff1 := 1, coeff2 := 0, coeff3 := 0, coeff4 := 1, gcd := 0 )
```

GcdRepresentation (56.7.3) provides similar functionality over arbitrary Euclidean rings.

14.3.6 LcmInt

▷ LcmInt(*m*, *n*) (function)

returns the least common multiple of the integers *m* and *n*.
LcmInt is a method used by the general operation Lcm (56.7.6).

Example

```
gap> LcmInt( 123, 66 );
2706
```

14.3.7 CoefficientsQadic

▷ CoefficientsQadic(*i*, *q*) (operation)

returns the q-adic representation of the integer *i* as a list l of coefficients satisfying the equality $i = \sum_{j=0} q^j \cdot l[j+1]$ for an integer $q > 1$.

Example

```
gap> l:=CoefficientsQadic(462,3);
[ 0, 1, 0, 2, 2, 1 ]
```

14.3.8 CoefficientsMultiadic

▷ CoefficientsMultiadic(*ints*, *int*) (function)

returns the multiadic expansion of the integer *int* modulo the integers given in *ints* (in ascending order). It returns a list of coefficients in the *reverse* order to that in *ints*.

14.3.9 ChineseRem

▷ ChineseRem(*moduli*, *residues*) (function)

ChineseRem returns the combination of the *residues* modulo the *moduli*, i.e., the unique integer c from [0..Lcm(*moduli*)-1] such that c = *residues*[i] modulo *moduli*[i] for all i, if it exists. If no such combination exists ChineseRem signals an error.

Such a combination does exist if and only if *residues*[i] = *residues*[k] mod Gcd(*moduli*[i], *moduli*[k]) for every pair i, k. Note that this implies that such a combination exists if the moduli are pairwise relatively prime. This is called the Chinese remainder theorem.

Example

```
gap> ChineseRem( [ 2, 3, 5, 7 ], [ 1, 2, 3, 4 ] );
53
gap> ChineseRem( [ 6, 10, 14 ], [ 1, 3, 5 ] );
103
```

Example

```
gap> ChineseRem( [ 6, 10, 14 ], [ 1, 2, 3 ] );
Error, the residues must be equal modulo 2 called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> gap>
```

14.3.10 PowerModInt

▷ PowerModInt(*r*, *e*, *m*) (function)

returns $r^e \pmod m$ for integers *r*, *e* and *m* ($e \geq 0$).

Note that PowerModInt can reduce intermediate results and thus will generally be faster than using *r*^e mod *m*, which would compute r^e first and reduces the result afterwards.

PowerModInt is a method for the general operation PowerMod (56.7.9).

14.4 Prime Integers and Factorization

14.4.1 Primes

▷ Primes (global variable)

Primes is a strictly sorted list of the 168 primes less than 1000.

This is used in IsPrimeInt (14.4.2) and FactorsInt (14.4.7) to cast out small primes quickly.

Example

```
gap> Primes[1];
2
gap> Primes[100];
541
```

14.4.2 IsPrimeInt

▷ IsPrimeInt(*n*) (function)

▷ IsProbablyPrimeInt(*n*) (function)

IsPrimeInt returns false if it can prove that the integer *n* is composite and true otherwise. By convention IsPrimeInt(0) = IsPrimeInt(1) = false and we define IsPrimeInt(-*n*) = IsPrimeInt(*n*).

IsPrimeInt will return true for every prime n. IsPrimeInt will return false for all composite $n < 10^{18}$ and for all composite n that have a factor $p < 1000$. So for integers $n < 10^{18}$, IsPrimeInt is a proper primality test. It is conceivable that IsPrimeInt may return true for some composite $n > 10^{18}$, but no such n is currently known. So for integers $n > 10^{18}$, IsPrimeInt is a probable-primality test. IsPrimeInt will issue a warning when its argument is probably prime but not a proven prime. (The function IsProbablyPrimeInt will do a similar calculation but not issue a warning.) The warning can be switched off by SetInfoLevel(InfoPrimeInt, 0);, the default level is 1 (also see SetInfoLevel (7.4.3)).

If composites that fool IsPrimeInt do exist, they would be extremely rare, and finding one by pure chance might be less likely than finding a bug in GAP. We would appreciate being informed about any example of a composite number n for which IsPrimeInt returns true.

IsPrimeInt is a deterministic algorithm, i.e., the computations involve no random numbers, and repeated calls will always return the same result. IsPrimeInt first does trial divisions by the primes less than 1000. Then it tests that n is a strong pseudoprime w.r.t. the base 2. Finally it tests whether n is a Lucas pseudoprime w.r.t. the smallest quadratic nonresidue of n. A better description can be found in the comment in the library file primality.gi.

The time taken by IsPrimeInt is approximately proportional to the third power of the number of digits of n. Testing numbers with several hundreds digits is quite feasible.

IsPrimeInt is a method for the general operation IsPrime (56.5.8).

Remark: In future versions of GAP we hope to change the definition of IsPrimeInt to return true only for proven primes (currently, we lack a sufficiently good primality proving function). In applications, use explicitly IsPrimeInt or IsProbablyPrimeInt with this change in mind.

Example

```
gap> IsPrimeInt( 2^31 - 1 );
true
gap> IsPrimeInt( 10^42 + 1 );
false
```

14.4.3 PrimalityProof

▷ PrimalityProof(n) (function)

Construct a machine verifiable proof of the primality of (the probable prime) n, following the ideas of [BLS75]. The proof consists of various Fermat and Lucas pseudoprimality tests, which taken as a whole prove the primality. The proof is represented as a list of witnesses of two kinds. The first kind, ["F", divisor, base], indicates a successful Fermat pseudoprimality test, where n is a strong pseudoprime at base with order not divisible by $(n-1)/divisor$. The second kind, ["L", divisor, discriminant, P] indicates a successful Lucas pseudoprimality test, for a quadratic form of given discriminant and middle term P with an extra check at $(n+1)/divisor$.

14.4.4 IsPrimePowerInt

▷ IsPrimePowerInt(n) (function)

IsPrimePowerInt returns true if the integer n is a prime power and false otherwise.

An integer n is a *prime power* if there exists a prime p and a positive integer i such that $p^i = n$. If n is negative the condition is that there must exist a negative prime p and an odd positive integer i such that $p^i = n$. The integers 1 and -1 are not prime powers.

Note that IsPrimePowerInt uses SmallestRootInt (14.2.10) and a probable-primality test (see IsPrimeInt (14.4.2)).

Example

```
gap> IsPrimePowerInt( 31^5 );
true
gap> IsPrimePowerInt( 2^31-1 );  # 2^31-1 is actually a prime
true
gap> IsPrimePowerInt( 2^63-1 );
false
gap> Filtered( [-10..10], IsPrimePowerInt );
[ -8, -7, -5, -3, -2, 2, 3, 4, 5, 7, 8, 9 ]
```

14.4.5 NextPrimeInt

▷ NextPrimeInt(*n*) (function)

NextPrimeInt returns the smallest prime which is strictly larger than the integer *n*.

Note that NextPrimeInt uses a probable-primality test (see IsPrimeInt (14.4.2)).

Example

```
gap> NextPrimeInt( 541 ); NextPrimeInt( -1 );
547
2
```

14.4.6 PrevPrimeInt

▷ PrevPrimeInt(*n*) (function)

PrevPrimeInt returns the largest prime which is strictly smaller than the integer *n*.

Note that PrevPrimeInt uses a probable-primality test (see IsPrimeInt (14.4.2)).

Example

```
gap> PrevPrimeInt( 541 ); PrevPrimeInt( 1 );
523
-2
```

14.4.7 FactorsInt

▷ FactorsInt(*n*) (function)

▷ FactorsInt(*n: RhoTrials := trials*) (function)

FactorsInt returns a list of factors of a given integer *n* such that Product(FactorsInt(*n*)) = *n*. If $|n| \leq 1$ the list [*n*] is returned. Otherwise the result contains probable primes, sorted by absolute value. The entries will all be positive except for the first one in case of a negative *n*.

See PrimeDivisors (14.4.8) for a function that returns a set of (probable) primes dividing *n*.

Note that FactorsInt uses a probable-primality test (see IsPrimeInt (14.4.2)). Thus FactorsInt might return a list which contains composite integers. In such a case you will get a warning about the use of a probable prime. You can switch off these warnings by SetInfoLevel(InfoPrimeInt, 0); (also see SetInfoLevel (7.4.3)).

The time taken by FactorsInt is approximately proportional to the square root of the second largest prime factor of *n*, which is the last one that FactorsInt has to find, since the largest factor

is simply what remains when all others have been removed. Thus the time is roughly bounded by the fourth root of *n*. FactorsInt is guaranteed to find all factors less than 10^6 and will find most factors less than 10^{10}. If *n* contains multiple factors larger than that FactorsInt may not be able to factor *n* and will then signal an error.

FactorsInt is used in a method for the general operation Factors (56.5.9).

In the second form, FactorsInt calls FactorsRho with a limit of *trials* on the number of trials it performs. The default is 8192. Note that Pollard's Rho is the fastest method for finding prime factors with roughly 5-10 decimal digits, but becomes more and more inferior to other factorization techniques like e.g. the Elliptic Curves Method (ECM) the bigger the prime factors are. Therefore instead of performing a huge number of Rho *trials*, it is usually more advisable to install the FactInt package and then simply to use the operation Factors (56.5.9). The factorization of the 8-th Fermat number by Pollard's Rho below takes already a while.

Example

```
gap> FactorsInt( -Factorial(6) );
[ -2, 2, 2, 2, 3, 3, 5 ]
gap> Set( FactorsInt( Factorial(13)/11 ) );
[ 2, 3, 5, 7, 13 ]
gap> FactorsInt( 2^63 - 1 );
[ 7, 7, 73, 127, 337, 92737, 649657 ]
gap> FactorsInt( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
gap> FactorsInt(2^256+1:RhoTrials:=100000000);
[ 1238926361552897,
  93461639715357977769163558199606896584051237541638188580280321 ]
```

14.4.8 PrimeDivisors

▷ PrimeDivisors(*n*) (attribute)

PrimeDivisors returns for a non-zero integer *n* a set of its positive (probable) primes divisors. In rare cases the result could contain a composite number which passed certain primality tests, see IsProbablyPrimeInt (14.4.2) and FactorsInt (14.4.7) for more details.

Example

```
gap> PrimeDivisors(-12);
[ 2, 3 ]
gap> PrimeDivisors(1);
[  ]
```

14.4.9 PartialFactorization

▷ PartialFactorization(*n[, effort]*) (operation)

PartialFactorization returns a partial factorization of the integer *n*. No assertions are made about the primality of the factors, except of those mentioned below.

The argument *effort*, if given, specifies how intensively the function should try to determine factors of *n*. The default is *effort* = 5.

- If *effort* = 0, trial division by the primes below 100 is done. Returned factors below 10^4 are guaranteed to be prime.

- If *effort* = 1, trial division by the primes below 1000 is done. Returned factors below 10^6 are guaranteed to be prime.
- If *effort* = 2, additionally trial division by the numbers in the lists `Primes2` and `ProbablePrimes2` is done, and perfect powers are detected. Returned factors below 10^6 are guaranteed to be prime.
- If *effort* = 3, additionally `FactorsRho` (Pollard's Rho) with `RhoTrials` = 256 is used.
- If *effort* = 4, as above, but `RhoTrials` = 2048.
- If *effort* = 5, as above, but `RhoTrials` = 8192. Returned factors below 10^{12} are guaranteed to be prime, and all prime factors below 10^6 are guaranteed to be found.
- If *effort* = 6 and the package FactInt is loaded, in addition to the above quite a number of special cases are handled.
- If *effort* = 7 and the package FactInt is loaded, the only thing which is not attempted to obtain a full factorization into Baillie-Pomerance-Selfridge-Wagstaff pseudoprimes is the application of the MPQS to a remaining composite with more than 50 decimal digits.

Increasing the value of the argument *effort* by one usually results in an increase of the runtime requirements by a factor of (very roughly!) 3 to 10. (Also see `CheapFactorsInt` (**EDIM: CheapFactorsInt**)).

Example

```
gap> List([0..5],i->PartialFactorization(97^35-1,i));
[ [ 2, 2, 2, 2, 2, 3, 11, 31, 43,
      24463389590595215209018263651689171101059728242295553190029650 29 ],
  [ 2, 2, 2, 2, 2, 3, 11, 31, 43, 967,
      25298231220884400422976487747351779835635706558733767518127 87 ],
  [ 2, 2, 2, 2, 2, 3, 11, 31, 43, 967,
      25298231220884400422976487747351779835635706558733767518127 87 ],
  [ 2, 2, 2, 2, 2, 3, 11, 31, 43, 967, 39761, 262321,
      24254917395032592185976942143565315344561696291422 7 ],
  [ 2, 2, 2, 2, 2, 3, 11, 31, 43, 967, 39761, 262321, 687121,
      35299339410427846312333551359317085847415078 7 ],
  [ 2, 2, 2, 2, 2, 3, 11, 31, 43, 967, 39761, 262321, 687121,
      20241187, 504769301, 345491738434515746299113615 01 ] ]
```

14.4.10 PrintFactorsInt

▷ `PrintFactorsInt(`*n*`)` (function)

prints the prime factorization of the integer *n* in human-readable form.

Example

```
gap> PrintFactorsInt( Factorial( 7 ) ); Print( "\n" );
2^4*3^2*5*7
```

14.4.11 PrimePowersInt

▷ `PrimePowersInt(n)` (function)

returns the prime factorization of the integer `n` as a list $[p_1, e_1, \ldots, p_k, e_k]$ with $n = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k}$.

Example

```
gap> PrimePowersInt( Factorial( 7 ) );
[ 2, 4, 3, 2, 5, 1, 7, 1 ]
```

14.4.12 DivisorsInt

▷ `DivisorsInt(n)` (function)

`DivisorsInt` returns a list of all divisors of the integer `n`. The list is sorted, so that it starts with 1 and ends with `n`. We define that `DivisorsInt( -n ) = DivisorsInt( n )`.

Since the set of divisors of 0 is infinite calling `DivisorsInt( 0 )` causes an error.

`DivisorsInt` may call `FactorsInt` (14.4.7) to obtain the prime factors. `Sigma` (15.5.1) and `Tau` (15.5.2) compute the sum and the number of positive divisors, respectively.

Example

```
gap> DivisorsInt( 1 ); DivisorsInt( 20 ); DivisorsInt( 541 );
[ 1 ]
[ 1, 2, 4, 5, 10, 20 ]
[ 1, 541 ]
```

14.5 Residue Class Rings

`ZmodnZ` (14.5.2) returns a residue class ring of `Integers` (14) modulo an ideal. These residue class rings are rings, thus all operations for rings (see Chapter 56) apply. See also Chapters 59 and 15.

14.5.1 \mod (for residue class rings)

▷ `\mod(r/s, n)` (operation)

If `r`, `s` and `n` are integers, `r / s` as a reduced fraction is `p/q`, where `q` and `n` are coprime, then `r / s mod n` is defined to be the product of `p` and the inverse of `q` modulo `n`. See Section 4.13 for more details and definitions.

With the above definition, `4 / 6 mod 32` equals `2 / 3 mod 32` and hence exists (and is equal to 22), despite the fact that 6 has no inverse modulo 32.

14.5.2 ZmodnZ

▷ `ZmodnZ(n)` (function)

▷ `ZmodpZ(p)` (function)

▷ `ZmodpZNC(p)` (function)

`ZmodnZ` returns a ring *R* isomorphic to the residue class ring of the integers modulo the positive integer `n`. The element corresponding to the residue class of the integer *i* in this ring can be obtained

by `i * One( R )`, and a representative of the residue class corresponding to the element $x \in R$ can be computed by `Int`(x).

`ZmodnZ( n )` is equal to `Integers mod n`.

`ZmodpZ` does the same if the argument `p` is a prime integer, additionally the result is a field. `ZmodpZNC` omits the check whether `p` is a prime.

Each ring returned by these functions contains the whole family of its elements if `n` is not a prime, and is embedded into the family of finite field elements of characteristic `n` if `n` is a prime.

14.5.3 ZmodnZObj (for a residue class family and integer)

▷ `ZmodnZObj(`*`Fam`*`, r)` (operation)
▷ `ZmodnZObj(r, n)` (operation)

If the first argument is a residue class family *Fam* then `ZmodnZObj` returns the element in *Fam* whose coset is represented by the integer `r`.

If the two arguments are an integer `r` and a positive integer `n` then `ZmodnZObj` returns the element in `ZmodnZ( n )` (see `ZmodnZ` (14.5.2)) whose coset is represented by the integer `r`.

Example

```
gap> r:= ZmodnZ(15);
(Integers mod 15)
gap> fam:=ElementsFamily(FamilyObj(r));;
gap> a:= ZmodnZObj(fam,9);
ZmodnZObj( 9, 15 )
gap> a+a;
ZmodnZObj( 3, 15 )
gap> Int(a+a);
3
```

14.5.4 IsZmodnZObj

▷ `IsZmodnZObj(`*`obj`*`)` (Category)
▷ `IsZmodnZObjNonprime(`*`obj`*`)` (Category)
▷ `IsZmodpZObj(`*`obj`*`)` (Category)
▷ `IsZmodpZObjSmall(`*`obj`*`)` (Category)
▷ `IsZmodpZObjLarge(`*`obj`*`)` (Category)

The elements in the rings Z/nZ are in the category `IsZmodnZObj`. If n is a prime then the elements are of course also in the category `IsFFE` (59.1.1), otherwise they are in `IsZmodnZObjNonprime`. `IsZmodpZObj` is an abbreviation of `IsZmodnZObj and IsFFE`. This category is the disjoint union of `IsZmodpZObjSmall` and `IsZmodpZObjLarge`, the former containing all elements with n at most `MAXSIZE_GF_INTERNAL`.

The reasons to distinguish the prime case from the nonprime case are

- that objects in `IsZmodnZObjNonprime` have an external representation (namely the residue in the range $[0,1,\ldots,n-1]$),
- that the comparison of elements can be defined as comparison of the residues, and

- that the elements lie in a family of type `IsZmodnZObjNonprimeFamily` (note that for prime *n*, the family must be an `IsFFEFamily`).

The reasons to distinguish the small and the large case are that for small *n* the elements must be compatible with the internal representation of finite field elements, whereas we are free to define comparison as comparison of residues for large *n*.

Note that we *cannot* claim that every finite field element of degree 1 is in `IsZmodnZObj`, since finite field elements in internal representation may not know that they lie in the prime field.

14.6 Check Digits

14.6.1 CheckDigitISBN

▷ `CheckDigitISBN(`*n*`)` (function)
▷ `CheckDigitISBN13(`*n*`)` (function)
▷ `CheckDigitPostalMoneyOrder(`*n*`)` (function)
▷ `CheckDigitUPC(`*n*`)` (function)

These functions can be used to compute, or check, check digits for some everyday items. In each case what is submitted as input is either the number with check digit (in which case the function returns `true` or `false`), or the number without check digit (in which case the function returns the missing check digit). The number can be specified as integer, as string (for example in case of leading zeros) or as a sequence of arguments, each representing a single digit. The check digits tested are the 10-digit ISBN (International Standard Book Number) using `CheckDigitISBN` (since arithmetic is module 11, a digit 11 is represented by an X); the newer 13-digit ISBN-13 using `CheckDigitISBN13`; the numbers of 11-digit US postal money orders using `CheckDigitPostalMoneyOrder`; and the 12-digit UPC bar code found on groceries using `CheckDigitUPC`.

Example

```
gap> CheckDigitISBN("052166103");
Check Digit is 'X'
'X'
gap> CheckDigitISBN("052166103X");
Checksum test satisfied
true
gap> CheckDigitISBN(0,5,2,1,6,6,1,0,3,1);
Checksum test failed
false
gap> CheckDigitISBN(0,5,2,1,6,6,1,0,3,'X'); # note single quotes!
Checksum test satisfied
true
gap> CheckDigitISBN13("9781420094527");
Checksum test satisfied
true
gap> CheckDigitUPC("07164183001");
Check Digit is 1
1
gap> CheckDigitPostalMoneyOrder(16786457155);
Checksum test satisfied
true
```

14.6.2 CheckDigitTestFunction

▷ CheckDigitTestFunction(*l, m, f*) (function)

This function creates check digit test functions such as CheckDigitISBN (14.6.1) for check digit schemes that use the inner products with a fixed vector modulo a number. The scheme creates will use strings of *l* digits (including the check digits), the check consists of taking the standard product of the vector of digits with the fixed vector *f* modulo *m*; the result needs to be 0. The function returns a function that then can be used for testing or determining check digits.

Example

```
gap> isbntest:=CheckDigitTestFunction(10,11,[1,2,3,4,5,6,7,8,9,-1]);
function( arg ) ... end
gap> isbntest("038794680");
Check Digit is 2
2
```

14.7 Random Sources

GAP provides Random (30.7.1) methods for many collections of objects. On a lower level these methods use *random sources* which provide random integers and random choices from lists.

14.7.1 IsRandomSource

▷ IsRandomSource(*obj*) (Category)

This is the category of random source objects which are defined to have, for an object *rs* in this category, methods available for the following operations which are explained in more detail below: Random(*rs*, *list*) giving a random element of a list, Random(*rs*, *low*, *high*) giving a random integer between *low* and *high* (inclusive), Init (14.7.3), State (14.7.3) and Reset (14.7.3).

Use RandomSource (14.7.5) to construct new random sources.

One idea behind providing several independent (pseudo) random sources is to make algorithms which use some sort of random choices deterministic. They can use their own new random source created with a fixed seed and so do exactly the same in different calls.

Random source objects lie in the family RandomSourcesFamily.

14.7.2 Random (for a list)

▷ Random(*rs, list*) (operation)
▷ Random(*rs, low, high*) (operation)

This operation returns a random element from list *list*, or an integer in the range from the given (possibly large) integers *low* to *high*, respectively. The choice should only depend on the random source *rs* and have no effect on other random sources.

14.7.3 State

▷ State(*rs*) (operation)
▷ Reset(*rs[, seed]*) (operation)

▷ Init(*prers[, seed]*) (operation)

These are the basic operations for which random sources (see IsRandomSource (14.7.1)) must have methods.

State should return a data structure which allows to recover the state of the random source such that a sequence of random calls using this random source can be reproduced. If a random source cannot be reset (say, it uses truly random physical data) then State should return fail.

Reset(*rs*, *seed*) resets the random source *rs* to a state described by *seed*, if the random source can be reset (otherwise it should do nothing). Here *seed* can be an output of State and then should reset to that state. Also, the methods should always allow integers as *seed*. Without the *seed* argument the default $seed = 1$ is used.

Init is the constructor of a random source, it gets an empty component object *prers* which has already the correct type and should fill in the actual data which are needed. Optionally, it should allow one to specify a *seed* for the initial state, as explained for Reset.

Most methods for Random (30.7.1) in the GAP library use the GlobalMersenneTwister (14.7.4) as random source. It can be reset into a known state as in the following example.

Example

```
gap> seed := State(GlobalMersenneTwister);;
gap> List([1..10],i->Random(Integers));
[ -1, -3, -2, 1, -2, -1, 0, 1, 0, 1 ]
gap> List([1..10],i->Random(Integers));
[ -1, 0, 2, 0, 4, -1, -3, 1, -4, -1 ]
gap> Reset(GlobalMersenneTwister, seed);;
gap> List([1..10],i->Random(Integers));
[ -1, -3, -2, 1, -2, -1, 0, 1, 0, 1 ]
```

14.7.4 IsMersenneTwister

▷ IsMersenneTwister(*rs*) (Category)
▷ IsGAPRandomSource(*rs*) (Category)
▷ IsGlobalRandomSource(*rs*) (Category)
▷ GlobalMersenneTwister (global variable)
▷ GlobalRandomSource (global variable)

Currently, the GAP library provides three types of random sources, distinguished by the three listed categories.

IsMersenneTwister are random sources which use a fast random generator of 32 bit numbers, called the Mersenne twister. The pseudo random sequence has a period of $2^{19937} - 1$ and the numbers have a 623-dimensional equidistribution. For more details and the origin of the code used in the GAP kernel, see: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

Use the Mersenne twister if possible, in particular for generating many large random integers.

There is also a predefined global random source GlobalMersenneTwister which is used by most of the library methods for Random (30.7.1).

IsGAPRandomSource uses the same number generator as IsGlobalRandomSource, but you can create several of these random sources which generate their random numbers independently of all other random sources.

IsGlobalRandomSource gives access to the *classical* global random generator which was used by GAP in former releases. You do not need to construct new random sources of this

kind which would all use the same global data structure. Just use the existing random source `GlobalRandomSource`. This uses the additive random number generator described in [Knu98] (Algorithm A in 3.2.2 with lag 30).

14.7.5 RandomSource

▷ `RandomSource(`*`cat[, seed]`*`)` (operation)

This operation is used to create new random sources. The first argument *`cat`* is the category describing the type of the random generator, an optional *`seed`* which can be an integer or a type specific data structure can be given to specify the initial state.

Example

```
gap> rs1 := RandomSource(IsMersenneTwister);
<RandomSource in IsMersenneTwister>
gap> state1 := State(rs1);;
gap> l1 := List([1..10000], i-> Random(rs1, [1..6]));;
gap> rs2 := RandomSource(IsMersenneTwister);;
gap> l2 := List([1..10000], i-> Random(rs2, [1..6]));;
gap> l1 = l2;
true
gap> l1 = List([1..10000], i-> Random(rs1, [1..6]));
false
gap> n := Random(rs1, 1, 2^220);
1598617776705343302477918831699169150767442847525442557699717518961
```

Chapter 15

Number Theory

GAP provides a couple of elementary number theoretic functions. Most of these deal with the group of integers coprime to m, called the *prime residue group*. The order of this group is $\phi(m)$ (see `Phi` (15.2.2)), and $\lambda(m)$ (see `Lambda` (15.2.3)) is its exponent. This group is cyclic if and only if m is 2, 4, an odd prime power p^n, or twice an odd prime power $2p^n$. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called *primitive roots* (see `PrimitiveRootMod` (15.3.3)).

Note that neither the arguments nor the return values of the functions listed below are groups or group elements in the sense of GAP. The arguments are simply integers.

15.1 InfoNumtheor (Info Class)

15.1.1 InfoNumtheor

▷ `InfoNumtheor` (info class)

`InfoNumtheor` is the info class (see 7.4) for the functions in the number theory chapter.

15.2 Prime Residues

15.2.1 PrimeResidues

▷ `PrimeResidues(`*m*`)` (function)

`PrimeResidues` returns the set of integers from the range `[ 0 .. Abs( `*m*` )-1 ]` that are coprime to the integer *m*.

`Abs(`*m*`)` must be less than 2^{28}, otherwise the set would probably be too large anyhow.

Example

```
gap> PrimeResidues( 0 );  PrimeResidues( 1 );  PrimeResidues( 20 );
[  ]
[ 0 ]
[ 1, 3, 7, 9, 11, 13, 17, 19 ]
```

15.2.2 Phi

▷ `Phi(m)` (operation)

`Phi` returns the number $\phi(m)$ of positive integers less than the positive integer `m` that are coprime to `m`.

Suppose that $m = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. Then $\phi(m)$ is $p_1^{e_1-1}(p_1-1)p_2^{e_2-1}(p_2-1)\cdots p_k^{e_k-1}(p_k-1)$.

Example

```
gap> Phi( 12 );
4
gap> Phi( 2^13-1 );  # this proves that 2^(13)-1 is a prime
8190
gap> Phi( 2^15-1 );
27000
```

15.2.3 Lambda

▷ `Lambda(m)` (operation)

`Lambda` returns the exponent $\lambda(m)$ of the group of prime residues modulo the integer `m`.

$\lambda(m)$ is the smallest positive integer l such that for every a relatively prime to `m` we have $a^l \equiv 1 \pmod{m}$. Fermat's theorem asserts $a^{\phi(m)} \equiv 1 \pmod{m}$; thus $\lambda(m)$ divides $\phi(m)$ (see `Phi` (15.2.2)).

Carmichael's theorem states that λ can be computed as follows: $\lambda(2) = 1$, $\lambda(4) = 2$ and $\lambda(2^e) = 2^{e-2}$ if $3 \leq e$, $\lambda(p^e) = (p-1)p^{e-1}$ (i.e. $\phi(m)$) if p is an odd prime and $\lambda(m*n) =$`Lcm`$(\lambda(m), \lambda(n))$ if m, n are coprime.

Composites for which $\lambda(m)$ divides $m-1$ are called Carmichaels. If $6k+1$, $12k+1$ and $18k+1$ are primes their product is such a number. There are only 1547 Carmichaels below 10^{10} but 455052511 primes.

Example

```
gap> Lambda( 10 );
4
gap> Lambda( 30 );
4
gap> Lambda( 561 );  # 561 is the smallest Carmichael number
80
```

15.2.4 GeneratorsPrimeResidues

▷ `GeneratorsPrimeResidues(n)` (function)

Let `n` be a positive integer. `GeneratorsPrimeResidues` returns a description of generators of the group of prime residues modulo `n`. The return value is a record with components

`primes`:
: a list of the prime factors of `n`,

`exponents`:
: a list of the exponents of these primes in the factorization of `n`, and

`generators`:

a list describing generators of the group of prime residues; for the prime factor 2, either a primitive root or a list of two generators is stored, for each other prime factor of n, a primitive root is stored.

Example

```
gap> GeneratorsPrimeResidues( 1 );
rec( exponents := [  ], generators := [  ], primes := [  ] )
gap> GeneratorsPrimeResidues( 4*3 );
rec( exponents := [ 2, 1 ], generators := [ 7, 5 ],
  primes := [ 2, 3 ] )
gap> GeneratorsPrimeResidues( 8*9*5 );
rec( exponents := [ 3, 2, 1 ],
  generators := [ [ 271, 181 ], 281, 217 ], primes := [ 2, 3, 5 ] )
```

15.3 Primitive Roots and Discrete Logarithms

15.3.1 OrderMod

▷ `OrderMod(n, m)` (function)

`OrderMod` returns the multiplicative order of the integer n modulo the positive integer m. If n and m are not coprime the order of n is not defined and `OrderMod` will return `0`.

If n and m are relatively prime the multiplicative order of n modulo m is the smallest positive integer i such that $n^i \equiv 1 \pmod{m}$. If the group of prime residues modulo m is cyclic then each element of maximal order is called a primitive root modulo m (see `IsPrimitiveRootMod` (15.3.4)).

`OrderMod` usually spends most of its time factoring m and $\phi(m)$ (see `FactorsInt` (14.4.7)).

Example

```
gap> OrderMod( 2, 7 );
3
gap> OrderMod( 3, 7 );  # 3 is a primitive root modulo 7
6
```

15.3.2 LogMod

▷ `LogMod(n, r, m)` (function)
▷ `LogModShanks(n, r, m)` (function)

computes the discrete r-logarithm of the integer n modulo the integer m. It returns a number l such that $r^l \equiv n \pmod{m}$ if such a number exists. Otherwise `fail` is returned.

`LogModShanks` uses the Baby Step - Giant Step Method of Shanks (see for example [Coh93, section 5.4.1]) and in general requires more memory than a call to `LogMod`.

Example

```
gap> l:= LogMod( 2, 5, 7 );  5^l mod 7 = 2;
4
true
gap> LogMod( 1, 3, 3 );  LogMod( 2, 3, 3 );
0
fail
```

15.3.3 PrimitiveRootMod

▷ PrimitiveRootMod(*m[, start]*) (function)

PrimitiveRootMod returns the smallest primitive root modulo the positive integer *m* and fail if no such primitive root exists. If the optional second integer argument *start* is given PrimitiveRootMod returns the smallest primitive root that is strictly larger than *start*.

Example

```
gap> # largest primitive root for a prime less than 2000:
gap> PrimitiveRootMod( 409 );
21
gap> PrimitiveRootMod( 541, 2 );
10
gap> # 327 is the largest primitive root mod 337:
gap> PrimitiveRootMod( 337, 327 );
fail
gap> # there exists no primitive root modulo 30:
gap> PrimitiveRootMod( 30 );
fail
```

15.3.4 IsPrimitiveRootMod

▷ IsPrimitiveRootMod(*r, m*) (function)

IsPrimitiveRootMod returns true if the integer *r* is a primitive root modulo the positive integer *m*, and false otherwise. If *r* is less than 0 or larger than *m* it is replaced by its remainder.

Example

```
gap> IsPrimitiveRootMod( 2, 541 );
true
gap> IsPrimitiveRootMod( -539, 541 );  # same computation as above;
true
gap> IsPrimitiveRootMod( 4, 541 );
false
gap> ForAny( [1..29], r -> IsPrimitiveRootMod( r, 30 ) );
false
gap> # there is no a primitive root modulo 30
```

15.4 Roots Modulo Integers

15.4.1 Jacobi

▷ Jacobi(*n, m*) (function)

Jacobi returns the value of the *Kronecker-Jacobi symbol* $J(n,m)$ of the integer *n* modulo the integer *m*. It is defined as follows:

If n and m are not coprime then $J(n,m) = 0$. Furthermore, $J(n,1) = 1$ and $J(n,-1) = -1$ if $m < 0$ and $+1$ otherwise. And for odd n it is $J(n,2) = (-1)^k$ with $k = (n^2-1)/8$. For odd primes m which are coprime to n the Kronecker-Jacobi symbol has the same value as the Legendre symbol (see Legendre (15.4.2)).

For the general case suppose that $m = p_1 \cdot p_2 \cdots p_k$ is a product of -1 and of primes, not necessarily distinct, and that n is coprime to m. Then $J(n,m) = J(n,p_1) \cdot J(n,p_2) \cdots J(n,p_k)$.

Note that the Kronecker-Jacobi symbol coincides with the Jacobi symbol that is defined for odd m in many number theory books. For odd primes m and n coprime to m it coincides with the Legendre symbol.

`Jacobi` is very efficient, even for large values of `n` and `m`, it is about as fast as the Euclidean algorithm (see `Gcd` (56.7.1)).

Example

```
gap> Jacobi( 11, 35 );  # 9^2 = 11 mod 35
1
gap> # this is -1, thus there is no r such that r^2 = 6 mod 35
gap> Jacobi( 6, 35 );
-1
gap> # this is 1 even though there is no r with r^2 = 3 mod 35
gap> Jacobi( 3, 35 );
1
```

15.4.2 Legendre

▷ `Legendre(n, m)` (function)

`Legendre` returns the value of the *Legendre symbol* of the integer `n` modulo the positive integer `m`.

The value of the Legendre symbol $L(n/m)$ is 1 if n is a *quadratic residue* modulo m, i.e., if there exists an integer r such that $r^2 \equiv n \pmod{m}$ and -1 otherwise.

If a root of `n` exists it can be found by `RootMod` (15.4.3).

While the value of the Legendre symbol usually is only defined for `m` a prime, we have extended the definition to include composite moduli too. The Jacobi symbol (see `Jacobi` (15.4.1)) is another generalization of the Legendre symbol for composite moduli that is much cheaper to compute, because it does not need the factorization of `m` (see `FactorsInt` (14.4.7)).

A description of the Jacobi symbol, the Legendre symbol, and related topics can be found in [Bak84].

Example

```
gap> Legendre( 5, 11 );  # 4^2 = 5 mod 11
1
gap> # this is -1, thus there is no r such that r^2 = 6 mod 11
gap> Legendre( 6, 11 );
-1
gap> # this is -1, thus there is no r such that r^2 = 3 mod 35
gap> Legendre( 3, 35 );
-1
```

15.4.3 RootMod

▷ `RootMod(n[, k], m)` (function)

`RootMod` computes a `k`th root of the integer `n` modulo the positive integer `m`, i.e., a r such that $r^k \equiv n \pmod{m}$. If no such root exists `RootMod` returns `fail`. If only the arguments `n` and `m` are given, the default value for `k` is 2.

A square root of `n` exists only if `Legendre(n,m) = 1` (see `Legendre` (15.4.2)). If `m` has r different prime factors then there are 2^r different roots of `n` mod `m`. It is unspecified which one `RootMod` returns. You can, however, use `RootsMod` (15.4.4) to compute the full set of roots.

`RootMod` is efficient even for large values of `m`, in fact the most time is usually spent factoring `m` (see `FactorsInt` (14.4.7)).

Example

```
gap> # note 'RootMod' does not return 8 in this case but -8:
gap> RootMod( 64, 1009 );
1001
gap> RootMod( 64, 3, 1009 );
518
gap> RootMod( 64, 5, 1009 );
656
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),
>       x -> x mod 1009 );  # set of all square roots of 64 mod 1009
[ 1001, 8 ]
```

15.4.4 RootsMod

▷ `RootsMod(n[, k], m)` (function)

`RootsMod` computes the set of `k`th roots of the integer `n` modulo the positive integer `m`, i.e., the list of all r such that $r^k \equiv n \pmod{m}$. If only the arguments `n` and `m` are given, the default value for `k` is 2.

Example

```
gap> RootsMod( 1, 7*31 );  # the same as 'RootsUnityMod( 7*31 )'
[ 1, 92, 125, 216 ]
gap> RootsMod( 7, 7*31 );
[ 21, 196 ]
gap> RootsMod( 5, 7*31 );
[  ]
gap> RootsMod( 1, 5, 7*31 );
[ 1, 8, 64, 78, 190 ]
```

15.4.5 RootsUnityMod

▷ `RootsUnityMod([k, ]m)` (function)

`RootsUnityMod` returns the set of k-th roots of unity modulo the positive integer `m`, i.e., the list of all solutions r of $r^k \equiv n \pmod{m}$. If only the argument `m` is given, the default value for `k` is 2.

In general there are k^n such roots if the modulus `m` has n different prime factors p such that $p \equiv 1 \pmod{k}$. If k^2 divides `m` then there are k^{n+1} such roots; and especially if $k = 2$ and 8 divides `m` there are 2^{n+2} such roots.

In the current implementation k must be a prime.

Example

```
gap> RootsUnityMod( 7*31 );  RootsUnityMod( 3, 7*31 );
[ 1, 92, 125, 216 ]
[ 1, 25, 32, 36, 67, 149, 156, 191, 211 ]
gap> RootsUnityMod( 5, 7*31 );
[ 1, 8, 64, 78, 190 ]
```

```
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),
>          x -> x mod 1009 );  # set of all square roots of 64 mod 1009
[ 1001, 8 ]
```

15.5 Multiplicative Arithmetic Functions

15.5.1 Sigma

▷ `Sigma(n)` (operation)

`Sigma` returns the sum of the positive divisors of the nonzero integer `n`.

`Sigma` is a multiplicative arithmetic function, i.e., if n and m are relatively prime we have that $\sigma(n \cdot m) = \sigma(n)\sigma(m)$.

Together with the formula $\sigma(p^k) = (p^{k+1}-1)/(p-1)$ this allows us to compute $\sigma(n)$.

Integers `n` for which $\sigma(n) = 2n$ are called perfect. Even perfect integers are exactly of the form $2^{n-1}(2^n-1)$ where 2^n-1 is prime. Primes of the form 2^n-1 are called *Mersenne primes*, and 42 among the known Mersenne primes are obtained for n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583 and 25964951. Please find more up to date information about Mersenne primes at `http://www.mersenne.org`. It is not known whether odd perfect integers exist, however [BC89] show that any such integer must have at least 300 decimal digits.

`Sigma` usually spends most of its time factoring `n` (see `FactorsInt` (14.4.7)).

Example

```
gap> Sigma( 1 );
1
gap> Sigma( 1009 );  # 1009 is a prime
1010
gap> Sigma( 8128 ) = 2*8128;  # 8128 is a perfect number
true
```

15.5.2 Tau

▷ `Tau(n)` (operation)

`Tau` returns the number of the positive divisors of the nonzero integer `n`.

`Tau` is a multiplicative arithmetic function, i.e., if n and m are relative prime we have $\tau(n \cdot m) = \tau(n)\tau(m)$. Together with the formula $\tau(p^k) = k+1$ this allows us to compute $\tau(n)$.

`Tau` usually spends most of its time factoring `n` (see `FactorsInt` (14.4.7)).

Example

```
gap> Tau( 1 );
1
gap> Tau( 1013 );  # thus 1013 is a prime
2
gap> Tau( 8128 );
14
gap> # result is odd if and only if argument is a perfect square:
```

```
gap> Tau( 36 );
9
```

15.5.3 MoebiusMu

▷ `MoebiusMu(n)` (function)

`MoebiusMu` computes the value of Moebius inversion function for the nonzero integer `n`. This is 0 for integers which are not squarefree, i.e., which are divided by a square r^2. Otherwise it is 1 if `n` has a even number and -1 if `n` has an odd number of prime factors.

The importance of μ stems from the so called inversion formula. Suppose f is a multiplicative arithmetic function defined on the positive integers and let $g(n) = \sum_{d|n} f(d)$. Then $f(n) = \sum_{d|n} \mu(d)g(n/d)$. As a special case we have $\phi(n) = \sum_{d|n} \mu(d)n/d$ since $n = \sum_{d|n} \phi(d)$ (see `Phi` (15.2.2)).

`MoebiusMu` usually spends all of its time factoring `n` (see `FactorsInt` (14.4.7)).

Example

```
gap> MoebiusMu( 60 );  MoebiusMu( 61 );  MoebiusMu( 62 );
0
-1
1
```

15.6 Continued Fractions

15.6.1 ContinuedFractionExpansionOfRoot

▷ `ContinuedFractionExpansionOfRoot(f, n)` (function)

The first `n` terms of the continued fraction expansion of the only positive real root of the polynomial `f` with integer coefficients. The leading coefficient of `f` must be positive and the value of `f` at 0 must be negative. If the degree of `f` is 2 and `n` = 0, the function computes one period of the continued fraction expansion of the root in question. Anything may happen if `f` has three or more positive real roots.

Example

```
gap> x := Indeterminate(Integers);;
gap> ContinuedFractionExpansionOfRoot(x^2-7,20);
[ 2, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1 ]
gap> ContinuedFractionExpansionOfRoot(x^2-7,0);
[ 2, 1, 1, 1, 4 ]
gap> ContinuedFractionExpansionOfRoot(x^3-2,20);
[ 1, 3, 1, 5, 1, 1, 4, 1, 1, 8, 1, 14, 1, 10, 2, 1, 4, 12, 2, 3 ]
gap> ContinuedFractionExpansionOfRoot(x^5-x-1,50);
[ 1, 5, 1, 42, 1, 3, 24, 2, 2, 1, 16, 1, 11, 1, 1, 2, 31, 1, 12, 5,
  1, 7, 11, 1, 4, 1, 4, 2, 2, 3, 4, 2, 1, 1, 11, 1, 41, 12, 1, 8, 1,
  1, 1, 1, 1, 9, 2, 1, 5, 4 ]
```

15.6.2 ContinuedFractionApproximationOfRoot

▷ `ContinuedFractionApproximationOfRoot(f, n)` (function)

The `n`th continued fraction approximation of the only positive real root of the polynomial `f` with integer coefficients. The leading coefficient of `f` must be positive and the value of `f` at 0 must be negative. Anything may happen if `f` has three or more positive real roots.

Example

```
gap> ContinuedFractionApproximationOfRoot(x^2-2,10);
3363/2378
gap> 3363^2-2*2378^2;
1
gap> z := ContinuedFractionApproximationOfRoot(x^5-x-1,20);
499898783527/428250732317
gap> z^5-z-1;
486192462527432755459620441970617283/
144042473823198424216973575588057090311169878262426312613 57
```

15.7 Miscellaneous

15.7.1 TwoSquares

▷ `TwoSquares(n)` (function)

`TwoSquares` returns a list of two integers $x \leq y$ such that the sum of the squares of x and y is equal to the nonnegative integer `n`, i.e., $n = x^2 + y^2$. If no such representation exists `TwoSquares` will return `fail`. `TwoSquares` will return a representation for which the gcd of x and y is as small as possible. It is not specified which representation `TwoSquares` returns if there is more than one.

Let a be the product of all maximal powers of primes of the form $4k+3$ dividing `n`. A representation of `n` as a sum of two squares exists if and only if a is a perfect square. Let b be the maximal power of 2 dividing `n` or its half, whichever is a perfect square. Then the minimal possible gcd of x and y is the square root c of $a \cdot b$. The number of different minimal representation with $x \leq y$ is 2^{l-1}, where l is the number of different prime factors of the form $4k+1$ of `n`.

The algorithm first finds a square root r of -1 modulo $n/(a \cdot b)$, which must exist, and applies the Euclidean algorithm to r and `n`. The first residues in the sequence that are smaller than $\sqrt{n/(a \cdot b)}$ times c are a possible pair x and y.

Better descriptions of the algorithm and related topics can be found in [Wag90] and [Zag90].

Example

```
gap> TwoSquares( 5 );
[ 1, 2 ]
gap> TwoSquares( 11 );  # there is no representation
fail
gap> TwoSquares( 16 );
[ 0, 4 ]
gap> # 3 is the minimal possible gcd because 9 divides 45:
gap> TwoSquares( 45 );
[ 3, 6 ]
gap> # it is not [5,10] because their gcd is not minimal:
gap> TwoSquares( 125 );
[ 2, 11 ]
gap> # [10,11] would be the other possible representation:
gap> TwoSquares( 13*17 );
[ 5, 14 ]
```

```
gap> TwoSquares( 848654483879497562821 );  # argument is prime
[ 6305894639, 28440994650 ]
```

Chapter 16

Combinatorics

This chapter describes functions that deal with combinatorics. We mainly concentrate on two areas. One is about *selections*, that is the ways one can select elements from a set. The other is about *partitions*, that is the ways one can partition a set into the union of pairwise disjoint subsets.

16.1 Combinatorial Numbers

16.1.1 Factorial

▷ `Factorial(n)` (function)

returns the *factorial* $n!$ of the positive integer `n`, which is defined as the product $1 \cdot 2 \cdot 3 \cdots n$.

$n!$ is the number of permutations of a set of n elements. $1/n!$ is the coefficient of x^n in the formal series $\exp(x)$, which is the generating function for factorial.

Example

```
gap> List( [0..10], Factorial );
[ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800 ]
gap> Factorial( 30 );
265252859812191058636308480000000
```

`PermutationsList` (16.2.12) computes the set of all permutations of a list.

16.1.2 Binomial

▷ `Binomial(n, k)` (function)

returns the *binomial coefficient* $\binom{n}{k}$ of integers `n` and `k`, which is defined as $n!/(k!(n-k)!)$ (see `Factorial` (16.1.1)). We define $\binom{0}{0} = 1$, $\binom{n}{k} = 0$ if $k < 0$ or $n < k$, and $\binom{n}{k} = (-1)^k \binom{-n+k-1}{k}$ if $n < 0$, which is consistent with the equivalent definition $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

$\binom{n}{k}$ is the number of combinations with k elements, i.e., the number of subsets with k elements, of a set with n elements. $\binom{n}{k}$ is the coefficient of the term x^k of the polynomial $(x+1)^n$, which is the generating function for $\binom{n}{\cdot}$, hence the name.

Example

```
gap> # Knuth calls this the trademark of Binomial:
gap> List( [0..4], k->Binomial( 4, k ) );
```

```
[ 1, 4, 6, 4, 1 ]
gap> List( [0..6], n->List( [0..6], k->Binomial( n, k ) ) );;
gap> # the lower triangle is called Pascal's triangle:
gap> PrintArray( last );
[ [   1,   0,   0,   0,   0,   0,   0 ],
  [   1,   1,   0,   0,   0,   0,   0 ],
  [   1,   2,   1,   0,   0,   0,   0 ],
  [   1,   3,   3,   1,   0,   0,   0 ],
  [   1,   4,   6,   4,   1,   0,   0 ],
  [   1,   5,  10,  10,   5,   1,   0 ],
  [   1,   6,  15,  20,  15,   6,   1 ] ]
gap> Binomial( 50, 10 );
10272278170
```

`NrCombinations` (16.2.3) is the generalization of `Binomial` for multisets. `Combinations` (16.2.1) computes the set of all combinations of a multiset.

16.1.3 Bell

▷ `Bell(n)` (function)

returns the *Bell number* $B(n)$. The Bell numbers are defined by $B(0) = 1$ and the recurrence $B(n+1) = \sum_{k=0}^{n} \binom{n}{k} B(k)$.

$B(n)$ is the number of ways to partition a set of `n` elements into pairwise disjoint nonempty subsets (see `PartitionsSet` (16.2.16)). This implies of course that $B(n) = \sum_{k=0}^{n} S_2(n,k)$ (see `Stirling2` (16.1.6)). $B(n)/n!$ is the coefficient of x^n in the formal series $\exp(\exp(x)-1)$, which is the generating function for $B(n)$.

Example

```
gap> List( [0..6], n -> Bell( n ) );
[ 1, 1, 2, 5, 15, 52, 203 ]
gap> Bell( 14 );
190899322
```

16.1.4 Bernoulli

▷ `Bernoulli(n)` (function)

returns the `n`-th *Bernoulli number* B_n, which is defined by $B_0 = 1$ and $B_n = -\sum_{k=0}^{n-1} \binom{n+1}{k} B_k/(n+1)$.

$B_n/n!$ is the coefficient of x^n in the power series of $x/(\exp(x)-1)$. Except for $B_1 = -1/2$ the Bernoulli numbers for odd indices are zero.

Example

```
gap> Bernoulli( 4 );
-1/30
gap> Bernoulli( 10 );
5/66
gap> Bernoulli( 12 );  # there is no simple pattern in Bernoulli numbers
-691/2730
gap> Bernoulli( 50 );  # and they grow fairly fast
495057205241079648212477525/66
```

16.1.5 Stirling1

▷ `Stirling1(n, k)` (function)

returns the *Stirling number of the first kind* $S_1(n,k)$ of the integers `n` and `k`. Stirling numbers of the first kind are defined by $S_1(0,0) = 1$, $S_1(n,0) = S_1(0,k) = 0$ if $n,k \neq 0$ and the recurrence $S_1(n,k) = (n-1)S_1(n-1,k) + S_1(n-1,k-1)$.

$S_1(n,k)$ is the number of permutations of `n` points with `k` cycles. Stirling numbers of the first kind appear as coefficients in the series $n!\binom{x}{n} = \sum_{k=0}^{n} S_1(n,k)x^k$ which is the generating function for Stirling numbers of the first kind. Note the similarity to $x^n = \sum_{k=0}^{n} S_2(n,k)k!\binom{x}{k}$ (see `Stirling2` (16.1.6)). Also the definition of S_1 implies $S_1(n,k) = S_2(-k,-n)$ if $n,k < 0$. There are many formulae relating Stirling numbers of the first kind to Stirling numbers of the second kind, Bell numbers, and Binomial coefficients.

Example

```
gap> # Knuth calls this the trademark of S_1:
gap> List( [0..4], k -> Stirling1( 4, k ) );
[ 0, 6, 11, 6, 1 ]
gap> List( [0..6], n->List( [0..6], k->Stirling1( n, k ) ) );;
gap> # note the similarity with Pascal's triangle for Binomial numbers
gap> PrintArray( last );
[ [    1,    0,    0,    0,    0,    0,    0 ],
  [    0,    1,    0,    0,    0,    0,    0 ],
  [    0,    1,    1,    0,    0,    0,    0 ],
  [    0,    2,    3,    1,    0,    0,    0 ],
  [    0,    6,   11,    6,    1,    0,    0 ],
  [    0,   24,   50,   35,   10,    1,    0 ],
  [    0,  120,  274,  225,   85,   15,    1 ] ]
gap> Stirling1(50,10);
1016230209263674900590437971193099440434055053805036656273653 76
```

16.1.6 Stirling2

▷ `Stirling2(n, k)` (function)

returns the *Stirling number of the second kind* $S_2(n,k)$ of the integers `n` and `k`. Stirling numbers of the second kind are defined by $S_2(0,0) = 1$, $S_2(n,0) = S_2(0,k) = 0$ if $n,k \neq 0$ and the recurrence $S_2(n,k) = kS_2(n-1,k) + S_2(n-1,k-1)$.

$S_2(n,k)$ is the number of ways to partition a set of `n` elements into `k` pairwise disjoint nonempty subsets (see `PartitionsSet` (16.2.16)). Stirling numbers of the second kind appear as coefficients in the expansion of $x^n = \sum_{k=0}^{n} S_2(n,k)k!\binom{x}{k}$. Note the similarity to $n!\binom{x}{n} = \sum_{k=0}^{n} S_1(n,k)x^k$ (see `Stirling1` (16.1.5)). Also the definition of S_2 implies $S_2(n,k) = S_1(-k,-n)$ if $n,k < 0$. There are many formulae relating Stirling numbers of the second kind to Stirling numbers of the first kind, Bell numbers, and Binomial coefficients.

Example

```
gap> # Knuth calls this the trademark of S_2:
gap> List( [0..4], k->Stirling2( 4, k ) );
[ 0, 1, 7, 6, 1 ]
gap> List( [0..6], n->List( [0..6], k->Stirling2( n, k ) ) );;
gap> # note the similarity with Pascal's triangle for Binomial numbers
gap> PrintArray( last );
```

```
[ [   1,   0,   0,   0,   0,   0,   0 ],
  [   0,   1,   0,   0,   0,   0,   0 ],
  [   0,   1,   1,   0,   0,   0,   0 ],
  [   0,   1,   3,   1,   0,   0,   0 ],
  [   0,   1,   7,   6,   1,   0,   0 ],
  [   0,   1,  15,  25,  10,   1,   0 ],
  [   0,   1,  31,  90,  65,  15,   1 ] ]
gap> Stirling2( 50, 10 );
2615471651586288129201277739657799378172701l
```

16.2 Combinations, Arrangements and Tuples

16.2.1 Combinations

▷ `Combinations(`*`mset[, k]`*`)` (function)

returns the set of all combinations of the multiset *mset* (a list of objects which may contain the same object several times) with *k* elements; if *k* is not given it returns all combinations of *mset*.

A *combination* of *mset* is an unordered selection without repetitions and is represented by a sorted sublist of *mset*. If *mset* is a proper set, there are $\binom{|mset|}{k}$ (see `Binomial` (16.1.2)) combinations with *k* elements, and the set of all combinations is just the *power set* of *mset*, which contains all *subsets* of *mset* and has cardinality $2^{|mset|}$.

To loop over combinations of a larger multiset use `IteratorOfCombinations` (16.2.2) which produces combinations one by one and may save a lot of memory. Another memory efficient representation of the list of all combinations is provided by `EnumeratorOfCombinations` (16.2.2).

16.2.2 Iterator and enumerator of combinations

▷ `IteratorOfCombinations(`*`mset[, k]`*`)` (function)
▷ `EnumeratorOfCombinations(`*`mset`*`)` (function)

`IteratorOfCombinations` returns an `Iterator` (30.8.1) for combinations (see `Combinations` (16.2.1)) of the given multiset *mset*. If a non-negative integer *k* is given as second argument then only the combinations with *k* entries are produced, otherwise all combinations.

`EnumeratorOfCombinations` returns an `Enumerator` (30.3.2) of the given multiset *mset*. Currently only a variant without second argument *k* is implemented.

The ordering of combinations from these functions can be different and also different from the list returned by `Combinations` (16.2.1).

Example

```
gap> m:=[1..15];; Add(m, 15);
gap> NrCombinations(m);
49152
gap> i := 0;; for c in Combinations(m) do i := i+1; od;
gap> i;
49152
gap> cm := EnumeratorOfCombinations(m);;
gap> cm[1000];
[ 1, 2, 3, 6, 7, 8, 9, 10 ]
```

```
gap> Position(cm, [1,13,15,15]);
36866
```

16.2.3 NrCombinations

▷ NrCombinations(*mset[, k]*) (function)

returns the number of Combinations(*mset*,*k*).

Example

```
gap> Combinations( [1,2,2,3] );
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 2 ], [ 1, 2, 2, 3 ], [ 1, 2, 3 ],
  [ 1, 3 ], [ 2 ], [ 2, 2 ], [ 2, 2, 3 ], [ 2, 3 ], [ 3 ] ]
gap> # number of different hands in a game of poker:
gap> NrCombinations( [1..52], 5 );
2598960
```

The function Arrangements (16.2.4) computes ordered selections without repetitions, UnorderedTuples (16.2.6) computes unordered selections with repetitions, and Tuples (16.2.8) computes ordered selections with repetitions.

16.2.4 Arrangements

▷ Arrangements(*mset[, k]*) (function)

returns the set of arrangements of the multiset *mset* that contain *k* elements. If *k* is not given it returns all arrangements of *mset*.

An *arrangement* of *mset* is an ordered selection without repetitions and is represented by a list that contains only elements from *mset*, but maybe in a different order. If *mset* is a proper set there are $|mset|!/(|mset|-k)!$ (see Factorial (16.1.1)) arrangements with *k* elements.

16.2.5 NrArrangements

▷ NrArrangements(*mset[, k]*) (function)

returns the number of Arrangements(*mset*,*k*).

As an example of arrangements of a multiset, think of the game Scrabble. Suppose you have the six characters of the word "settle" and you have to make a four letter word. Then the possibilities are given by

Example

```
gap> Arrangements( ["s","e","t","t","l","e"], 4 );
[ [ "e", "e", "l", "s" ], [ "e", "e", "l", "t" ], [ "e", "e", "s", "l" ],
  [ "e", "e", "s", "t" ], [ "e", "e", "t", "l" ], [ "e", "e", "t", "s" ],
  ... 93 more possibilities ...
  [ "t", "t", "l", "s" ], [ "t", "t", "s", "e" ], [ "t", "t", "s", "l" ] ]
```

Can you find the five proper English words, where "lets" does not count? Note that the fact that the list returned by Arrangements (16.2.4) is a proper set means in this example that the possibilities are listed in the same order as they appear in the dictionary.

Example

```
gap> NrArrangements( ["s","e","t","t","l","e"] );
523
```

The function Combinations (16.2.1) computes unordered selections without repetitions, UnorderedTuples (16.2.6) computes unordered selections with repetitions, and Tuples (16.2.8) computes ordered selections with repetitions.

16.2.6 UnorderedTuples

▷ UnorderedTuples(*set*, *k*) (function)

returns the set of all unordered tuples of length *k* of the set *set*.

An *unordered tuple* of length *k* of *set* is an unordered selection with repetitions of *set* and is represented by a sorted list of length *k* containing elements from *set*. There are $\binom{|set|+k-1}{k}$ (see Binomial (16.1.2)) such unordered tuples.

Note that the fact that UnorderedTuples returns a set implies that the last index runs fastest. That means the first tuple contains the smallest element from *set* *k* times, the second tuple contains the smallest element of *set* at all positions except at the last positions, where it contains the second smallest element from *set* and so on.

16.2.7 NrUnorderedTuples

▷ NrUnorderedTuples(*set*, *k*) (function)

returns the number of UnorderedTuples(*set*,*k*).

As an example for unordered tuples think of a poker-like game played with 5 dice. Then each possible hand corresponds to an unordered five-tuple from the set $\{1,2,\ldots,6\}$.

Example

```
gap> NrUnorderedTuples( [1..6], 5 );
252
gap> UnorderedTuples( [1..6], 5 );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 2 ], [ 1, 1, 1, 1, 3 ], [ 1, 1, 1, 1, 4 ],
  [ 1, 1, 1, 1, 5 ], [ 1, 1, 1, 1, 6 ], [ 1, 1, 1, 2, 2 ], [ 1, 1, 1, 2, 3 ],
  ... 100 more tuples ...
  [ 1, 3, 5, 5, 6 ], [ 1, 3, 5, 6, 6 ], [ 1, 3, 6, 6, 6 ], [ 1, 4, 4, 4, 4 ],
  ... 100 more tuples ...
  [ 3, 3, 5, 5, 5 ], [ 3, 3, 5, 5, 6 ], [ 3, 3, 5, 6, 6 ], [ 3, 3, 6, 6, 6 ],
  ... 32 more tuples ...
  [ 5, 5, 5, 6, 6 ], [ 5, 5, 6, 6, 6 ], [ 5, 6, 6, 6, 6 ], [ 6, 6, 6, 6, 6 ] ]
```

The function Combinations (16.2.1) computes unordered selections without repetitions, Arrangements (16.2.4) computes ordered selections without repetitions, and Tuples (16.2.8) computes ordered selections with repetitions.

16.2.8 Tuples

▷ Tuples(*set*, *k*) (function)

returns the set of all ordered tuples of length *k* of the set *set*.

An *ordered tuple* of length *k* of *set* is an ordered selection with repetition and is represented by a list of length *k* containing elements of *set*. There are $|set|^k$ such ordered tuples.

Note that the fact that Tuples returns a set implies that the last index runs fastest. That means the first tuple contains the smallest element from *set* *k* times, the second tuple contains the smallest element of *set* at all positions except at the last positions, where it contains the second smallest element from *set* and so on.

16.2.9 EnumeratorOfTuples

▷ EnumeratorOfTuples(*set*, *k*) (function)

This function is referred to as an example of enumerators that are defined by functions but are not constructed from a domain. The result is equal to that of Tuples(*set*, *k*). However, the entries are not stored physically in the list but are created/identified on demand.

16.2.10 IteratorOfTuples

▷ IteratorOfTuples(*set*, *k*) (function)

For a set *set* and a positive integer *k*, IteratorOfTuples returns an iterator (see 30.8) of the set of all ordered tuples (see Tuples (16.2.8)) of length *k* of the set *set*. The tuples are returned in lexicographic order.

16.2.11 NrTuples

▷ NrTuples(*set*, *k*) (function)

returns the number of Tuples(*set*,*k*).

Example

```
gap> Tuples( [1,2,3], 2 );
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 3, 1 ], [ 3, 2 ], [ 3, 3 ] ]
gap> NrTuples( [1..10], 5 );
100000
```

Tuples(*set*,*k*) can also be viewed as the *k*-fold cartesian product of *set* (see Cartesian (21.20.16)).

The function Combinations (16.2.1) computes unordered selections without repetitions, Arrangements (16.2.4) computes ordered selections without repetitions, and finally the function UnorderedTuples (16.2.6) computes unordered selections with repetitions.

16.2.12 PermutationsList

▷ PermutationsList(*mset*) (function)

PermutationsList returns the set of permutations of the multiset *mset*.

A *permutation* is represented by a list that contains exactly the same elements as *mset*, but possibly in different order. If *mset* is a proper set there are $|mset|!$ (see Factorial (16.1.1)) such

permutations. Otherwise if the first elements appears k_1 times, the second element appears k_2 times and so on, the number of permutations is $|mset|!/(k_1!k_2!\ldots)$, which is sometimes called multinomial coefficient.

16.2.13 NrPermutationsList

▷ `NrPermutationsList(mset)` (function)

returns the number of `PermutationsList(mset)`.

Example

```
gap> PermutationsList( [1,2,3] );
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 2, 1, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ],
  [ 3, 2, 1 ] ]
gap> PermutationsList( [1,1,2,2] );
[ [ 1, 1, 2, 2 ], [ 1, 2, 1, 2 ], [ 1, 2, 2, 1 ], [ 2, 1, 1, 2 ],
  [ 2, 1, 2, 1 ], [ 2, 2, 1, 1 ] ]
gap> NrPermutationsList( [1,2,2,3,3,3,4,4,4,4] );
12600
```

The function `Arrangements` (16.2.4) is the generalization of `PermutationsList` (16.2.12) that allows you to specify the size of the permutations. `Derangements` (16.2.14) computes permutations that have no fixed points.

16.2.14 Derangements

▷ `Derangements(list)` (function)

returns the set of all derangements of the list *list*.

A *derangement* is a fixpointfree permutation of *list* and is represented by a list that contains exactly the same elements as *list*, but in such an order that the derangement has at no position the same element as *list*. If the list *list* contains no element twice there are exactly $|list|!(1/2! - 1/3! + 1/4! - \cdots + (-1)^n/n!)$ derangements.

Note that the ratio `NrPermutationsList( [ 1 .. n ] ) / NrDerangements( [ 1 .. n ] )`, which is $n!/(n!(1/2! - 1/3! + 1/4! - \cdots + (-1)^n/n!))$ is an approximation for the base of the natural logarithm $e = 2.7182818285\ldots$, which is correct to about n digits.

16.2.15 NrDerangements

▷ `NrDerangements(list)` (function)

returns the number of `Derangements(list)`.

As an example of derangements suppose that you have to send four different letters to four different people. Then a derangement corresponds to a way to send those letters such that no letter reaches the intended person.

Example

```
gap> Derangements( [1,2,3,4] );
[ [ 2, 1, 4, 3 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 3, 1, 4, 2 ],
  [ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 3, 1, 2 ],
  [ 4, 3, 2, 1 ] ]
```

```
gap> NrDerangements( [1..10] );
1334961
gap> Int( 10^7*NrPermutationsList([1..10])/last );
27182816
gap> Derangements( [1,1,2,2,3,3] );
[ [ 2, 2, 3, 3, 1, 1 ], [ 2, 3, 1, 3, 1, 2 ], [ 2, 3, 1, 3, 2, 1 ],
  [ 2, 3, 3, 1, 1, 2 ], [ 2, 3, 3, 1, 2, 1 ], [ 3, 2, 1, 3, 1, 2 ],
  [ 3, 2, 1, 3, 2, 1 ], [ 3, 2, 3, 1, 1, 2 ], [ 3, 2, 3, 1, 2, 1 ],
  [ 3, 3, 1, 1, 2, 2 ] ]
gap> NrDerangements( [1,2,2,3,3,3,4,4,4,4] );
338
```

The function PermutationsList (16.2.12) computes all permutations of a list.

16.2.16 PartitionsSet

▷ PartitionsSet(*set[, k]*) (function)

returns the set of all unordered partitions of the set *set* into *k* pairwise disjoint nonempty sets. If *k* is not given it returns all unordered partitions of *set* for all *k*.

An *unordered partition* of *set* is a set of pairwise disjoint nonempty sets with union *set* and is represented by a sorted list of such sets. There are $B(|set|)$ (see Bell (16.1.3)) partitions of the set *set* and $S_2(|set|,k)$ (see Stirling2 (16.1.6)) partitions with *k* elements.

16.2.17 NrPartitionsSet

▷ NrPartitionsSet(*set[, k]*) (function)

returns the number of PartitionsSet(*set*,*k*).

Example

```
gap> PartitionsSet( [1,2,3] );
[ [ [ 1 ], [ 2 ], [ 3 ] ], [ [ 1 ], [ 2, 3 ] ], [ [ 1, 2 ], [ 3 ] ],
  [ [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 2 ] ] ]
gap> PartitionsSet( [1,2,3,4], 2 );
[ [ [ 1 ], [ 2, 3, 4 ] ], [ [ 1, 2 ], [ 3, 4 ] ],
  [ [ 1, 2, 3 ], [ 4 ] ], [ [ 1, 2, 4 ], [ 3 ] ],
  [ [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 3, 4 ], [ 2 ] ],
  [ [ 1, 4 ], [ 2, 3 ] ] ]
gap> NrPartitionsSet( [1..6] );
203
gap> NrPartitionsSet( [1..10], 3 );
9330
```

Note that PartitionsSet (16.2.16) does currently not support multisets and that there is currently no ordered counterpart.

16.2.18 Partitions

▷ Partitions(*n[, k]*) (function)

returns the set of all (unordered) partitions of the positive integer `n` into sums with `k` summands. If `k` is not given it returns all unordered partitions of `set` for all `k`.

An *unordered partition* is an unordered sum $n = p_1 + p_2 + \cdots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order, i.e., $p_1 \geq p_2 \geq \ldots \geq p_k$. We write $p \vdash n$. There are approximately $\exp(\pi\sqrt{2/3n})/(4\sqrt{3}n)$ such partitions, use `NrPartitions` (16.2.20) to compute the precise number.

If you want to loop over all partitions of some larger `n` use the more memory efficient `IteratorOfPartitions` (16.2.19).

It is possible to associate with every partition of the integer `n` a conjugacy class of permutations in the symmetric group on `n` points and vice versa. Therefore $p(n) :=$`NrPartitions`(n) is the number of conjugacy classes of the symmetric group on `n` points.

Ramanujan found the identities $p(5i+4) = 0 \bmod 5$, $p(7i+5) = 0 \bmod 7$ and $p(11i+6) = 0 \bmod 11$ and many other fascinating things about the number of partitions.

16.2.19 IteratorOfPartitions

▷ `IteratorOfPartitions(n)` (function)

For a positive integer `n`, `IteratorOfPartitions` returns an iterator (see 30.8) of the set of partitions of `n` (see `Partitions` (16.2.18)). The partitions of `n` are returned in lexicographic order.

16.2.20 NrPartitions

▷ `NrPartitions(n[, k])` (function)

returns the number of `Partitions(set,k)`.

Example

```
gap> Partitions( 7 );
[ [ 1, 1, 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1, 1, 1 ], [ 2, 2, 1, 1, 1 ],
  [ 2, 2, 2, 1 ], [ 3, 1, 1, 1, 1 ], [ 3, 2, 1, 1 ], [ 3, 2, 2 ],
  [ 3, 3, 1 ], [ 4, 1, 1, 1 ], [ 4, 2, 1 ], [ 4, 3 ], [ 5, 1, 1 ],
  [ 5, 2 ], [ 6, 1 ], [ 7 ] ]
gap> Partitions( 8, 3 );
[ [ 3, 3, 2 ], [ 4, 2, 2 ], [ 4, 3, 1 ], [ 5, 2, 1 ], [ 6, 1, 1 ] ]
gap> NrPartitions( 7 );
15
gap> NrPartitions( 100 );
190569292
```

The function `OrderedPartitions` (16.2.21) is the ordered counterpart of `Partitions` (16.2.18).

16.2.21 OrderedPartitions

▷ `OrderedPartitions(n[, k])` (function)

returns the set of all ordered partitions of the positive integer `n` into sums with `k` summands. If `k` is not given it returns all ordered partitions of `set` for all `k`.

An *ordered partition* is an ordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $[p_1, p_2, \ldots, p_k]$. There are totally 2^{n-1} ordered partitions and $\binom{n-1}{k-1}$ (see `Binomial` (16.1.2)) ordered partitions with `k` summands.

Do not call `OrderedPartitions` with an `n` much larger than 15, the list will simply become too large.

16.2.22 NrOrderedPartitions

▷ `NrOrderedPartitions(n[, k])` (function)

returns the number of `OrderedPartitions(set,k)`.

Example

```
gap> OrderedPartitions( 5 );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 1, 3 ],
  [ 1, 2, 1, 1 ], [ 1, 2, 2 ], [ 1, 3, 1 ], [ 1, 4 ], [ 2, 1, 1, 1 ],
  [ 2, 1, 2 ], [ 2, 2, 1 ], [ 2, 3 ], [ 3, 1, 1 ], [ 3, 2 ],
  [ 4, 1 ], [ 5 ] ]
gap> OrderedPartitions( 6, 3 );
[ [ 1, 1, 4 ], [ 1, 2, 3 ], [ 1, 3, 2 ], [ 1, 4, 1 ], [ 2, 1, 3 ],
  [ 2, 2, 2 ], [ 2, 3, 1 ], [ 3, 1, 2 ], [ 3, 2, 1 ], [ 4, 1, 1 ] ]
gap> NrOrderedPartitions(20);
524288
```

The function `Partitions` (16.2.18) is the unordered counterpart of `OrderedPartitions` (16.2.21).

16.2.23 PartitionsGreatestLE

▷ `PartitionsGreatestLE(n, m)` (function)

returns the set of all (unordered) partitions of the integer `n` having parts less or equal to the integer `m`.

16.2.24 PartitionsGreatestEQ

▷ `PartitionsGreatestEQ(n, m)` (function)

returns the set of all (unordered) partitions of the integer `n` having greatest part equal to the integer `m`.

16.2.25 RestrictedPartitions

▷ `RestrictedPartitions(n, set[, k])` (function)

In the first form `RestrictedPartitions` returns the set of all restricted partitions of the positive integer `n` into sums with `k` summands with the summands of the partition coming from the set `set`. If `k` is not given all restricted partitions for all `k` are returned.

A *restricted partition* is like an ordinary partition (see `Partitions` (16.2.18)) an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order. The difference is that here the p_i must be elements from the set *`set`*, while for ordinary partitions they may be elements from `[ 1 .. n ]`.

16.2.26 NrRestrictedPartitions

▷ `NrRestrictedPartitions(`*`n, set[, k]`*`)` (function)

returns the number of `RestrictedPartitions(`*`n`*`,`*`set`*`,`*`k`*`)`.

Example

```
gap> RestrictedPartitions( 8, [1,3,5,7] );
[ [ 1, 1, 1, 1, 1, 1, 1, 1 ], [ 3, 1, 1, 1, 1, 1 ], [ 3, 3, 1, 1 ],
  [ 5, 1, 1, 1 ], [ 5, 3 ], [ 7, 1 ] ]
gap> NrRestrictedPartitions(50,[1,2,5,10,20,50]);
451
```

The last example tells us that there are 451 ways to return 50 pence change using 1, 2, 5, 10, 20 and 50 pence coins.

16.2.27 SignPartition

▷ `SignPartition(`*`pi`*`)` (function)

returns the sign of a permutation with cycle structure *`pi`*.

This function actually describes a homomorphism from the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n (see `SignPerm` (42.4.1)). Partitions of sign 1 are called *even* partitions while partitions of sign -1 are called *odd*.

Example

```
gap> SignPartition([6,5,4,3,2,1]);
-1
```

16.2.28 AssociatedPartition

▷ `AssociatedPartition(`*`pi`*`)` (function)

`AssociatedPartition` returns the associated partition of the partition *`pi`* which is obtained by transposing the corresponding Young diagram.

Example

```
gap> AssociatedPartition([4,2,1]);
[ 3, 2, 1, 1 ]
gap> AssociatedPartition([6]);
[ 1, 1, 1, 1, 1, 1 ]
```

16.2.29 PowerPartition

▷ `PowerPartition(pi, k)` (function)

`PowerPartition` returns the partition corresponding to the `k`-th power of a permutation with cycle structure `pi`.

Each part l of `pi` is replaced by $d = \gcd(l,k)$ parts l/d. So if `pi` is a partition of n then `pi^k` also is a partition of n. `PowerPartition` describes the power map of symmetric groups.

Example

```
gap> PowerPartition([6,5,4,3,2,1], 3);
[ 5, 4, 2, 2, 2, 2, 1, 1, 1, 1 ]
```

16.2.30 PartitionTuples

▷ `PartitionTuples(n, r)` (function)

`PartitionTuples` returns the list of all `r`-tuples of partitions which together form a partition of `n`.

`r`-tuples of partitions describe the classes and the characters of wreath products of groups with `r` conjugacy classes with the symmetric group S_n.

16.2.31 NrPartitionTuples

▷ `NrPartitionTuples(n, r)` (function)

returns the number of `PartitionTuples( n, r )`.

Example

```
gap> PartitionTuples(3, 2);
[ [ [ 1, 1, 1 ], [  ] ], [ [ 1, 1 ], [ 1 ] ], [ [ 1 ], [ 1, 1 ] ],
  [ [  ], [ 1, 1, 1 ] ], [ [ 2, 1 ], [  ] ], [ [ 1 ], [ 2 ] ],
  [ [ 2 ], [ 1 ] ], [ [  ], [ 2, 1 ] ], [ [ 3 ], [  ] ],
  [ [  ], [ 3 ] ] ]
```

16.3 Fibonacci and Lucas Sequences

16.3.1 Fibonacci

▷ `Fibonacci(n)` (function)

returns the `n`th number of the *Fibonacci sequence*. The Fibonacci sequence F_n is defined by the initial conditions $F_1 = F_2 = 1$ and the recurrence relation $F_{n+2} = F_{n+1} + F_n$. For negative n we define $F_n = (-1)^{n+1} F_{-n}$, which is consistent with the recurrence relation.

Using generating functions one can prove that $F_n = \phi^n - 1/\phi^n$, where ϕ is $(\sqrt{5}+1)/2$, i.e., one root of $x^2 - x - 1 = 0$. Fibonacci numbers have the property $\gcd(F_m, F_n) = F_{\gcd(m,n)}$. But a pair of Fibonacci numbers requires more division steps in Euclid's algorithm (see `Gcd` (56.7.1)) than any other pair of integers of the same size. `Fibonacci(k)` is the special case `Lucas(1,-1,k)[1]` (see `Lucas` (16.3.2)).

Example

```
gap> Fibonacci( 10 );
55
gap> Fibonacci( 35 );
9227465
gap> Fibonacci( -10 );
-55
```

16.3.2 Lucas

▷ `Lucas(P, Q, k)` (function)

returns the `k`-th values of the *Lucas sequence* with parameters `P` and `Q`, which must be integers, as a list of three integers. If `k` is a negative integer, then the values of the Lucas sequence may be nonintegral rational numbers, with denominator roughly `Q^k`.

Let α, β be the two roots of $x^2 - Px + Q$ then we define `Lucas( P, Q, k )[1]` $= U_k = (\alpha^k - \beta^k)/(\alpha - \beta)$ and `Lucas( P, Q, k )[2]` $= V_k = (\alpha^k + \beta^k)$ and as a convenience `Lucas( P, Q, k )[3]` $= Q^k$.

The following recurrence relations are easily derived from the definition $U_0 = 0, U_1 = 1, U_k = PU_{k-1} - QU_{k-2}$ and $V_0 = 2, V_1 = P, V_k = PV_{k-1} - QV_{k-2}$. Those relations are actually used to define `Lucas` if $\alpha = \beta$.

Also the more complex relations used in `Lucas` can be easily derived $U_{2k} = U_k V_k$, $U_{2k+1} = (PU_{2k} + V_{2k})/2$ and $V_{2k} = V_k^2 - 2Q^k$, $V_{2k+1} = ((P^2 - 4Q)U_{2k} + PV_{2k})/2$.

`Fibonacci(k)` (see `Fibonacci` (16.3.1)) is simply `Lucas(1,-1,k)[1]`. In an abuse of notation, the sequence `Lucas(1,-1,k)[2]` is sometimes called the Lucas sequence.

Example

```
gap> List( [0..10], i -> Lucas(1,-2,i)[1] );     # 2^k - (-1)^k)/3
[ 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341 ]
gap> List( [0..10], i -> Lucas(1,-2,i)[2] );     # 2^k + (-1)^k
[ 2, 1, 5, 7, 17, 31, 65, 127, 257, 511, 1025 ]
gap> List( [0..10], i -> Lucas(1,-1,i)[1] );     # Fibonacci sequence
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
gap> List( [0..10], i -> Lucas(2,1,i)[1] );      # the roots are equal
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

16.4 Permanent of a Matrix

16.4.1 Permanent

▷ `Permanent(mat)` (function)

returns the *permanent* of the matrix `mat`. The permanent is defined by $\sum_{p \in Sym(n)} \prod_{i=1}^{n} mat[i][i^p]$.

Note the similarity of the definition of the permanent to the definition of the determinant (see `DeterminantMat` (24.4.4)). In fact the only difference is the missing sign of the permutation. However the permanent is quite unlike the determinant, for example it is not multilinear or alternating. It has however important combinatorial properties.

Example

```
gap> Permanent( [[0,1,1,1],
>      [1,0,1,1],
>      [1,1,0,1],
>      [1,1,1,0]] );  # inefficient way to compute NrDerangements([1..4])
9
gap> # 24 permutations fit the projective plane of order 2:
gap> Permanent( [[1,1,0,1,0,0,0],
>      [0,1,1,0,1,0,0],
>      [0,0,1,1,0,1,0],
>      [0,0,0,1,1,0,1],
>      [1,0,0,0,1,1,0],
>      [0,1,0,0,0,1,1],
>      [1,0,1,0,0,0,1]] );
24
```

Chapter 17

Rational Numbers

The *rationals* form a very important field. On the one hand it is the quotient field of the integers (see chapter 14). On the other hand it is the prime field of the fields of characteristic zero (see chapter 60).

The former comment suggests the representation actually used. A rational is represented as a pair of integers, called *numerator* and *denominator*. Numerator and denominator are *reduced*, i.e., their greatest common divisor is 1. If the denominator is 1, the rational is in fact an integer and is represented as such. The numerator holds the sign of the rational, thus the denominator is always positive.

Because the underlying integer arithmetic can compute with arbitrary size integers, the rational arithmetic is always exact, even for rationals whose numerators and denominators have thousands of digits.

Example

```
gap> 2/3;
2/3
gap> 66/123;  # numerator and denominator are made relatively prime
22/41
gap> 17/-13;  # the numerator carries the sign;
-17/13
gap> 121/11;  # rationals with denominator 1 (when canceled) are integers
11
```

17.1 Rationals: Global Variables

17.1.1 Rationals

▷ `Rationals` (global variable)

▷ `IsRationals(`*obj*`)` (property)

`Rationals` is the field $\mathbb{Q}$ of rational integers, as a set of cyclotomic numbers, see Chapter 18 for basic operations, Functions for the field `Rationals` can be found in the chapters 58 and 60.

`IsRationals` returns `true` for a prime field that consists of cyclotomic numbers –for example the GAP object `Rationals`– and `false` for all other GAP objects.

Example

```
gap> Size( Rationals ); 2/3 in Rationals;
infinity
true
```

17.2 Elementary Operations for Rationals

17.2.1 IsRat

▷ IsRat(*obj*) (Category)

Every rational number lies in the category IsRat, which is a subcategory of IsCyc (18.1.3).

Example

```
gap> IsRat( 2/3 );
true
gap> IsRat( 17/-13 );
true
gap> IsRat( 11 );
true
gap> IsRat( IsRat );  # 'IsRat' is a function, not a rational
false
```

17.2.2 IsPosRat

▷ IsPosRat(*obj*) (Category)

Every positive rational number lies in the category IsPosRat.

17.2.3 IsNegRat

▷ IsNegRat(*obj*) (Category)

Every negative rational number lies in the category IsNegRat.

17.2.4 NumeratorRat

▷ NumeratorRat(*rat*) (function)

NumeratorRat returns the numerator of the rational *rat*. Because the numerator holds the sign of the rational it may be any integer. Integers are rationals with denominator 1, thus NumeratorRat is the identity function for integers.

Example

```
gap> NumeratorRat( 2/3 );
2
gap> # numerator and denominator are made relatively prime:
gap> NumeratorRat( 66/123 );
22
gap> NumeratorRat( 17/-13 );  # numerator holds the sign of the rational
-17
gap> NumeratorRat( 11 );      # integers are rationals with denominator 1
11
```

17.2.5 DenominatorRat

▷ DenominatorRat(*rat*) (function)

DenominatorRat returns the denominator of the rational *rat*. Because the numerator holds the sign of the rational the denominator is always a positive integer. Integers are rationals with the denominator 1, thus DenominatorRat returns 1 for integers.

Example

```
gap> DenominatorRat( 2/3 );
3
gap> # numerator and denominator are made relatively prime:
gap> DenominatorRat( 66/123 );
41
gap> # the denominator holds the sign of the rational:
gap> DenominatorRat( 17/-13 );
13
gap> DenominatorRat( 11 ); # integers are rationals with denominator 1
1
```

17.2.6 Rat

▷ Rat(*elm*) (attribute)

Rat returns a rational number *rat* whose meaning depends on the type of *elm*.

If *elm* is a string consisting of digits '0', '1', ..., '9' and '-' (at the first position), '/' and the decimal dot '.' then *rat* is the rational described by this string. The operation String (27.6.6) can be used to compute a string for rational numbers, in fact for all cyclotomics.

Example

```
gap> Rat( "1/2" );  Rat( "35/14" );  Rat( "35/-27" );  Rat( "3.14159" );
1/2
5/2
-35/27
314159/100000
```

17.2.7 Random (for rationals)

▷ Random(*Rationals*) (operation)

Random for rationals returns pseudo random rationals which are the quotient of two random integers. See the description of Random (14.2.12) for details. (Also see Random (30.7.1).)

Chapter 18

Cyclotomic Numbers

GAP admits computations in abelian extension fields of the rational number field $\mathbb{Q}$, that is fields with abelian Galois group over $\mathbb{Q}$. These fields are subfields of *cyclotomic fields* $\mathbb{Q}(e_n)$ where $e_n = \exp(2\pi i/n)$ is a primitive complex n-th root of unity. The elements of these fields are called *cyclotomics*.

Information concerning operations for domains of cyclotomics, for example certain integral bases of fields of cyclotomics, can be found in Chapter 60. For more general operations that take a field extension as a –possibly optional– argument, e.g., `Trace` (58.3.5) or `Coefficients` (61.6.3), see Chapter 58.

18.1 Operations for Cyclotomics

18.1.1 E

▷ `E(n)` (function)

`E` returns the primitive `n`-th root of unity $e_n = \exp(2\pi i/n)$. Cyclotomics are usually entered as sums of roots of unity, with rational coefficients, and irrational cyclotomics are displayed in such a way. (For special cyclotomics, see 18.4.)

Example

```
gap> E(9); E(9)^3; E(6); E(12) / 3;
-E(9)^4-E(9)^7
E(3)
-E(3)^2
-1/3*E(12)^7
```

A particular basis is used to express cyclotomics, see 60.3; note that `E(9)` is *not* a basis element, as the above example shows.

18.1.2 Cyclotomics

▷ `Cyclotomics` (global variable)

is the domain of all cyclotomics.

Example

```
gap> E(9) in Cyclotomics; 37 in Cyclotomics; true in Cyclotomics;
true
true
false
```

As the cyclotomics are field elements, the usual arithmetic operators +, -, * and / (and ^ to take powers by integers) are applicable. Note that ^ does *not* denote the conjugation of group elements, so it is *not* possible to explicitly construct groups of cyclotomics. (However, it is possible to compute the inverse and the multiplicative order of a nonzero cyclotomic.) Also, taking the k-th power of a root of unity z defines a Galois automorphism if and only if k is coprime to the conductor (see Conductor (18.1.7)) of z.

Example

```
gap> E(5) + E(3); (E(5) + E(5)^4) ^ 2; E(5) / E(3); E(5) * E(3);
-E(15)^2-2*E(15)^8-E(15)^11-E(15)^13-E(15)^14
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
E(15)^13
E(15)^8
gap> Order( E(5) ); Order( 1+E(5) );
5
infinity
```

18.1.3 IsCyclotomic

▷ IsCyclotomic(*obj*) (Category)
▷ IsCyc(*obj*) (Category)

Every object in the family CyclotomicsFamily lies in the category IsCyclotomic. This covers integers, rationals, proper cyclotomics, the object infinity (18.2.1), and unknowns (see Chapter 74). All these objects except infinity (18.2.1) and unknowns lie also in the category IsCyc, infinity (18.2.1) lies in (and can be detected from) the category IsInfinity (18.2.1), and unknowns lie in IsUnknown (74.1.3).

Example

```
gap> IsCyclotomic(0); IsCyclotomic(1/2*E(3)); IsCyclotomic( infinity );
true
true
true
gap> IsCyc(0); IsCyc(1/2*E(3)); IsCyc( infinity );
true
true
false
```

18.1.4 IsIntegralCyclotomic

▷ IsIntegralCyclotomic(*obj*) (property)

A cyclotomic is called *integral* or a *cyclotomic integer* if all coefficients of its minimal polynomial over the rationals are integers. Since the underlying basis of the external representation of cyclotomics

is an integral basis (see 60.3), the subring of cyclotomic integers in a cyclotomic field is formed by those cyclotomics for which the external representation is a list of integers. For example, square roots of integers are cyclotomic integers (see 18.4), any root of unity is a cyclotomic integer, character values are always cyclotomic integers, but all rationals which are not integers are not cyclotomic integers.

Example

```
gap> r:= ER( 5 );               # The square root of 5 ...
E(5)-E(5)^2-E(5)^3+E(5)^4
gap> IsIntegralCyclotomic( r ); # ... is a cyclotomic integer.
true
gap> r2:= 1/2 * r;              # This is not a cyclotomic integer, ...
1/2*E(5)-1/2*E(5)^2-1/2*E(5)^3+1/2*E(5)^4
gap> IsIntegralCyclotomic( r2 );
false
gap> r3:= 1/2 * r - 1/2;        # ... but this is one.
E(5)+E(5)^4
gap> IsIntegralCyclotomic( r3 );
true
```

18.1.5 Int (for a cyclotomic)

▷ `Int(`*cyc*`)` (function)

The operation `Int` can be used to find a cyclotomic integer near to an arbitrary cyclotomic, by applying `Int` (14.2.3) to the coefficients.

Example

```
gap> Int( E(5)+1/2*E(5)^2 ); Int( 2/3*E(7)-3/2*E(4) );
E(5)
-E(4)
```

18.1.6 String (for a cyclotomic)

▷ `String(`*cyc*`)` (method)

The operation `String` returns for a cyclotomic *cyc* a string corresponding to the way the cyclotomic is printed by `ViewObj` (6.3.5) and `PrintObj` (6.3.5).

Example

```
gap> String( E(5)+1/2*E(5)^2 ); String( 17/3 );
"E(5)+1/2*E(5)^2"
"17/3"
```

18.1.7 Conductor (for a cyclotomic)

▷ `Conductor(`*cyc*`)` (attribute)
▷ `Conductor(`*C*`)` (attribute)

For an element *cyc* of a cyclotomic field, `Conductor` returns the smallest integer n such that *cyc* is contained in the n-th cyclotomic field. For a collection *C* of cyclotomics (for example a dense list of

cyclotomics or a field of cyclotomics), Conductor returns the smallest integer n such that all elements of C are contained in the n-th cyclotomic field.

Example

```
gap> Conductor( 0 ); Conductor( E(10) ); Conductor( E(12) );
1
5
12
```

18.1.8 AbsoluteValue

▷ AbsoluteValue(*cyc*) (attribute)

returns the absolute value of a cyclotomic number *cyc*. At the moment only methods for rational numbers exist.

Example

```
gap> AbsoluteValue(-3);
3
```

18.1.9 RoundCyc

▷ RoundCyc(*cyc*) (operation)

is a cyclotomic integer z (see IsIntegralCyclotomic (18.1.4)) near to the cyclotomic *cyc* in the following sense: Let c be the i-th coefficient in the external representation (see CoeffsCyc (18.1.10)) of *cyc*. Then the i-th coefficient in the external representation of z is Int(c + 1/2) or Int(c - 1/2), depending on whether c is nonnegative or negative, respectively.

Expressed in terms of the Zumbroich basis (see 60.3), rounding the coefficients of *cyc* w.r.t. this basis to the nearest integer yields the coefficients of z.

Example

```
gap> RoundCyc( E(5)+1/2*E(5)^2 ); RoundCyc( 2/3*E(7)+3/2*E(4) );
E(5)+E(5)^2
-2*E(28)^3+E(28)^4-2*E(28)^11-2*E(28)^15-2*E(28)^19-2*E(28)^23
 -2*E(28)^27
```

18.1.10 CoeffsCyc

▷ CoeffsCyc(*cyc, N*) (function)

Let *cyc* be a cyclotomic with conductor n (see Conductor (18.1.7)). If N is not a multiple of n then CoeffsCyc returns fail because *cyc* cannot be expressed in terms of N-th roots of unity. Otherwise CoeffsCyc returns a list of length N with entry at position j equal to the coefficient of $\exp(2\pi i(j-1)/N)$ if this root belongs to the N-th Zumbroich basis (see 60.3), and equal to zero otherwise. So we have *cyc* = CoeffsCyc(*cyc*, N) * List([1..N], j -> E(N)^(j-1)).

Example

```
gap> cyc:= E(5)+E(5)^2;
E(5)+E(5)^2
gap> CoeffsCyc( cyc, 5 );  CoeffsCyc( cyc, 15 );  CoeffsCyc( cyc, 7 );
[ 0, 1, 1, 0, 0 ]
```

```
[ 0, -1, 0, 0, 0, 0, 0, 0, -1, 0, 0, -1, 0, -1, 0 ]
fail
```

18.1.11 DenominatorCyc

▷ DenominatorCyc(*cyc*) (function)

For a cyclotomic number *cyc* (see IsCyclotomic (18.1.3)), this function returns the smallest positive integer *n* such that *n* * *cyc* is a cyclotomic integer (see IsIntegralCyclotomic (18.1.4)). For rational numbers *cyc*, the result is the same as that of DenominatorRat (17.2.5).

18.1.12 ExtRepOfObj (for a cyclotomic)

▷ ExtRepOfObj(*cyc*) (method)

The external representation of a cyclotomic *cyc* with conductor *n* (see Conductor (18.1.7) is the list returned by CoeffsCyc (18.1.10), called with *cyc* and *n*.

Example

```
gap> ExtRepOfObj( E(5) ); CoeffsCyc( E(5), 5 );
[ 0, 1, 0, 0, 0 ]
[ 0, 1, 0, 0, 0 ]
gap> CoeffsCyc( E(5), 15 );
[ 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, -1, 0 ]
```

18.1.13 DescriptionOfRootOfUnity

▷ DescriptionOfRootOfUnity(*root*) (function)

Given a cyclotomic *root* that is known to be a root of unity (this is *not* checked), DescriptionOfRootOfUnity returns a list $[n,e]$ of coprime positive integers such that *root* $= \mathrm{E}(n)^e$ holds.

Example

```
gap> E(9);  DescriptionOfRootOfUnity( E(9) );
-E(9)^4-E(9)^7
[ 9, 1 ]
gap> DescriptionOfRootOfUnity( -E(3) );
[ 6, 5 ]
```

18.1.14 IsGaussInt

▷ IsGaussInt(*x*) (function)

IsGaussInt returns true if the object *x* is a Gaussian integer (see GaussianIntegers (60.5.1)), and false otherwise. Gaussian integers are of the form $a+b$*E(4), where a and b are integers.

18.1.15 IsGaussRat

▷ IsGaussRat(*x*) (function)

IsGaussRat returns true if the object *x* is a Gaussian rational (see GaussianRationals (60.1.3)), and false otherwise. Gaussian rationals are of the form $a + b$*E(4), where a and b are rationals.

18.1.16 DefaultField (for cyclotomics)

▷ DefaultField(*list*) (function)

DefaultField for cyclotomics is defined to return the smallest *cyclotomic* field containing the given elements.

Note that Field (58.1.3) returns the smallest field containing all given elements, which need not be a cyclotomic field. In both cases, the fields represent vector spaces over the rationals (see 60.3).

Example

```
gap> Field( E(5)+E(5)^4 );  DefaultField( E(5)+E(5)^4 );
NF(5,[ 1, 4 ])
CF(5)
```

18.2 Infinity and negative Infinity

18.2.1 IsInfinity

▷ IsInfinity(*obj*) (Category)
▷ IsNegInfinity(*obj*) (Category)
▷ infinity (global variable)
▷ -infinity (global variable)

infinity and -infinity are special GAP objects that lie in CyclotomicsFamily. They are larger or smaller than all other objects in this family respectively. infinity is mainly used as return value of operations such as Size (30.4.6) and Dimension (57.3.3) for infinite and infinite dimensional domains, respectively.

Some arithmetic operations are provided for convenience when using infinity and -infinity as top and bottom element respectively.

Example

```
gap> -infinity + 1;
-infinity
gap> infinity + infinity;
infinity
```

Often it is useful to distinguish infinity from "proper" cyclotomics. For that, infinity lies in the category IsInfinity but not in IsCyc (18.1.3), and the other cyclotomics lie in the category IsCyc (18.1.3) but not in IsInfinity.

Example

```
gap> s:= Size( Rationals );
infinity
```

```
gap> s = infinity; IsCyclotomic( s ); IsCyc( s ); IsInfinity( s );
true
true
false
true
gap> s in Rationals; s > 17;
false
true
gap> Set( [ s, 2, s, E(17), s, 19 ] );
[ 2, 19, E(17), infinity ]
```

18.3 Comparisons of Cyclotomics

To compare cyclotomics, the operators <, <=, =, >=, >, and <> can be used, the result will be `true` if the first operand is smaller, smaller or equal, equal, larger or equal, larger, or unequal, respectively, and `false` otherwise.

Cyclotomics are ordered as follows: The relation between rationals is the natural one, rationals are smaller than irrational cyclotomics, and `infinity` (18.2.1) is the largest cyclotomic. For two irrational cyclotomics with different conductors (see `Conductor` (18.1.7)), the one with smaller conductor is regarded as smaller. Two irrational cyclotomics with same conductor are compared via their external representation (see `ExtRepOfObj` (18.1.12)).

For comparisons of cyclotomics and other GAP objects, see Section 4.12.

Example

```
gap> E(5) < E(6);      # the latter value has conductor 3
false
gap> E(3) < E(3)^2;    # both have conductor 3, compare the ext. repr.
false
gap> 3 < E(3); E(5) < E(7);
true
true
```

18.4 ATLAS Irrationalities

18.4.1 EB, EC, ..., EH

▷ `EB(`*N*`)` (function)
▷ `EC(`*N*`)` (function)
▷ `ED(`*N*`)` (function)
▷ `EE(`*N*`)` (function)
▷ `EF(`*N*`)` (function)
▷ `EG(`*N*`)` (function)
▷ `EH(`*N*`)` (function)

For a positive integer N, let $z = \texttt{E(}N\texttt{)} = \exp(2\pi i/N)$. The following so-called *atomic irrationalities* (see [CCN+85, Chapter 7, Section 10]) can be entered using functions. (Note that the values are not necessary irrational.)

$$\begin{array}{lllll}
\texttt{EB(}N\texttt{)} & = & b_N & = & \left(\sum_{j=1}^{N-1} z^{j^2}\right)/2 \quad , \quad N \equiv 1 \pmod 2 \\
\texttt{EC(}N\texttt{)} & = & c_N & = & \left(\sum_{j=1}^{N-1} z^{j^3}\right)/3 \quad , \quad N \equiv 1 \pmod 3 \\
\texttt{ED(}N\texttt{)} & = & d_N & = & \left(\sum_{j=1}^{N-1} z^{j^4}\right)/4 \quad , \quad N \equiv 1 \pmod 4 \\
\texttt{EE(}N\texttt{)} & = & e_N & = & \left(\sum_{j=1}^{N-1} z^{j^5}\right)/5 \quad , \quad N \equiv 1 \pmod 5 \\
\texttt{EF(}N\texttt{)} & = & f_N & = & \left(\sum_{j=1}^{N-1} z^{j^6}\right)/6 \quad , \quad N \equiv 1 \pmod 6 \\
\texttt{EG(}N\texttt{)} & = & g_N & = & \left(\sum_{j=1}^{N-1} z^{j^7}\right)/7 \quad , \quad N \equiv 1 \pmod 7 \\
\texttt{EH(}N\texttt{)} & = & h_N & = & \left(\sum_{j=1}^{N-1} z^{j^8}\right)/8 \quad , \quad N \equiv 1 \pmod 8
\end{array}$$

(Note that in `EC(`N`)`, ..., `EH(`N`)`, N must be a prime.)

Example

```
gap> EB(5);  EB(9);
E(5)+E(5)^4
1
```

18.4.2 EI and ER

▷ `EI(`N`)` (function)
▷ `ER(`N`)` (function)

For a rational number N, `ER` returns the square root $\sqrt{N}$ of N, and `EI` returns $\sqrt{-N}$. By the chosen embedding of cyclotomic fields into the complex numbers, `ER` returns the positive square root if N is positive, and if N is negative then `ER(`N`) = EI(-`N`)` holds. In any case, `EI(`N`) = E(4) * ER(`N`)`.

`ER` is installed as method for the operation `Sqrt` (31.12.5), for rational argument.

From a theorem of Gauss we know that $b_N =$

$$\begin{array}{ll}
(-1+\sqrt{N})/2 & \text{if} \quad N \equiv 1 \pmod 4 \\
(-1+i\sqrt{N})/2 & \text{if} \quad N \equiv -1 \pmod 4
\end{array}$$

So $\sqrt{N}$ can be computed from b_N, see `EB` (18.4.1).

Example

```
gap> ER(3); EI(3);
-E(12)^7+E(12)^11
E(3)-E(3)^2
```

18.4.3 EY, EX, ..., ES

▷ `EY(`N`[, `d`])` (function)
▷ `EX(`N`[, `d`])` (function)
▷ `EW(`N`[, `d`])` (function)
▷ `EV(`N`[, `d`])` (function)
▷ `EU(`N`[, `d`])` (function)
▷ `ET(`N`[, `d`])` (function)
▷ `ES(`N`[, `d`])` (function)

For the given integer $N > 2$, let N_k denote the first integer with multiplicative order exactly k modulo N, chosen in the order of preference

$$1, -1, 2, -2, 3, -3, 4, -4, \ldots.$$

We define (with $z = \exp(2\pi i/N)$)

$$\begin{array}{lllll}
\texttt{EY(}N\texttt{)} & = & y_N & = & z+z^n & (n = N_2) \\
\texttt{EX(}N\texttt{)} & = & x_N & = & z+z^n+z^{n^2} & (n = N_3) \\
\texttt{EW(}N\texttt{)} & = & w_N & = & z+z^n+z^{n^2}+z^{n^3} & (n = N_4) \\
\texttt{EV(}N\texttt{)} & = & v_N & = & z+z^n+z^{n^2}+z^{n^3}+z^{n^4} & (n = N_5) \\
\texttt{EU(}N\texttt{)} & = & u_N & = & z+z^n+z^{n^2}+\ldots+z^{n^5} & (n = N_6) \\
\texttt{ET(}N\texttt{)} & = & t_N & = & z+z^n+z^{n^2}+\ldots+z^{n^6} & (n = N_7) \\
\texttt{ES(}N\texttt{)} & = & s_N & = & z+z^n+z^{n^2}+\ldots+z^{n^7} & (n = N_8)
\end{array}$$

For the two-argument versions of the functions, see Section NK (18.4.5).

Example

```
gap> EY(5);
E(5)+E(5)^4
gap> EW(16,3); EW(17,2);
0
E(17)+E(17)^4+E(17)^13+E(17)^16
```

18.4.4 EM, EL, ..., EJ

▷ EM(*N*[, *d*]) (function)
▷ EL(*N*[, *d*]) (function)
▷ EK(*N*[, *d*]) (function)
▷ EJ(*N*[, *d*]) (function)

Let N be an integer, $N > 2$. We define (with $z = \exp(2\pi i/N)$)

$$\begin{array}{lllll}
\texttt{EM(}N\texttt{)} & = & m_N & = & z-z^n & (n = N_2) \\
\texttt{EL(}N\texttt{)} & = & l_N & = & z-z^n+z^{n^2}-z^{n^3} & (n = N_4) \\
\texttt{EK(}N\texttt{)} & = & k_N & = & z-z^n+\ldots-z^{n^5} & (n = N_6) \\
\texttt{EJ(}N\texttt{)} & = & j_N & = & z-z^n+\ldots-z^{n^7} & (n = N_8)
\end{array}$$

For the two-argument versions of the functions, see Section NK (18.4.5).

18.4.5 NK

▷ NK(*N*, *k*, *d*) (function)

Let $N_k^{(d)}$ be the $(d+1)$-th integer with multiplicative order exactly k modulo N, chosen in the order of preference defined in Section 18.4.3; NK returns $N_k^{(d)}$; if there is no integer with the required multiplicative order, NK returns fail.

We write $N_k = N_k^{(0)}, N'_k = N_k^{(1)}, N''_k = N_k^{(2)}$ and so on.

The algebraic numbers

$$y'_N = y_N^{(1)}, y''_N = y_N^{(2)}, \ldots, x'_N, x''_N, \ldots, j'_N, j''_N, \ldots$$

are obtained on replacing N_k in the definitions in the sections 18.4.3 and 18.4.4 by $N'_k, N''_k, \ldots$; they can be entered as

$$\begin{array}{lcl} \texttt{EY(}N\texttt{,}d\texttt{)} & = & y_N^{(d)} \\ \texttt{EX(}N\texttt{,}d\texttt{)} & = & x_N^{(d)} \\ & \ldots & \\ \texttt{EJ(}N\texttt{,}d\texttt{)} & = & j_N^{(d)} \end{array}$$

18.4.6 AtlasIrrationality

▷ `AtlasIrrationality(`*irratname*`)` (function)

Let *irratname* be a string that describes an irrational value as a linear combination in terms of the atomic irrationalities introduced in the sections 18.4.1, 18.4.2, 18.4.3, 18.4.4. These irrational values are defined in [CCN+85, Chapter 6, Section 10], and the following description is mainly copied from there. If q_N is such a value (e.g. y''_{24}) then linear combinations of algebraic conjugates of q_N are abbreviated as in the following examples:

`2qN+3&5-4&7+&9`	means	$2q_N + 3q_N^{*5} - 4q_N^{*7} + q_N^{*9}$
`4qN&3&5&7-3&4`	means	$4(q_N + q_N^{*3} + q_N^{*5} + q_N^{*7}) - 3q_N^{*11}$
`4qN*3&5+&7`	means	$4(q_N^{*3} + q_N^{*5}) + q_N^{*7}$

To explain the "ampersand" syntax in general we remark that "&k" is interpreted as q_N^{*k}, where q_N is the most recently named atomic irrationality, and that the scope of any premultiplying coefficient is broken by a $+$ or $-$ sign, but not by & or $*k$. The algebraic conjugations indicated by the ampersands apply directly to the *atomic* irrationality q_N, even when, as in the last example, q_N first appears with another conjugacy $*k$.

Example

```
gap> AtlasIrrationality( "b7*3" );
E(7)^3+E(7)^5+E(7)^6
gap> AtlasIrrationality( "y'''24" );
E(24)-E(24)^19
gap> AtlasIrrationality( "-3y'''24*13&5" );
3*E(8)-3*E(8)^3
gap> AtlasIrrationality( "3y'''24*13-2&5" );
-3*E(24)-2*E(24)^11+2*E(24)^17+3*E(24)^19
gap> AtlasIrrationality( "3y'''24*13-&5" );
-3*E(24)-E(24)^11+E(24)^17+3*E(24)^19
gap> AtlasIrrationality( "3y'''24*13-4&5&7" );
-7*E(24)-4*E(24)^11+4*E(24)^17+7*E(24)^19
gap> AtlasIrrationality( "3y'''24&7" );
6*E(24)-6*E(24)^19
```

18.5 Galois Conjugacy of Cyclotomics

18.5.1 GaloisCyc (for a cyclotomic)

▷ GaloisCyc(*cyc*, *k*) (operation)
▷ GaloisCyc(*list*, *k*) (operation)

For a cyclotomic *cyc* and an integer *k*, GaloisCyc returns the cyclotomic obtained by raising the roots of unity in the Zumbroich basis representation of *cyc* to the *k*-th power. If *k* is coprime to the integer *n*, GaloisCyc(., *k*) acts as a Galois automorphism of the *n*-th cyclotomic field (see 60.4); to get the Galois automorphisms themselves, use GaloisGroup (58.3.1).

The *complex conjugate* of *cyc* is GaloisCyc(*cyc*, -1), which can also be computed using ComplexConjugate (18.5.2).

For a list or matrix *list* of cyclotomics, GaloisCyc returns the list obtained by applying GaloisCyc to the entries of *list*.

18.5.2 ComplexConjugate

▷ ComplexConjugate(*z*) (attribute)
▷ RealPart(*z*) (attribute)
▷ ImaginaryPart(*z*) (attribute)

For a cyclotomic number *z*, ComplexConjugate returns GaloisCyc(*z*, -1), see GaloisCyc (18.5.1). For a quaternion $z = c_1e+c_2i+c_3j+c_4k$, ComplexConjugate returns $c_1e-c_2i-c_3j-c_4k$, see IsQuaternion (62.8.8).

When ComplexConjugate is called with a list then the result is the list of return values of ComplexConjugate for the list entries in the corresponding positions.

When ComplexConjugate is defined for an object *z* then RealPart and ImaginaryPart return (*z* + ComplexConjugate(*z*)) / 2 and (*z* - ComplexConjugate(*z*)) / 2 i, respectively, where i denotes the corresponding imaginary unit.

Example

```
gap> GaloisCyc( E(5) + E(5)^4, 2 );
E(5)^2+E(5)^3
gap> GaloisCyc( E(5), -1 );           # the complex conjugate
E(5)^4
gap> GaloisCyc( E(5) + E(5)^4, -1 );  # this value is real
E(5)+E(5)^4
gap> GaloisCyc( E(15) + E(15)^4, 3 );
E(5)+E(5)^4
gap> ComplexConjugate( E(7) );
E(7)^6
```

18.5.3 StarCyc

▷ StarCyc(*cyc*) (function)

If the cyclotomic *cyc* is an irrational element of a quadratic extension of the rationals then StarCyc returns the unique Galois conjugate of *cyc* that is different from *cyc*, otherwise fail is returned. In the first case, the return value is often called $cyc*$ (see 71.13).

Example

```
gap> StarCyc( EB(5) ); StarCyc( E(5) );
E(5)^2+E(5)^3
fail
```

18.5.4 Quadratic

▷ `Quadratic(cyc)` (function)

Let *cyc* be a cyclotomic integer that lies in a quadratic extension field of the rationals. Then we have $cyc = (a + b\sqrt{n})/d$, for integers a, b, n, d, such that d is either 1 or 2. In this case, `Quadratic` returns a record with the components `a`, `b`, `root`, `d`, `ATLAS`, and `display`; the values of the first four are a, b, n, and d, the `ATLAS` value is a (not necessarily shortest) representation of *cyc* in terms of the Atlas irrationalities $b_{|n|}$, $i_{|n|}$, $r_{|n|}$, and the `display` value is a string that expresses *cyc* in GAP notation, corresponding to the value of the `ATLAS` component.

If *cyc* is not a cyclotomic integer or does not lie in a quadratic extension field of the rationals then `fail` is returned.

If the denominator d is 2 then necessarily n is congruent to 1 modulo 4, and r_n, i_n are not possible; we have *cyc* `= x + y * EB( root )` with `y = b`, `x = ( a + b ) / 2`.

If $d = 1$, we have the possibilities $i_{|n|}$ for $n < -1$, $a + b * i$ for $n = -1$, $a + b * r_n$ for $n > 0$. Furthermore if n is congruent to 1 modulo 4, also $cyc = (a+b) + 2 * b * b_{|n|}$ is possible; the shortest string of these is taken as the value for the component `ATLAS`.

Example

```
gap> Quadratic( EB(5) ); Quadratic( EB(27) );
rec( ATLAS := "b5", a := -1, b := 1, d := 2,
  display := "(-1+Sqrt(5))/2", root := 5 )
rec( ATLAS := "1+3b3", a := -1, b := 3, d := 2,
  display := "(-1+3*Sqrt(-3))/2", root := -3 )
gap> Quadratic(0); Quadratic( E(5) );
rec( ATLAS := "0", a := 0, b := 0, d := 1, display := "0", root := 1 )
fail
```

18.5.5 GaloisMat

▷ `GaloisMat(mat)` (attribute)

Let *mat* be a matrix of cyclotomics. `GaloisMat` calculates the complete orbits under the operation of the Galois group of the (irrational) entries of *mat*, and the permutations of rows corresponding to the generators of the Galois group.

If some rows of *mat* are identical, only the first one is considered for the permutations, and a warning will be printed.

`GaloisMat` returns a record with the components `mat`, `galoisfams`, and `generators`.

`mat` a list with initial segment being the rows of *mat* (*not* shallow copies of these rows); the list consists of full orbits under the action of the Galois group of the entries of *mat* defined above. The last rows in the list are those not contained in *mat* but must be added in order to complete the orbits; so if the orbits were already complete, *mat* and `mat` have identical rows.

`galoisfams`
: a list that has the same length as the `mat` component, its entries are either 1, 0, -1, or lists.

  `galoisfams[i] = 1`
  : means that `mat[i]` consists of rationals, i.e., `[ mat[i] ]` forms an orbit;

  `galoisfams[i] = -1`
  : means that `mat[i]` contains unknowns (see Chapter 74); in this case `[ mat[i] ]` is regarded as an orbit, too, even if `mat[i]` contains irrational entries;

  `galoisfams[i] =` $[l_1, l_2]$
  : (a list) means that `mat[i]` is the first element of its orbit in `mat`, l_1 is the list of positions of rows that form the orbit, and l_2 is the list of corresponding Galois automorphisms (as exponents, not as functions); so we have `mat`$[l_1[j]][k] =$ `GaloisCyc( mat`$[i][k], l_2[j]$ `);`

  `galoisfams[i] = 0`
  : means that `mat[i]` is an element of a nontrivial orbit but not the first element of it.

`generators`
: a list of permutations generating the permutation group corresponding to the action of the Galois group on the rows of `mat`.

Example

```
gap> GaloisMat( [ [ E(3), E(4) ] ] );
rec( galoisfams := [ [ [ 1, 2, 3, 4 ], [ 1, 7, 5, 11 ] ], 0, 0, 0 ],
  generators := [ (1,2)(3,4), (1,3)(2,4) ],
  mat := [ [ E(3), E(4) ], [ E(3), -E(4) ], [ E(3)^2, E(4) ],
      [ E(3)^2, -E(4) ] ] )
gap> GaloisMat( [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ] ] );
rec( galoisfams := [ 1, [ [ 2, 3 ], [ 1, 2 ] ], 0 ],
  generators := [ (2,3) ],
  mat := [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ] )
```

18.5.6 RationalizedMat

▷ `RationalizedMat(`*mat*`)` (attribute)

returns the list of rationalized rows of *mat*, which must be a matrix of cyclotomics. This is the set of sums over orbits under the action of the Galois group of the entries of *mat* (see `GaloisMat` (18.5.5)), so the operation may be viewed as a kind of trace on the rows.

Note that no two rows of *mat* should be equal.

Example

```
gap> mat:= [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ];;
gap> RationalizedMat( mat );
[ [ 1, 1, 1 ], [ 2, -1, -1 ] ]
```

18.6 Internally Represented Cyclotomics

The implementation of an *internally represented cyclotomic* is based on a list of length equal to its conductor. This means that the internal representation of a cyclotomic does *not* refer to the smallest number field but the smallest *cyclotomic* field containing it. The reason for this is the wish to reflect

the natural embedding of two cyclotomic fields into a larger one that contains both. With such embeddings, it is easy to construct the sum or the product of two arbitrary cyclotomics (in possibly different fields) as an element of a cyclotomic field.

The disadvantage of this approach is that the arithmetical operations are quite expensive, so the use of internally represented cyclotomics is not recommended for doing arithmetics over number fields, such as calculations with matrices of cyclotomics. But internally represented cyclotomics are good enough for dealing with irrationalities in character tables (see chapter 71).

For the representation of cyclotomics one has to recall that the n-th cyclotomic field $\mathbb{Q}(e_n)$ is a vector space of dimension $\varphi(n)$ over the rationals where φ denotes Euler's phi-function (see `Phi` (15.2.2)).

A special integral basis of cyclotomic fields is chosen that allows one to easily convert arbitrary sums of roots of unity into the basis, as well as to convert a cyclotomic represented w.r.t. the basis into the smallest possible cyclotomic field. This basis is accessible in GAP, see 60.3 for more information and references.

Note that the set of all n-th roots of unity is linearly dependent for $n > 1$, so multiplication is *not* the multiplication of the group ring $\mathbb{Q}\langle e_n\rangle$; given a $\mathbb{Q}$-basis of $\mathbb{Q}(e_n)$ the result of the multiplication (computed as multiplication of polynomials in e_n, using $(e_n)^n = 1$) will be converted to the basis.

Example

```
gap> E(5) * E(5)^2; ( E(5) + E(5)^4 ) * E(5)^2;
E(5)^3
E(5)+E(5)^3
gap> ( E(5) + E(5)^4 ) * E(5);
-E(5)-E(5)^3-E(5)^4
```

An internally represented cyclotomic is always represented in the smallest cyclotomic field it is contained in. The internal coefficients list coincides with the external representation returned by `ExtRepOfObj` (18.1.12).

To avoid calculations becoming unintentionally very long, or consuming very large amounts of memory, there is a limit on the conductor of internally represented cyclotomics, by default set to one million. This can be raised (although not lowered) using `SetCyclotomicsLimit` (18.6.1) and accessed using `GetCyclotomicsLimit` (18.6.1). The maximum value of the limit is $2^{28}-1$ on 32 bit systems, and 2^{32} on 64 bit systems. So the maximal cyclotomic field implemented in GAP is not really the field $\mathbb{Q}^{ab}$.

It should be emphasized that one disadvantage of representing a cyclotomic in the smallest *cyclotomic* field (and not in the smallest field) is that arithmetic operations in a fixed small extension field of the rational number field are comparatively expensive. For example, take a prime integer p and suppose that we want to work with a matrix group over the field $\mathbb{Q}(\sqrt{p})$. Then each matrix entry could be described by two rational coefficients, whereas the representation in the smallest cyclotomic field requires $p-1$ rational coefficients for each entry. So it is worth thinking about using elements in a field constructed with `AlgebraicExtension` (67.1.1) when natural embeddings of cyclotomic fields are not needed.

18.6.1 SetCyclotomicsLimit

▷ `SetCyclotomicsLimit(`*newlimit*`)` (function)

▷ `GetCyclotomicsLimit()` (function)

`GetCyclotomicsLimit` returns the current limit on conductors of internally represented cyclotomic numbers

`SetCyclotomicsLimit` can be called to increase the limit on conductors of internally represented cyclotomic numbers. Note that computing in large cyclotomic fields using this representation can be both slow and memory-consuming, and that other approaches may be better for some problems. See 18.6.

Chapter 19

Floats

Starting with version 4.5, GAP has built-in support for floating-point numbers in machine format, and allows package to implement arbitrary-precision floating-point arithmetic in a uniform manner. For now, one such package, Float exists, and is based on the arbitrary-precision routines in mpfr.

A word of caution: GAP deals primarily with algebraic objects, which can be represented exactly in a computer. Numerical imprecision means that floating-point numbers do not form a ring in the strict GAP sense, because addition is in general not associative (`(1.0e-100+1.0)-1.0` is not the same as `1.0e-100+(1.0-1.0)`, in the default precision setting).

Most algorithms in GAP which require ring elements will therefore not be applicable to floating-point elements. In some cases, such a notion would not even make any sense (what is the greatest common divisor of two floating-point numbers?)

19.1 A sample run

Floating-point numbers can be input into GAP in the standard floating-point notation:

Example

```
gap> 3.14;
3.14
gap> last^2/6;
1.64327
gap> h := 6.62606896e-34;
6.62607e-34
gap> pi := 4*Atan(1.0);
3.14159
gap> hbar := h/(2*pi);
1.05457e-34
```

Floating-point numbers can also be created using `Float`, from strings or rational numbers; and can be converted back using `String,Rat,Int`.

GAP allows rational and floating-point numbers to be mixed in the elementary operations `+,-,*,/`. However, floating-point numbers and rational numbers may not be compared. Conversions are performed using the creator `Float`:

Example

```
gap> Float("3.1416");
3.1416
```

```
gap> Float(355/113);
3.14159
gap> Rat(last);
355/113
gap> Rat(0.33333);
1/3
gap> Int(1.e10);
10000000000
gap> Int(1.e20);
100000000000000000000
gap> Int(1.e30);
1000000000000000019884624838656
```

19.2 Methods

Floating-point numbers may be directly input, as in any usual mathematical software or language; with the exception that every floating-point number must contain a decimal digit. Therefore `.1`, `.1e1`, `-.999` etc. are all valid GAP inputs.

Floating-point numbers so entered in GAP are stored as strings. They are converted to floating-point when they are first used. This means that, if the floating-point precision is increased, the constants are reevaluated to fit the new format.

Floating-point numbers may be followed by an underscore, as in `1._`. This means that they are to be immediately converted to the current floating-point format. The underscore may be followed by a single letter, which specifies which format/precision to use. By default, GAP has a single floating-point handler, with fixed (53 bits) precision, and its format specifier is `'l'` as in `1._l`. Higher-precision floating-point computations is available via external packages; float for example.

A record, `FLOAT` (19.2.6), contains all relevant constants for the current floating-point format; see its documentation for details. Typical fields are `FLOAT.MANT_DIG=53`, the constant `FLOAT.VIEW_DIG=6` specifying the number of digits to view, and `FLOAT.PI` for the constant π. The constants have the same name as their C counterparts, except for the missing initial `DBL_` or `M_`.

Floating-point numbers may be created using the single function `Float` (19.2.7), which accepts as arguments rational, string, or floating-point numbers. Floating-point numbers may also be created, in any floating-point representation, using `NewFloat` (19.2.7) as in `NewFloat(IsIEEE754FloatRep,355/113)`, by supplying the category filter of the desired new floating-point number; or using `MakeFloat` (19.2.7) as in `NewFloat(1.0,355/113)`, by supplying a sample floating-point number.

Floating-point numbers may also be converted to other GAP formats using the usual commands `Int` (14.2.3), `Rat` (17.2.6), `String` (27.6.6).

Exact conversion to and from floating-point format may be done using external representations. The "external representation" of a floating-point number `x` is a pair `[m,e]` of integers, such that `x=m*2^(-1+e-LogInt(AbsInt(m),2))`. Conversion to and from external representation is performed as usual using `ExtRepOfObj` (79.16.1) and `ObjByExtRep` (79.16.1):

Example

```
gap> ExtRepOfObj(3.14);
[ 7070651414971679, 2 ]
gap> ObjByExtRep(IEEE754FloatsFamily,last);
3.14
```

Computations with floating-point numbers never raise any error. Division by zero is allowed, and produces a signed infinity. Illegal operations, such as `0./0.`, produce `NaN`'s (not-a-number); this is the only floating-point number `x` such that `not EqFloat(x+0.0,x)`.

The IEEE754 standard requires `NaN` to be non-equal to itself. On the other hand, GAP requires every object to be equal to itself. To respect the IEEE754 standard, the function `EqFloat` (19.2.2) should be used instead of =.

The category a floating-point belongs to can be checked using the filters `IsFinite` (30.4.2), `IsPInfinity` (19.2.5), `IsNInfinity` (19.2.5), `IsXInfinity` (19.2.5), `IsNaN` (19.2.5).

Comparisons between floating-point numbers and rationals are explicitly forbidden. The rationale is that objects belonging to different families should in general not be comparable in GAP. Floating-point numbers are also approximations of real numbers, and don't follow the same rules; consider for example, using the default GAP implementation of floating-point numbers,

Example

```
gap> 1.0/3.0 = Float(1/3);
true
gap> (1.0/3.0)^5 = Float((1/3)^5);
false
```

19.2.1 Mathematical operations

▷ `Cos(`*x*`)` (operation)
▷ `Sin(`*x*`)` (operation)
▷ `SinCos(`*x*`)` (operation)
▷ `Tan(`*x*`)` (operation)
▷ `Sec(`*x*`)` (operation)
▷ `Csc(`*x*`)` (operation)
▷ `Cot(`*x*`)` (operation)
▷ `Asin(`*x*`)` (operation)
▷ `Acos(`*x*`)` (operation)
▷ `Atan(`*x*`)` (operation)
▷ `Atan2(`*y*`, `*x*`)` (operation)
▷ `Cosh(`*x*`)` (operation)
▷ `Sinh(`*x*`)` (operation)
▷ `Tanh(`*x*`)` (operation)
▷ `Sech(`*x*`)` (operation)
▷ `Csch(`*x*`)` (operation)
▷ `Coth(`*x*`)` (operation)
▷ `Asinh(`*x*`)` (operation)
▷ `Acosh(`*x*`)` (operation)
▷ `Atanh(`*x*`)` (operation)
▷ `Log(`*x*`)` (operation)
▷ `Log2(`*x*`)` (operation)
▷ `Log10(`*x*`)` (operation)
▷ `Log1p(`*x*`)` (operation)
▷ `Exp(`*x*`)` (operation)
▷ `Exp2(`*x*`)` (operation)
▷ `Exp10(`*x*`)` (operation)

▷ Expm1(*x*) (operation)
▷ Cuberoot(*x*) (operation)
▷ Square(*x*) (operation)
▷ Hypothenuse(*x*, *y*) (operation)
▷ Ceil(*x*) (operation)
▷ Floor(*x*) (operation)
▷ Round(*x*) (operation)
▷ Trunc(*x*) (operation)
▷ Frac(*x*) (operation)
▷ SignFloat(*x*) (operation)
▷ Argument(*x*) (operation)
▷ Erf(*x*) (operation)
▷ Zeta(*x*) (operation)
▷ Gamma(*x*) (operation)
▷ ComplexI(*x*) (operation)

Usual mathematical functions.

19.2.2 EqFloat

▷ EqFloat(*x*, *y*) (operation)

Returns: Whether the floateans *x* and *y* are equal

This function compares two floating-point numbers, and returns `true` if they are equal, and `false` otherwise; with the exception that `NaN` is always considered to be different from itself.

19.2.3 PrecisionFloat

▷ PrecisionFloat(*x*) (operation)

Returns: The precision of *x*

This function returns the precision, counted in number of binary digits, of the floating-point number *x*.

19.2.4 Interval operations

▷ Sup(*interval*) (operation)
▷ Inf(*interval*) (operation)
▷ Mid(*interval*) (operation)
▷ AbsoluteDiameter(*interval*) (operation)
▷ RelativeDiameter(*interval*) (operation)
▷ Overlaps(*interval1*, *interval2*) (operation)
▷ IsDisjoint(*interval1*, *interval2*) (operation)
▷ IncreaseInterval(*interval*, *delta*) (operation)
▷ BlowupInterval(*interval*, *ratio*) (operation)
▷ BisectInterval(*interval*) (operation)

Most are self-explanatory. `BlowupInterval` returns an interval with same midpoint but relative diameter increased by *ratio*; `IncreaseInterval` returns an interval with same midpoint but ab-

solute diameter increased by *delta*; BisectInterval returns a list of two intervals whose union equals *interval*.

19.2.5 IsPInfinity

▷ IsPInfinity(*x*) (property)
▷ IsNInfinity(*x*) (property)
▷ IsXInfinity(*x*) (property)
▷ IsFinite(*x*) (property)
▷ IsNaN(*x*) (property)

Returns true if the floating-point number *x* is respectively $+\infty$, $-\infty$, $\pm\infty$, finite, or 'not a number', such as the result of 0.0/0.0.

19.2.6 FLOAT (constants)

▷ FLOAT (global variable)

This record contains useful floating-point constants:

DECIMAL_DIG
Maximal number of useful digits;

DIG Number of significant digits;

VIEW_DIG
Number of digits to print in short view;

EPSILON
Smallest number such that $1 \neq 1 + \varepsilon$;

MANT_DIG
Number of bits in the mantissa;

MAX
Maximal representable number;

MAX_10_EXP
Maximal decimal exponent;

MAX_EXP
Maximal binary exponent;

MIN
Minimal positive representable number;

MIN_10_EXP
Minimal decimal exponent;

MIN_EXP
Minimal exponent;

INFINITY
Positive infinity;

NINFINITY
Negative infinity;

NAN
Not-a-number,

as well as mathematical constants `E`, `LOG2E`, `LOG10E`, `LN2`, `LN10`, `PI`, `PI_2`, `PI_4`, `1_PI`, `2_PI`, `2_SQRTPI`, `SQRT2`, `SQRT1_2`.

19.2.7 Float

▷ `Float(obj)` (operation)
▷ `NewFloat(filter, obj)` (operation)
▷ `MakeFloat(sample, obj, obj)` (operation)

Returns: A new floating-point number, based on *obj*

This function creates a new floating-point number.

If *obj* is a rational number, the created number is created with sufficient precision so that the number can (usually) be converted back to the original number (see `Rat` (**Reference: Rat**) and `Rat` (17.2.6)). For an integer, the precision, if unspecified, is chosen sufficient so that `Int(Float(obj))=obj` always holds, but at least 64 bits.

obj may also be a string, which may be of the form `"3.14e0"` or `".314e1"` or `".314@1"` etc.

An option may be passed to specify, it bits, a desired precision. The format is `Float("3.14":PrecisionFloat:=1000)` to create a 1000-bit approximation of 3.14.

In particular, if *obj* is already a floating-point number, then `Float(obj:PrecisionFloat:=prec)` creates a copy of *obj* with a new precision. prec

19.2.8 Rat (for floats)

▷ `Rat(f)` (operation)

Returns: A rational approximation to *f*

This command constructs a rational approximation to the floating-point number *f*. Of course, it is not guaranteed to return the original rational number *f* was created from, though it returns the most 'reasonable' one given the precision of *f*.

Two options control the precision of the rational approximation: In the form `Rat(f:maxdenom:=md,maxpartial:=mp)`, the rational returned is such that the denominator is at most *md* and the partials in its continued fraction expansion are at most *mp*. The default values are `maxpartial:=10000` and `maxdenom:=2^(precision/2)`.

19.2.9 SetFloats

▷ `SetFloats(rec[, bits][, install])` (function)

Installs a new interface to floating-point numbers in GAP, optionally with a desired precision *bits* in binary digits. The last optional argument *install* is a boolean value; if false, it only installs the eager handler and the precision for the floateans, without making them the default.

19.3 High-precision-specific methods

GAP provides a mechanism for packages to implement new floating-point numerical interfaces. The following describes that mechanism, actual examples of packages are documented separately.

A package must create a record with fields (all optional)

creator
a function converting strings to floating-point;

eager
a character allowing immediate conversion to floating-point;

objbyextrep
a function creating a floating-point number out of a list `[mantissa,exponent]`;

filter
a filter for the new floating-point objects;

constants
a record containing numerical constants, such as `MANT_DIG`, `MAX`, `MIN`, `NAN`.

The package must install methods `Int`, `Rat`, `String` for its objects, and creators `NewFloat(filter,IsRat)`, `NewFloat(IsString)`.

It must then install methods for all arithmetic and numerical operations: `PLUS`, `Exp`, ...

The user chooses that implementation by calling `SetFloats` (19.2.9) with the record as argument, and with an optional second argument requesting a precision in binary digits.

19.4 Complex arithmetic

Complex arithmetic may be implemented in packages, and is present in float. Complex numbers are treated as usual numbers; they may be input with an extra "i" as in `-0.5+0.866i`.

Methods should then be implemented for `Norm`, `RealPart`, `ImaginaryPart`, `ComplexConjugate`, ...

19.5 Interval-specific methods

Interval arithmetic may also be implemented in packages. Intervals are in fact efficient implementations of sets of real numbers. The only non-trivial issue is how they should be compared. The standard `EQ` tests if the intervals are equal; however, it is usually more useful to know if intervals overlap, or are disjoint, or are contained in each other. The methods provided by the package should include `Sup,Inf,Mid,DiameterOfInterval,Overlaps,IsSubset,IsDisjoint`.

Note the usual convention that intervals are compared as in $[a,b] \leq [c,d]$ if and only if $a \leq c$ and $b \leq d$.

Chapter 20

Booleans

The two main *boolean* values are `true` and `false`. They stand for the *logical* values of the same name. They appear as values of the conditions in `if`-statements and `while`-loops. Booleans are also important as return values of *filters* (see 13.2) such as `IsFinite` (30.4.2) and `IsBool` (20.1.1). Note that it is a convention that the name of a function that returns `true` or `false` according to the outcome, starts with `Is`.

For technical reasons, also the value `fail` (see 20.2) is regarded as a boolean.

20.1 IsBool (Filter)

20.1.1 IsBool

▷ `IsBool(`*`obj`*`)` (Category)

tests whether *obj* is `true`, `false` or `fail`.

Example

```
gap> IsBool( true );  IsBool( false );  IsBool( 17 );
true
true
false
```

20.2 Fail (Variable)

20.2.1 fail

▷ `fail` (global variable)

The value `fail` is used to indicate situations when an operation could not be performed for the given arguments, either because of shortcomings of the arguments or because of restrictions in the implementation or computability. So for example `Position` (21.16.1) will return `fail` if the point searched for is not in the list.

`fail` is simply an object that is different from every other object than itself.

For technical reasons, `fail` is a boolean value. But note that `fail` cannot be used to form boolean expressions with `and`, `or`, and `not` (see 20.4 below), and `fail` cannot appear in boolean lists (see Chapter 22).

20.3 Comparisons of Booleans

20.3.1 Equality and inequality of Booleans

bool1 = *bool2*
bool1 <> *bool2*

The equality operator = evaluates to `true` if the two boolean values *bool1* and *bool2* are equal, i.e., both are `true` or both are `false` or both `fail`, and `false` otherwise. The inequality operator <> evaluates to `true` if the two boolean values *bool1*, *bool2* are different, and `false` otherwise. This operation is also called the *exclusive or*, because its value is `true` if exactly one of *bool1* or *bool2* is `true`.

You can compare boolean values with objects of other types. Of course they are never equal.

Example

```
gap> true = false;
false
gap> false = (true = fail);
true
gap> true <> 17;
true
```

20.3.2 Ordering of Booleans

bool1 < *bool2*

The ordering of boolean values is defined by `true` < `false` < `fail`. For the comparison of booleans with other GAP objects, see Section 4.12.

Example

```
gap> true < false;  fail >= false;
true
true
```

20.4 Operations for Booleans

The following boolean operations are only applicable to `true` and `false`.

20.4.1 Logical disjunction

bool1 `or` *bool2*

The logical operator `or` evaluates to `true` if at least one of the two boolean operands *bool1* and *bool2* is `true`, and to `false` otherwise.

`or` first evaluates *bool1*. If the value is neither `true` nor `false` an error is signalled. If the value is `true`, then `or` returns `true` *without* evaluating *bool2*. If the value is `false`, then `or` evaluates *bool2*. Again, if the value is neither `true` nor `false` an error is signalled. Otherwise `or` returns the value of *bool2*. This *short-circuited* evaluation is important if the value of *bool1* is `true` and evaluation of *bool2* would take much time or cause an error.

`or` is associative, i.e., it is allowed to write *b1* `or` *b2* `or` *b3*, which is interpreted as (*b1* `or` *b2*) `or` *b3*. `or` has the lowest precedence of the logical operators. All logical operators have lower precedence than the comparison operators =, <, `in`, etc.

Example

```
gap> true or false;
true
gap> false or false;
false
gap> i := -1;; l := [1,2,3];;
gap> if i <= 0 or l[i] = false then     # this does not cause an error,
>    Print("aha\n"); fi;                # because 'l[i]' is not evaluated
aha
```

20.4.2 Logical conjunction

bool1 and *bool2*
fil1 and *fil2*

The logical operator and evaluates to true if both boolean operands *bool1*, *bool2* are true, and to false otherwise.

and first evaluates *bool1*. If the value is neither true nor false an error is signalled. If the value is false, then and returns false *without* evaluating *bool2*. If the value is true, then and evaluates *bool2*. Again, if the value is neither true nor false an error is signalled. Otherwise and returns the value of *bool2*. This *short-circuited* evaluation is important if the value of *bool1* is false and evaluation of *bool2* would take much time or cause an error.

and is associative, i.e., it is allowed to write *b1* and *b2* and *b3*, which is interpreted as (*b1* and *b2*) and *b3*. and has higher precedence than the logical or operator, but lower than the unary logical not operator. All logical operators have lower precedence than the comparison operators =, <, in, etc.

Example

```
gap> true and false;
false
gap> true and true;
true
gap> false and 17;  # does not cause error, because 17 is never looked at
false
```

and can also be applied to filters. It returns a filter that when applied to some argument *x*, tests *fil1*(x) and *fil2*(x).

Example

```
gap> andfilt:= IsPosRat and IsInt;;
gap> andfilt( 17 );  andfilt( 1/2 );
true
false
```

20.4.3 Logical negation

not *bool*

The logical operator not returns true if the boolean value *bool* is false, and true otherwise. An error is signalled if *bool* does not evaluate to true or false.

not has higher precedence than the other logical operators, or and and. All logical operators have lower precedence than the comparison operators =, <, in, etc.

Example

```
gap> true and false;
false
gap> not true;
false
gap> not false;
true
```

Chapter 21

Lists

Lists are the most important way to treat objects together. A *list* arranges objects in a definite order. So each list implies a partial mapping from the integers to the elements of the list. I.e., there is a first element of a list, a second, a third, and so on. Lists can occur in mutable or immutable form, see 12.6 for the concept of mutability, and 21.7 for the case of lists.

This chapter deals mainly with the aspect of lists in GAP as *data structures*. Chapter 30 tells more about the *collection* aspect of certain lists, and more about lists as *arithmetic objects* can be found in the chapters 23 and 24.

Lists are used to implement ranges (see 21.22), sets (see 21.19), strings (see 27), row vectors (see 23), and matrices (see 24); Boolean lists (see 22) are a further special kind of lists.

Several operations for lists, such as `Intersection` (30.5.2) and `Random` (30.7.1), will be described in Chapter 30, in particular see 30.3.

21.1 List Categories

A list can be written by writing down the elements in order between square brackets `[`, `]`, and separating them with commas `,`. An *empty list*, i.e., a list with no elements, is written as `[]`.

Example

```
gap> [ 1, 2, 3 ];              # a list with three elements
[ 1, 2, 3 ]
gap> [ [], [ 1 ], [ 1, 2 ] ];  # a list may contain other lists
[ [  ], [ 1 ], [ 1, 2 ] ]
```

Each list constructed this way is mutable (see 12.6).

21.1.1 IsList

▷ `IsList(`*obj*`)` (Category)

tests whether *obj* is a list.

Example

```
gap> IsList( [ 1, 3, 5, 7 ] );  IsList( 1 );
true
false
```

21.1.2 IsDenseList

▷ IsDenseList(*obj*) (Category)

A list is *dense* if it has no holes, i.e., contains an element at every position up to the length. It is absolutely legal to have lists with holes. They are created by leaving the entry between the commas empty. Holes at the end of a list are ignored. Lists with holes are sometimes convenient when the list represents a mapping from a finite, but not consecutive, subset of the positive integers.

Example

```
gap> IsDenseList( [ 1, 2, 3 ] );
true
gap> l := [ , 4, 9,, 25,, 49,,,, 121 ];;  IsDenseList( l );
false
gap> l[3];
9
gap> l[4];
List Element: <list>[4] must have an assigned value
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' after assigning a value to continue
brk> l[4] := 16;;  # assigning a value
brk> return;       # to escape the break-loop
16
gap>
```

Observe that requesting the value of `l[4]`, which was not assigned, caused the entry of a `break`-loop (see Section 6.4). After assigning a value and typing `return;`, GAP is finally able to comply with our request (by responding with `16`).

21.1.3 IsHomogeneousList

▷ IsHomogeneousList(*obj*) (Category)

returns `true` if *obj* is a list and it is homogeneous, and `false` otherwise.

A *homogeneous* list is a dense list whose elements lie in the same family (see 13.1). The empty list is homogeneous but not a collection (see 30), a nonempty homogeneous list is also a collection.

Example

```
gap> IsHomogeneousList( [ 1, 2, 3 ] );  IsHomogeneousList( [] );
true
true
gap> IsHomogeneousList( [ 1, false, () ] );
false
```

21.1.4 IsTable

▷ IsTable(*obj*) (Category)

A *table* is a nonempty list of homogeneous lists which lie in the same family. Typical examples of tables are matrices (see 24).

Example

```
gap> IsTable( [ [ 1, 2 ], [ 3, 4 ] ] );    # in fact a matrix
true
gap> IsTable( [ [ 1 ], [ 2, 3 ] ] );       # not rectangular but a table
true
gap> IsTable( [ [ 1, 2 ], [ () , (1,2) ] ] );  # not homogeneous
false
```

21.1.5 IsRectangularTable

▷ IsRectangularTable(*list*) (property)

A list lies in IsRectangularTable when it is nonempty and its elements are all homogeneous lists of the same family and the same length.

21.1.6 IsConstantTimeAccessList

▷ IsConstantTimeAccessList(*list*) (Category)

This category indicates whether the access to each element of the list *list* will take roughly the same time. This is implied for example by IsList and IsInternalRep, so all strings, Boolean lists, ranges, and internally represented plain lists are in this category.

But also other enumerators (see 21.23) can lie in this category if they guarantee constant time access to their elements.

21.2 Basic Operations for Lists

The basic operations for lists are element access (see 21.3), assignment of elements to a list (see 21.4), fetching the length of a list (see Length (21.17.5)), the test for a hole at a given position, and unbinding an element at a given position (see 21.5).

The term basic operation means that each other list operation can be formulated in terms of the basic operations. (But note that usually a more efficient method than this one is implemented.)

Any GAP object *list* in the category IsList (21.1.1) is regarded as a list, and if methods for the basic list operations are installed for *list* then *list* can be used also for the other list operations.

For internally represented lists, kernel methods are provided for the basic list operations. For other lists, it is possible to install appropriate methods for these operations. This permits the implementation of lists that do not need to store all list elements (see also 21.23); for example, the elements might be described by an algorithm, such as the elements list of a group. For this reduction of space requirements, however, a price in access time may have to be paid (see ConstantTimeAccessList (21.17.6)).

21.2.1 \[\]

▷ \[\](*list, pos*) (operation)
▷ IsBound\[\](*list, pos*) (operation)
▷ \[\]\:\=(*list, pos, val*) (operation)

▷ Unbind\[\](*list, pos*) (operation)

These operations implement element access, test for element boundedness, list element assignment, and removal of the element at position *pos*. In all cases, the index *pos* must be a positive integer.

Note that the special characters [,], :, and = must be escaped with a backslash \ (see 4.3); so \[\] (21.2.1) denotes the operation for element access in a list, whereas [] denotes an empty list. (Maybe the variable names involving special characters look strange, but nevertheless they are quite suggestive.)

\[\](*list*, *pos*) is equivalent to *list*[*pos*], which clearly will usually be preferred; the former is useful mainly if one wants to access the operation itself, for example if one wants to install a method for element access in a special kind of lists.

Similarly, IsBound\[\] (21.2.1) is used explicitly mainly in method installations. In other situations, one can simply call IsBound (21.5.1), which then delegates to IsBound\[\] (21.2.1) if the first argument is a list, and to IsBound\. (29.7.3) if the first argument is a record.

Analogous statements hold for \[\]\:\= (21.2.1) and Unbind\[\] (21.2.1).

21.3 List Elements

list[*pos*]

The above construct evaluates to the *pos*-th element of the list *list*, where *pos* must be a positive integer. List indexing is done with origin 1, i.e., the first element of the list is the element at position 1.

Example

```
gap> l := [ 2, 3, 5, 7, 11, 13 ];;  l[1];  l[2];  l[6];
2
3
13
```

If *list* is not a list, or *pos* does not evaluate to a positive integer, or *list*[*pos*] is unbound an error is signalled.

list{ *poss* }

The above construct evaluates to a new list *new* whose first element is *list*[*poss*[1]], whose second element is *list*[*poss*[2]], and so on. *poss* must be a dense list of positive integers. However, it does not need to be sorted and may contain duplicate elements. If for any *i*, *list*[*poss*[*i*]] is unbound, an error is signalled.

Example

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[4..6]};  l{[1,7,1,8]};
[ 7, 11, 13 ]
[ 2, 17, 2, 19 ]
```

The result is a *new* list, that is not identical to any other list. The elements of that list, however, are identical to the corresponding elements of the left operand (see 21.6).

It is possible to nest such *sublist extractions*, as can be seen in the example below.

Example

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;  m{[1,2,3]}{[3,2]};
[ [ 3, 2 ], [ 6, 5 ], [ 9, 8 ] ]
```

```
gap> l := m{[1,2,3]};; l{[3,2]};
[ [ 7, 8, 9 ], [ 4, 5, 6 ] ]
```

Note the difference between the two examples. The latter extracts elements 1, 2, and 3 from *m* and then extracts the elements 3 and 2 from *this list*. The former extracts elements 1, 2, and 3 from *m* and then extracts the elements 3 and 2 from *each of those element lists*.

To be precise: With each selector [*pos*] or {*poss*} we associate a *level* that is defined as the number of selectors of the form {*poss*} to its left in the same expression. For example

```
     l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1     2      2     3
```

Then a selector *list*[*pos*] of level *level* is computed as ListElement(*list*,*pos*,*level*), where ListElement is defined as follows. (Note that ListElement is *not* a GAP function.)

Example

```
ListElement := function ( list, pos, level )
 if level = 0 then
  return list[pos];
 else
  return List( list, elm -> ListElement(elm,pos,level-1) );
 fi;
end;
```

and a selector *list*{*poss*} of level *level* is computed as ListElements(*list*,*poss*,*level*), where ListElements is defined as follows. (Note that ListElements is *not* a GAP function.)

Example

```
ListElements := function ( list, poss, level )
 if level = 0 then
  return list{poss};
  else
   return List( list, elm -> ListElements(elm,poss,level-1) );
  fi;
end;
```

21.3.1 \{\}

▷ \{\}(*list, poss*) (operation)

This operation implements *sublist access*. For any list, the default method is to loop over the entries in the list *poss*, and to delegate to the element access operation. (For the somewhat strange variable name, cf. 21.2.)

21.4 List Assignment

list[*pos*] := *object*;

The list element assignment assigns the object *object*, which can be of any type, to the list entry at the position *pos*, which must be a positive integer, in the mutable (see 12.6) list *list*. That means that accessing the *pos*-th element of the list *list* will return *object* after this assignment.

Example

```
gap> l := [ 1, 2, 3 ];;
gap> l[1] := 3;; l;             # assign a new object
[ 3, 2, 3 ]
gap> l[2] := [ 4, 5, 6 ];; l;   # <object> may be of any type
[ 3, [ 4, 5, 6 ], 3 ]
gap> l[ l[1] ] := 10;; l;       # <index> may be an expression
[ 3, [ 4, 5, 6 ], 10 ]
```

If the index *pos* is larger than the length of the list *list* (see Length (21.17.5)), the list is automatically enlarged to make room for the new element. Note that it is possible to generate lists with holes that way.

Example

```
gap> l[4] := "another entry";; l;  # <list> is enlarged
[ 3, [ 4, 5, 6 ], 10, "another entry" ]
gap> l[ 10 ] := 1;; l;             # now <list> has a hole
[ 3, [ 4, 5, 6 ], 10, "another entry",,,,,, 1 ]
```

The function Add (21.4.2) should be used if you want to add an element to the end of the list.

Note that assigning to a list changes the list, thus this list must be mutable (see 12.6). See 21.6 for subtleties of changing lists.

If *list* does not evaluate to a list, *pos* does not evaluate to a positive integer or *object* is a call to a function which does not return a value (for example Print) an error is signalled.

list{ *poss* } := *objects*;

The sublist assignment assigns the object *objects*[1], which can be of any type, to the list *list* at the position *poss*[1], the object *objects*[2] to *list*[*poss*[2]], and so on. *poss* must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. *objects* must be a dense list and must have the same length as *poss*.

Example

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[1..4]} := [10..13];; l;
[ 10, 11, 12, 13, 11, 13, 17, 19 ]
gap> l{[1,7,1,10]} := [ 1, 2, 3, 4 ];; l;
[ 3, 11, 12, 13, 11, 13, 2, 19,, 4 ]
```

The next example shows that it is possible to nest such sublist assignments.

Example

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]} := [ [11,12], [13,14], [15,16] ];; m;
[ [ 1, 12, 11 ], [ 4, 14, 13 ], [ 7, 16, 15 ], [ 10, 11, 12 ] ]
```

The exact behaviour is defined in the same way as for list extractions (see 21.3). Namely, with each selector [*pos*] or {*poss*} we associate a *level* that is defined as the number of selectors of the form {*poss*} to its left in the same expression. For example

Example

```
    l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1     1      1     2
```

Then a list assignment *list*[*pos*] := *vals*; of level *level* is computed as ListAssignment(*list*, *pos*, *vals*, *level*), where ListAssignment is defined as follows. (Note that ListAssignment is *not* a GAP function.)

Example

```
ListAssignment := function ( list, pos, vals, level )
 local i;
 if level = 0 then
  list[pos] := vals;
 else
  for i in [1..Length(list)] do
   ListAssignment( list[i], pos, vals[i], level-1 );
  od;
 fi;
end;
```

and a list assignment *list*{*poss*} := *vals* of level *level* is computed as ListAssignments(*list*, *poss*, *vals*, *level*), where ListAssignments is defined as follows. (Note that ListAssignments is *not* a GAP function.)

Example

```
ListAssignments := function ( list, poss, vals, level )
 local i;
 if level = 0 then
  list{poss} := vals;
 else
  for i in [1..Length(list)] do
   ListAssignments( list[i], poss, vals[i], level-1 );
  od;
 fi;
end;
```

21.4.1 \{\}\:\=

▷ \{\}\:\=(*list, poss, val*) (operation)

This operation implements sublist assignment. For any list, the default method is to loop over the entries in the list *poss*, and to delegate to the element assignment operation. (For the somewhat strange variable name, cf. 21.2.)

21.4.2 Add

▷ Add(*list, obj[, pos]*) (operation)

adds the element *obj* to the mutable list *list*. The two argument version adds *obj* at the end of *list*, i.e., it is equivalent to the assignment *list*[Length(*list*) + 1] := *obj*, see 21.4.

The three argument version adds *obj* in position *pos*, moving all later elements of the list (if any) up by one position. Any holes at or after position *pos* are also moved up by one position, and new holes are created before *pos* if they are needed.

Nothing is returned by Add, the function is only called for its side effect.

21.4.3 Remove

▷ Remove(*list[, pos]*) (operation)

removes an element from *list*. The one argument form removes the last element. The two argument form removes the element in position *pos*, moving all subsequent elements down one position. Any holes after position *pos* are also moved down by one position.

The one argument form always returns the removed element. In this case *list* must be non-empty.

The two argument form returns the old value of *list*[*pos*] if it was bound, and nothing if it was not. Note that accessing or assigning the return value of this form of the Remove operation is only safe when you *know* that there will be a value, otherwise it will cause an error.

Example

```
gap> l := [ 2, 3, 5 ];; Add( l, 7 ); l;
[ 2, 3, 5, 7 ]
gap> Add(l,4,2); l;
[ 2, 4, 3, 5, 7 ]
gap> Remove(l,2); l;
4
[ 2, 3, 5, 7 ]
gap> Remove(l); l;
7
[ 2, 3, 5 ]
gap> Remove(l,5); l;
[ 2, 3, 5 ]
```

21.4.4 CopyListEntries

▷ CopyListEntries(*fromlst, fromind, fromstep, tolst, toind, tostep, n*) (function)

This function copies *n* elements from *fromlst*, starting at position *fromind* and incrementing the position by *fromstep* each time, into *tolst* starting at position *toind* and incrementing the position by *tostep* each time. *fromlst* and *tolst* must be plain lists. *fromstep* and/or *tostep* can be negative. Unbound positions of *fromlst* are simply copied to *tolst*.

CopyListEntries is used in methods for the operations Add (21.4.2) and Remove (21.4.3).

21.4.5 Append

▷ Append(*list1, list2*) (operation)

adds the elements of the list *list2* to the end of the mutable list *list1*, see 21.4. *list2* may contain holes, in which case the corresponding entries in *list1* will be left unbound. Append returns nothing, it is only called for its side effect.

Note that Append changes its first argument, while Concatenation (21.20.1) creates a new list and leaves its arguments unchanged.

Example

```
gap> l := [ 2, 3, 5 ];; Append( l, [ 7, 11, 13 ] ); l;
[ 2, 3, 5, 7, 11, 13 ]
```

```
gap> Append( l, [ 17,, 23 ] ); l;
[ 2, 3, 5, 7, 11, 13, 17,, 23 ]
```

21.5 IsBound and Unbind for Lists

21.5.1 IsBound (for a list position)

▷ IsBound(*list[, n]*) (operation)

IsBound returns true if the list *list* has a element at the position *n*, and false otherwise. *list* must evaluate to a list, otherwise an error is signalled.

Example

```
gap> l := [ , 2, 3, , 5, , 7, , , , 11 ];;
gap> IsBound( l[7] );
true
gap> IsBound( l[4] );
false
gap> IsBound( l[101] );
false
```

21.5.2 Unbind (unbind a list entry)

▷ Unbind(*list[, n]*) (operation)

Unbind deletes the element at the position *n* in the mutable list *list*. That is, after execution of Unbind, *list* no longer has an assigned value at the position *n*. Thus Unbind can be used to produce holes in a list. Note that it is not an error to unbind a nonexisting list element. *list* must evaluate to a list, otherwise an error is signalled.

Example

```
gap> l := [ , 2, 3, 5, , 7, , , , 11 ];;
gap> Unbind( l[3] ); l;
[ , 2,, 5,, 7,,,, 11 ]
gap> Unbind( l[4] ); l;
[ , 2,,,, 7,,,, 11 ]
```

Note that IsBound (21.5.1) and Unbind are special in that they do not evaluate their argument, otherwise IsBound (21.5.1) would always signal an error when it is supposed to return false and there would be no way to tell Unbind which component to remove.

21.6 Identical Lists

With the list assignment (see 21.4) it is possible to change a mutable list. This section describes the semantic consequences of this fact. (See also 12.5.)

First we define what it means when we say that "an object is changed". You may think that in the following example the second assignment changes the integer.

Example

```
i := 3;
i := i + 1;
```

But in this example it is not the *integer* `3` which is changed, by adding one to it. Instead the *variable* `i` is changed by assigning the value of `i+1`, which happens to be `4`, to `i`. The same thing happens in the example below.

Example

```
l := [ 1, 2 ];
l := [ 1, 2, 3 ];
```

The second assignment does not change the first list, instead it assigns a new list to the variable `l`. On the other hand, in the following example the list *is* changed by the second assignment.

Example

```
l := [ 1, 2 ];
l[3] := 3;
```

To understand the difference, think of a variable as a name for an object. The important point is that a list can have several names at the same time. An assignment *`var`*`:= `*`list`*`;` means in this interpretation that *`var`* is a name for the object *`list`*. At the end of the following example `l2` still has the value `[ 1, 2 ]` as this list has not been changed and nothing else has been assigned to it.

Example

```
l1 := [ 1, 2 ];
l2 := l1;
l1 := [ 1, 2, 3 ];
```

But after the following example the list for which `l2` is a name has been changed and thus the value of `l2` is now `[ 1, 2, 3 ]`.

Example

```
l1 := [ 1, 2 ];
l2 := l1;
l1[3] := 3;
```

We say that two lists are *identical* if changing one of them by a list assignment also changes the other one. This is slightly incorrect, because if *two* lists are identical, there are actually only two names for *one* list. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical lists must be equal, because there is only one list with two different names. Thus identity is an equivalence relation that is a refinement of equality. Identity of objects can be detected using `IsIdenticalObj` (12.5.1).

Let us now consider under which circumstances two lists are identical.

If you enter a list literal then the list denoted by this literal is a new list that is not identical to any other list. Thus in the following example `l1` and `l2` are not identical, though they are equal of course.

Example

```
l1 := [ 1, 2 ];
l2 := [ 1, 2 ];
```

Also in the following example, no lists in the list `l` are identical.

Example
```
l := [];
for i in [1..10] do l[i] := [ 1, 2 ]; od;
```

If you assign a list to a variable no new list is created. Thus the list value of the variable on the left hand side and the list on the right hand side of the assignment are identical. So in the following example `l1` and `l2` are identical lists.

Example
```
l1 := [ 1, 2 ];
l2 := l1;
```

If you pass a list as an argument, the old list and the argument of the function are identical. Also if you return a list from a function, the old list and the value of the function call are identical. So in the following example `l1` and `l2` are identical lists:

Example
```
l1 := [ 1, 2 ];
f := function ( l ) return l; end;
l2 := f( l1 );
```

If you change a list it keeps its identity. Thus if two lists are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two lists that are not identical will never become identical if you change one of them. So in the following example both `l1` and `l2` are changed, and are still identical.

Example
```
l1 := [ 1, 2 ];
l2 := l1;
l1[1] := 2;
```

21.7 Duplication of Lists

Here we describe the meaning of `ShallowCopy` (12.7.1) and `StructuralCopy` (12.7.2) for lists. For the general definition of these functions, see 12.7.

The subobjects (see `ShallowCopy` (12.7.1)) of a list are exactly its elements.

This means that for any list *`list`*, `ShallowCopy` (12.7.1) returns a mutable *new* list *`new`* that is *not identical* to any other list (see 21.6), and whose elements are identical to the elements of *`list`*.

Analogously, for a *mutable* list *`list`*, `StructuralCopy` (12.7.2) returns a mutable *new* list *`scp`* that is *not identical* to any other list, and whose elements are structural copies (defined recursively) of the elements of *`list`*; an element of *`scp`* is mutable (and then a *new* list) if and only if the corresponding element of *`list`* is mutable.

In both cases, modifying the copy *`new`* resp. *`scp`* by assignments (see 21.4) does not modify the original object *`list`*.

`ShallowCopy` (12.7.1) basically executes the following code for lists.

Example
```
new := [];
for i in [ 1 .. Length( list ) ] do
  if IsBound( list[i] ) then
    new[i] := list[i];
```

```
  fi;
od;
```

Example

```
gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];;  list2 := ShallowCopy( list1 );;
gap> IsIdenticalObj( list1, list2 );
false
gap> IsIdenticalObj( list1[1], list2[1] );
true
gap> list2[1] := 0;;  list1;  list2;
[ [ 1, 2 ], [ 3, 4 ] ]
[ 0, [ 3, 4 ] ]
```

StructuralCopy (12.7.2) basically executes the following code for lists.

Example

```
new := [];
for i in [ 1 .. Length( list ) ] do
  if IsBound( list[i] ) then
    new[i] := StructuralCopy( list[i] );
  fi;
od;
```

Example

```
gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];;  list2 := StructuralCopy( list1 );;
gap> IsIdenticalObj( list1, list2 );
false
gap> IsIdenticalObj( list1[1], list2[1] );
false
gap> list2[1][1] := 0;;  list1;  list2;
[ [ 1, 2 ], [ 3, 4 ] ]
[ [ 0, 2 ], [ 3, 4 ] ]
```

The above code is not entirely correct. If the object *list* contains a mutable object twice this object is not copied twice, as would happen with the above definition, but only once. This means that the copy *new* and the object *list* have exactly the same structure when viewed as a general graph.

Example

```
gap> sub := [ 1, 2 ];; list1 := [ sub, sub ];;
gap> list2 := StructuralCopy( list1 );
[ [ 1, 2 ], [ 1, 2 ] ]
gap> list2[1][1] := 0;; list2;
[ [ 0, 2 ], [ 0, 2 ] ]
gap> list1;
[ [ 1, 2 ], [ 1, 2 ] ]
```

21.8 Membership Test for Lists

21.8.1 \in (element test for lists)

▷ \in(*obj, list*) (operation)

This function call or the infix variant *obj* `in` *list* tests whether there is a positive integer *i* such that *list*$[i] =$ *obj* holds.

If the list *list* knows that it is strictly sorted (see `IsSSortedList` (21.17.4)), the membership test is much quicker, because a binary search can be used instead of the linear search used for arbitrary lists, see `\in` (21.19.1).

Example

```
gap> 1 in [ 2, 2, 1, 3 ];  1 in [ 4, -1, 0, 3 ];
true
false
gap> s := SSortedList( [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32] );;
gap> 17 in s;  # uses binary search and only 4 comparisons
false
```

For finding the position of an element in a list, see 21.16.

21.9 Enlarging Internally Represented Lists

Section 21.4 told you (among other things) that it is possible to assign beyond the logical end of a mutable list, automatically enlarging the list. This section tells you how this is done for internally represented lists.

It would be extremely wasteful to make all lists large enough so that there is room for all assignments, because some lists may have more than 100000 elements, while most lists have less than 10 elements.

On the other hand suppose every assignment beyond the end of a list would be done by allocating new space for the list and copying all entries to the new space. Then creating a list of 1000 elements by assigning them in order, would take half a million copy operations and also create a lot of garbage that the garbage collector would have to reclaim.

So the following strategy is used. If a list is created it is created with exactly the correct size. If a list is enlarged, because of an assignment beyond the end of the list, it is enlarged by at least *length*`/8 + 4` entries. Therefore the next assignments beyond the end of the list do not need to enlarge the list. For example creating a list of 1000 elements by assigning them in order, would now take only 32 enlargements.

The result of this is of course that the *physical length* of a list may be larger than the *logical length*, which is usually called simply the length of the list. Aside from the implications for the performance you need not be aware of the physical length. In fact all you can ever observe, for example by calling `Length` (21.17.5), is the logical length.

Suppose that `Length` (21.17.5) would have to take the physical length and then test how many entries at the end of a list are unassigned, to compute the logical length of the list. That would take too much time. In order to make `Length` (21.17.5), and other functions that need to know the logical length, more efficient, the length of a list is stored along with the list.

For fine tuning code dealing with plain lists we provide the following two functions.

21.9.1 EmptyPlist

▷ `EmptyPlist(`*len*`)` (function)

Returns: a plain list

▷ ShrinkAllocationPlist(*l*) (function)

Returns: nothing

The function EmptyPlist returns an empty plain list which has enough memory allocated for *len* entries. This can be useful for creating and filling a plain list with a known number of entries.

The function ShrinkAllocationPlist gives back to GAP's memory manager the physical memory which is allocated for the plain list *l* but not needed by the current number of entries.

Note that there are similar functions EmptyString (27.3.5) and ShrinkAllocationString (27.3.5) for strings instead of plain lists.

Example

```
gap> l:=[]; for i in [1..160] do Add(l, i^2); od;
[ ]
gap> m:=EmptyPlist(160); for i in [1..160] do Add(m, i^2); od;
[ ]
gap> # now l uses about 25% more memory than the equal list m
gap> ShrinkAllocationPlist(l);
gap> # now l and m use the same amount of memory
```

21.10 Comparisons of Lists

list1 = *list2*

list1 <> *list2*

Two lists *list1* and *list2* are equal if and only if for every index *i*, either both entries *list1*[*i*] and *list2*[*i*] are unbound, or both are bound and are equal, i.e., *list1*[*i*] = *list2*[*i*] is true.

Example

```
gap> [ 1, 2, 3 ] = [ 1, 2, 3 ];
true
gap> [ , 2, 3 ] = [ 1, 2, ];
false
gap> [ 1, 2, 3 ] = [ 3, 2, 1 ];
false
```

This definition will cause problems with lists which are their own entries. Comparing two such lists for equality may lead to an infinite recursion in the kernel if the list comparison has to compare the list entries which are in fact the lists themselves, and then GAP crashes.

list1 < *list2*

list1 <= *list2*

Lists are ordered *lexicographically*. Unbound entries are smaller than any bound entry. That implies the following behaviour. Let *i* be the smallest positive integer *i* such that *list1* and *list2* at position *i* differ, i.e., either exactly one of *list1*[*i*], *list2*[*i*] is bound or both entries are bound and differ. Then *list1* is less than *list2* if either *list1*[*i*] is unbound (and *list2*[*i*] is not) or both are bound and *list1*[*i*] < *list2*[*i*] is true.

Example

```
gap> [ 1, 2, 3, 4 ] < [ 1, 2, 4, 8 ]; # <list1>[3] < <list2>[3]
true
gap> [ 1, 2, 3 ] < [ 1, 2, 3, 5 ];  # <list1>[4] is unbound and thus < 5
true
gap> [ 1, , 3, 4 ] < [ 1, -1, 3 ];  # <list1>[2] is unbound and thus < -1
true
```

Note that for comparing two lists with < or <=, the (relevant) list elements must be comparable with <, which is usually *not* the case for objects in different families, see 13.1. Also for the possibility to compare lists with other objects, see 13.1.

21.11 Arithmetic for Lists

It is convenient to have arithmetic operations for lists, in particular because in GAP row vectors and matrices are special kinds of lists. However, it is the wide variety of list objects because of which we prescribe arithmetic operations *not for all* of them. (Keep in mind that "list" means just an object in the category `IsList` (21.1.1).)

(Due to the intended generality and flexibility, the definitions given in the following sections are quite technical. But for not too complicated cases such as matrices (see 24.3) and row vectors (see 23.2) whose entries aren't lists, the resulting behaviour should be intuitive.)

For example, we want to deal with matrices which can be added and multiplied in the usual way, via the infix operators + and *; and we want also Lie matrices, with the same additive behaviour but with the multiplication defined by the Lie bracket. Both kinds of matrices shall be lists, with the usual access to their rows, with `Length` (21.17.5) returning the number of rows etc.

For the categories and attributes that control the arithmetic behaviour of lists, see 21.12.

For the definition of return values of additive and multiplicative operations whose arguments are lists in these filters, see 21.13 and 21.14, respectively. It should be emphasized that these sections describe only what the return values are, and not how they are computed.

For the mutability status of the return values, see 21.15. (Note that this is not dealt with in the sections about the result values.)

Further details about the special cases of row vectors and matrices can be found in 23.2 and in 24.3, the compression status is dealt with in 23.3 and 24.14.

21.12 Filters Controlling the Arithmetic Behaviour of Lists

The arithmetic behaviour of lists is controlled by their types. The following categories and attributes are used for that.

Note that we distinguish additive and multiplicative behaviour. For example, Lie matrices have the usual additive behaviour but not the usual multiplicative behaviour.

21.12.1 IsGeneralizedRowVector

▷ `IsGeneralizedRowVector(`*list*`)` (Category)

For a list *list*, the value `true` for `IsGeneralizedRowVector` indicates that the additive arithmetic behaviour of *list* is as defined in 21.13, and that the attribute `NestingDepthA` (21.12.4) will return a nonzero value when called with *list*.

Example

```
gap> IsList( "abc" ); IsGeneralizedRowVector( "abc" );
true
false
gap> liemat:= LieObject( [ [ 1, 2 ], [ 3, 4 ] ] );
LieObject( [ [ 1, 2 ], [ 3, 4 ] ] )
```

```
gap> IsGeneralizedRowVector( liemat );
true
```

21.12.2 IsMultiplicativeGeneralizedRowVector

▷ IsMultiplicativeGeneralizedRowVector(*list*) (Category)

For a list *list*, the value true for IsMultiplicativeGeneralizedRowVector indicates that the multiplicative arithmetic behaviour of *list* is as defined in 21.14, and that the attribute NestingDepthM (21.12.5) will return a nonzero value when called with *list*.

Example

```
gap> IsMultiplicativeGeneralizedRowVector( liemat );
false
gap> bas:= CanonicalBasis( FullRowSpace( Rationals, 3 ) );
CanonicalBasis( ( Rationals^3 ) )
gap> IsMultiplicativeGeneralizedRowVector( bas );
true
```

Note that the filters IsGeneralizedRowVector (21.12.1), IsMultiplicativeGeneralizedRowVector do *not* enable default methods for addition or multiplication (cf. IsListDefault (21.12.3)).

21.12.3 IsListDefault

▷ IsListDefault(*list*) (Category)

For a list *list*, IsListDefault indicates that the default methods for arithmetic operations of lists, such as pointwise addition and multiplication as inner product or matrix product, shall be applicable to *list*.

IsListDefault implies IsGeneralizedRowVector (21.12.1) and IsMultiplicativeGeneralizedRowVector (21.12.2).

All internally represented lists are in this category, and also all lists in the representations IsGF2VectorRep, Is8BitVectorRep, IsGF2MatrixRep, and Is8BitMatrixRep (see 23.3 and 24.14). Note that the result of an arithmetic operation with lists in IsListDefault will in general be an internally represented list, so most "wrapped list objects" will not lie in IsListDefault.

Example

```
gap> v:= [ 1, 2 ];;  m:= [ v, 2*v ];;
gap> IsListDefault( v );  IsListDefault( m );
true
true
gap> IsListDefault( bas );  IsListDefault( liemat );
true
false
```

21.12.4 NestingDepthA

▷ NestingDepthA(*obj*) (attribute)

For a GAP object *obj*, NestingDepthA returns the *additive nesting depth* of *obj*. This is defined recursively as the integer 0 if *obj* is not in IsGeneralizedRowVector (21.12.1), as the integer 1 if *obj* is an empty list in IsGeneralizedRowVector (21.12.1), and as 1 plus the additive nesting depth of the first bound entry in *obj* otherwise.

21.12.5 NestingDepthM

▷ NestingDepthM(*obj*) (attribute)

For a GAP object *obj*, NestingDepthM returns the *multiplicative nesting depth* of *obj*. This is defined recursively as the integer 0 if *obj* is not in IsMultiplicativeGeneralizedRowVector (21.12.2), as the integer 1 if *obj* is an empty list in IsMultiplicativeGeneralizedRowVector (21.12.2), and as 1 plus the multiplicative nesting depth of the first bound entry in *obj* otherwise.

Example

```
gap> NestingDepthA( v );  NestingDepthM( v );
1
1
gap> NestingDepthA( m );  NestingDepthM( m );
2
2
gap> NestingDepthA( liemat );  NestingDepthM( liemat );
2
0
gap> l1:= [ [ 1, 2 ], 3 ];;  l2:= [ 1, [ 2, 3 ] ];;
gap> NestingDepthA( l1 );  NestingDepthM( l1 );
2
2
gap> NestingDepthA( l2 );  NestingDepthM( l2 );
1
1
```

21.13 Additive Arithmetic for Lists

In this general context, we define the results of additive operations only in the following situations. For unary operations (zero and additive inverse), the unique argument must be in IsGeneralizedRowVector (21.12.1); for binary operations (addition and subtraction), at least one argument must be in IsGeneralizedRowVector (21.12.1), and the other either is not a list or also in IsGeneralizedRowVector (21.12.1).

(For non-list GAP objects, defining the results of unary operations is not an issue here, and if at least one argument is a list not in IsGeneralizedRowVector (21.12.1), it shall be left to this argument whether the result in question is defined and what it is.)

21.13.1 Zero for lists

The zero (see Zero (31.10.3)) of a list x in IsGeneralizedRowVector (21.12.1) is defined as the list whose entry at position i is the zero of $x[i]$ if this entry is bound, and is unbound otherwise.

Example

```
gap> Zero( [ 1, 2, 3 ] );  Zero( [ [ 1, 2 ], 3 ] );  Zero( liemat );
[ 0, 0, 0 ]
```

```
[ [ 0, 0 ], 0 ]
LieObject( [ [ 0, 0 ], [ 0, 0 ] ] )
```

21.13.2 AdditiveInverse for lists

The additive inverse (see `AdditiveInverse` (31.10.9)) of a list x in `IsGeneralizedRowVector` (21.12.1) is defined as the list whose entry at position i is the additive inverse of $x[i]$ if this entry is bound, and is unbound otherwise.

Example

```
gap> AdditiveInverse( [ 1, 2, 3 ] );  AdditiveInverse( [ [ 1, 2 ], 3 ] );
[ -1, -2, -3 ]
[ [ -1, -2 ], -3 ]
```

21.13.3 Addition of lists

If x and y are in `IsGeneralizedRowVector` (21.12.1) and have the same additive nesting depth (see `NestingDepthA` (21.12.4)), the sum $x+y$ is defined *pointwise*, in the sense that the result is a list whose entry at position i is $x[i]+y[i]$ if these entries are bound, is a shallow copy (see `ShallowCopy` (12.7.1)) of $x[i]$ or $y[i]$ if the other argument is not bound at position i, and is unbound if both x and y are unbound at position i.

If x is in `IsGeneralizedRowVector` (21.12.1) and y is in `IsGeneralizedRowVector` (21.12.1) and has lower additive nesting depth, or is neither a list nor a domain, the sum $x+y$ is defined as a list whose entry at position i is $x[i]+y$ if x is bound at position i, and is unbound if not. The equivalent holds in the reversed case, where the order of the summands is kept, as addition is not always commutative.

Example

```
gap> 1 + [ 1, 2, 3 ];  [ 1, 2, 3 ] + [ 0, 2, 4 ];  [ 1, 2 ] + [ Z(2) ];
[ 2, 3, 4 ]
[ 1, 4, 7 ]
[ 0*Z(2), 2 ]
gap> l1:= [ 1, , 3, 4 ];;             l2:= [ , 2, 3, 4, 5 ];;
gap> l3:= [ [ 1, 2 ], , [ 5, 6 ] ];;  l4:= [ , [ 3, 4 ], [ 5, 6 ] ];;
gap> NestingDepthA( l1 );  NestingDepthA( l2 );
1
1
gap> NestingDepthA( l3 );  NestingDepthA( l4 );
2
2
gap> l1 + l2;
[ 1, 2, 6, 8, 5 ]
gap> l1 + l3;
[ [ 2, 2, 3, 4 ],, [ 6, 6, 3, 4 ] ]
gap> l2 + l4;
[ , [ 3, 6, 3, 4, 5 ], [ 5, 8, 3, 4, 5 ] ]
gap> l3 + l4;
[ [ 1, 2 ], [ 3, 4 ], [ 10, 12 ] ]
gap> l1 + [];
[ 1,, 3, 4 ]
```

21.13.4 Subtraction of lists

For two GAP objects x and y of which one is in `IsGeneralizedRowVector` (21.12.1) and the other is also in `IsGeneralizedRowVector` (21.12.1) or is neither a list nor a domain, $x-y$ is defined as $x+(-y)$.

Example

```
gap> l1 - l2;
[ 1, -2, 0, 0, -5 ]
gap> l1 - l3;
[ [ 0, -2, 3, 4 ],, [ -4, -6, 3, 4 ] ]
gap> l2 - l4;
[ , [ -3, -2, 3, 4, 5 ], [ -5, -4, 3, 4, 5 ] ]
gap> l3 - l4;
[ [ 1, 2 ], [ -3, -4 ], [ 0, 0 ] ]
gap> l1 - [];
[ 1,, 3, 4 ]
```

21.14 Multiplicative Arithmetic for Lists

In this general context, we define the results of multiplicative operations only in the following situations. For unary operations (one and inverse), the unique argument must be in `IsMultiplicativeGeneralizedRowVector` (21.12.2); for binary operations (multiplication and division), at least one argument must be in `IsMultiplicativeGeneralizedRowVector` (21.12.2), and the other either not a list or also in `IsMultiplicativeGeneralizedRowVector` (21.12.2).

(For non-list GAP objects, defining the results of unary operations is not an issue here, and if at least one argument is a list not in `IsMultiplicativeGeneralizedRowVector` (21.12.2), it shall be left to this argument whether the result in question is defined and what it is.)

21.14.1 One for lists

The one (see `One` (31.10.2)) of a dense list x in `IsMultiplicativeGeneralizedRowVector` (21.12.2) such that x has even multiplicative nesting depth and has the same length as each of its rows is defined as the usual identity matrix on the outer two levels, that is, an identity matrix of the same dimensions, with diagonal entries `One( x[1][1] )` and off-diagonal entries `Zero( x[1][1] )`.

Example

```
gap> One( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ 1, 0 ], [ 0, 1 ] ]
gap> One( [ [ [ [ 1 ] ], [ [ 2 ] ] ], [ [ [ 3 ] ], [ [ 4 ] ] ] ] );
[ [ [ [ 1 ] ], [ [ 0 ] ] ], [ [ [ 0 ] ], [ [ 1 ] ] ] ]
```

21.14.2 Inverse for lists

The inverse (see `Inverse` (31.10.8)) of an invertible square table x in `IsMultiplicativeGeneralizedRowVector` (21.12.2) whose entries lie in a common field is defined as the usual inverse y, i.e., a square matrix over the same field such that xy and yx is equal to `One( x )`.

Example

```
gap> Inverse( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ -2, 1 ], [ 3/2, -1/2 ] ]
```

21.14.3 Multiplication of lists

There are three possible computations that might be triggered by a multiplication involving a list in `IsMultiplicativeGeneralizedRowVector` (21.12.2). Namely, $x * y$ might be

(I) the inner product $x[1] * y[1] + x[2] * y[2] + \cdots + x[n] * y[n]$, where summands are omitted for which the entry in x or y is unbound (if this leaves no summand then the multiplication is an error), or

(L) the left scalar multiple, i.e., a list whose entry at position i is $x * y[i]$ if y is bound at position i, and is unbound if not, or

(R) the right scalar multiple, i.e., a list whose entry at position i is $x[i] * y$ if x is bound at position i, and is unbound if not.

Our aim is to generalize the basic arithmetic of simple row vectors and matrices, so we first summarize the situations that shall be covered.

	scl	vec	mat
scl		(L)	(L)
vec	(R)	(I)	(I)
mat	(R)	(R)	(R)

This means for example that the product of a scalar (scl) with a vector (vec) or a matrix (mat) is computed according to (L). Note that this is asymmetric.

Now we can state the general multiplication rules.

If exactly one argument is in `IsMultiplicativeGeneralizedRowVector` (21.12.2) then we regard the other argument (which is then neither a list nor a domain) as a scalar, and specify result (L) or (R), depending on ordering.

In the remaining cases, both x and y are in `IsMultiplicativeGeneralizedRowVector` (21.12.2), and we distinguish the possibilities by their multiplicative nesting depths. An argument with *odd* multiplicative nesting depth is regarded as a vector, and an argument with *even* multiplicative nesting depth is regarded as a scalar or a matrix.

So if both arguments have odd multiplicative nesting depth, we specify result (I).

If exactly one argument has odd nesting depth, the other is treated as a scalar if it has lower multiplicative nesting depth, and as a matrix otherwise. In the former case, we specify result (L) or (R), depending on ordering; in the latter case, we specify result (L) or (I), depending on ordering.

We are left with the case that each argument has even multiplicative nesting depth. If the two depths are equal, we treat the computation as a matrix product, and specify result (R). Otherwise, we treat the less deeply nested argument as a scalar and the other as a matrix, and specify result (L) or (R), depending on ordering.

Example

```
gap> [ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ] * (1,4);
[ (1,4), (1,4)(2,3), (1,2,4), (1,2,3,4), (1,3,2,4), (1,3,4) ]
gap> [ 1, 2, , 4 ] * 2;
```

```
[ 2, 4,, 8 ]
gap> [ 1, 2, 3 ] * [ 1, 3, 5, 7 ];
22
gap> m:= [ [ 1, 2 ], 3 ];;  m * m;
[ [ 7, 8 ], [ [ 3, 6 ], 9 ] ]
gap> m * m = [ m[1] * m, m[2] * m ];
true
gap> n:= [ 1, [ 2, 3 ] ];;  n * n;
14
gap> n * n = n[1] * n[1] + n[2] * n[2];
true
```

21.14.4 Division of lists

For two GAP objects x and y of which one is in `IsMultiplicativeGeneralizedRowVector` (21.12.2) and the other is also in `IsMultiplicativeGeneralizedRowVector` (21.12.2) or is neither a list nor a domain, x/y is defined as $x * y^{-1}$.

Example

```
gap> [ 1, 2, 3 ] / 2;  [ 1, 2 ] / [ [ 1, 2 ], [ 3, 4 ] ];
[ 1/2, 1, 3/2 ]
[ 1, 0 ]
```

21.14.5 mod for lists

If x and y are in `IsMultiplicativeGeneralizedRowVector` (21.12.2) and have the same multiplicative nesting depth (see `NestingDepthM` (21.12.5)), `x mod y` is defined *pointwise*, in the sense that the result is a list whose entry at position i is `x[i] mod y[i]` if these entries are bound, is a shallow copy (see `ShallowCopy` (12.7.1)) of $x[i]$ or $y[i]$ if the other argument is not bound at position i, and is unbound if both x and y are unbound at position i.

If x is in `IsMultiplicativeGeneralizedRowVector` (21.12.2) and y is in `IsMultiplicativeGeneralizedRowVector` (21.12.2) and has lower multiplicative nesting depth or is neither a list nor a domain, `x mod y` is defined as a list whose entry at position i is `x[i] mod y` if x is bound at position i, and is unbound if not. The equivalent holds in the reversed case, where the order of the arguments is kept.

Example

```
gap> 4711 mod [ 2, 3,, 5, 7 ];
[ 1, 1,, 1, 0 ]
gap> [ 2, 3, 4, 5, 6 ] mod 3;
[ 2, 0, 1, 2, 0 ]
gap> [ 10, 12, 14, 16 ] mod [ 3, 5, 7 ];
[ 1, 2, 0, 16 ]
```

21.14.6 Left quotients of lists

For two GAP objects x and y of which one is in `IsMultiplicativeGeneralizedRowVector` (21.12.2) and the other is also in `IsMultiplicativeGeneralizedRowVector` (21.12.2) or is neither a list nor a domain, `LeftQuotient( x, y )` is defined as $x^{-1} * y$.

Example

```
gap> LeftQuotient( [ [ 1, 2 ], [ 3, 4 ] ], [ 1, 2 ] );
[ 0, 1/2 ]
```

21.15 Mutability Status and List Arithmetic

Many results of arithmetic operations, when applied to lists, are again lists, and it is of interest whether their entries are mutable or not (if applicable). Note that the mutability status of the result itself is already defined by the general rule for any result of an arithmetic operation, not only for lists (see 12.6).

However, we do *not* define exactly the mutability status for each element on each level of a nested list returned by an arithmetic operation. (Of course it would be possible to define this recursively, but since the methods used are in general not recursive, in particular for efficient multiplication of compressed matrices, such a general definition would be a burden in these cases.) Instead we consider, for a list x in `IsGeneralizedRowVector` (21.12.1), the sequence $x = x_1, x_2, \ldots x_n$ where x_{i+1} is the first bound entry in x_i if exists (that is, if x_i is a nonempty list), and n is the largest i such that x_i lies in `IsGeneralizedRowVector` (21.12.1). The *immutability level* of x is defined as infinity if x is immutable, and otherwise the number of x_i which are immutable. (So the immutability level of a mutable empty list is 0.)

Thus a fully mutable matrix has immutability level 0, and a mutable matrix with immutable first row has immutability level 1 (independent of the mutability of other rows).

The immutability level of the result of any of the binary operations discussed here is the minimum of the immutability levels of the arguments, provided that objects of the required mutability status exist in GAP.

Moreover, the results have a "homogeneous" mutability status, that is, if the first bound entry at nesting depth i is immutable (mutable) then all entries at nesting depth i are immutable (mutable, provided that a mutable version of this entry exists in GAP).

Thus the sum of two mutable matrices whose first rows are mutable is a matrix all of whose rows are mutable, and the product of two matrices whose first rows are immutable is a matrix all of whose rows are immutable, independent of the mutability status of the other rows of the arguments.

For example, the sum of a matrix (mutable or immutable, i.e., of immutability level one of 0, 1, or 2) and a mutable row vector (i.e., immutability level 0) is a fully mutable matrix. The product of two mutable row vectors of integers is an integer, and since GAP does not support mutable integers, the result is immutable.

For unary arithmetic operations, there are three operations available, an attribute that returns an immutable result (`Zero` (31.10.3), `AdditiveInverse` (31.10.9), `One` (31.10.2), `Inverse` (31.10.8)), an operation that returns a result that is mutable (`ZeroOp` (31.10.3), `AdditiveInverseOp` (31.10.9), `OneOp` (31.10.2), `InverseOp` (31.10.8)), and an operation whose result has the same immutability level as the argument (`ZeroSM` (31.10.3), `AdditiveInverseSM` (31.10.9), `OneSM` (31.10.2), `InverseSM` (31.10.8)). The last kind of operations is equivalent to the corresponding infix operations `0 * `*`list`*, `- `*`list`*, *`list`*`^0`, and *`list`*`^-1`. (This holds not only for lists, see 12.6.)

Example

```
gap> IsMutable( l1 );  IsMutable( 2 * Immutable( [ 1, 2, 3 ] ) );
true
false
gap> IsMutable( l2 );  IsMutable( l3 );
```

```
true
true
```

An example motivating the mutability rule is the use of syntactic constructs such as *obj* `*` *list* and `-` *list* as an elegant and efficient way to create mutable lists needed for further manipulations from mutable lists. In particular one can construct a mutable zero vector of length *n* by `0 * [ 1 .. n ]`. The latter can be done also using `ListWithIdenticalEntries` (21.15.1).

21.15.1 ListWithIdenticalEntries

▷ `ListWithIdenticalEntries(`*n*`,` *obj*`)` (function)

is a list *list* of length *n* that has the object *obj* stored at each of the positions from `1` to *n*. Note that all elements of *lists* are identical, see 21.6.

Example

```
gap> ListWithIdenticalEntries( 10, 0 );
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

21.16 Finding Positions in Lists

21.16.1 Position

▷ `Position(`*list*`,` *obj*`[,` *from*`])` (operation)

returns the position of the first occurrence *obj* in *list*, or `fail` if *obj* is not contained in *list*. If a starting index *from* is given, it returns the position of the first occurrence starting the search *after* position *from*.

Each call to the two argument version is translated into a call of the three argument version, with third argument the integer zero `0`. (Methods for the two argument version must be installed as methods for the version with three arguments, the third being described by `IsZeroCyc`.)

Example

```
gap> Position( [ 2, 2, 1, 3 ], 1 );
3
gap> Position( [ 2, 1, 1, 3 ], 1 );
2
gap> Position( [ 2, 1, 1, 3 ], 1, 2 );
3
gap> Position( [ 2, 1, 1, 3 ], 1, 3 );
fail
```

21.16.2 Positions

▷ `Positions(`*list*`,` *obj*`)` (function)
▷ `PositionsOp(`*list*`,` *obj*`)` (operation)

returns the positions of *all* occurrences of *obj* in *list*.

Example

```
gap> Positions([1,2,1,2,3,2,2],2);
[ 2, 4, 6, 7 ]
gap> Positions([1,2,1,2,3,2,2],4);
[  ]
```

21.16.3 PositionCanonical

▷ PositionCanonical(*list, obj*) (operation)

returns the position of the canonical associate of *obj* in *list*. The definition of this associate depends on *list*. For internally represented lists it is defined as the element itself (and PositionCanonical thus defaults to Position (21.16.1), but for example for certain enumerators (see 21.23) other canonical associates can be defined.

For example RightTransversal (39.8.1) defines the canonical associate to be the element in the transversal defining the same coset of a subgroup in a group.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;u:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);;
gap> rt:=RightTransversal(g,u);;AsList(rt);
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4) ]
gap> Position(rt,(1,2));
fail
gap> PositionCanonical(rt,(1,2));
2
```

21.16.4 PositionNthOccurrence

▷ PositionNthOccurrence(*list, obj, n*) (operation)

returns the position of the *n*-th occurrence of *obj* in *list* and returns fail if *obj* does not occur *n* times.

Example

```
gap> PositionNthOccurrence([1,2,3,2,4,2,1],1,1);
1
gap> PositionNthOccurrence([1,2,3,2,4,2,1],1,2);
7
gap> PositionNthOccurrence([1,2,3,2,4,2,1],2,3);
6
gap> PositionNthOccurrence([1,2,3,2,4,2,1],2,4);
fail
```

21.16.5 PositionSorted

▷ PositionSorted(*list, elm[, func]*) (function)

Called with two arguments, PositionSorted returns the position of the element *elm* in the sorted list *list*.

Called with three arguments, PositionSorted returns the position of the element *elm* in the list *list*, which must be sorted with respect to *func*. *func* must be a function of two arguments that returns true if the first argument is less than the second argument, and false otherwise.

PositionSorted returns *pos* such that $list[pos-1] < elm$ and $elm \leq list[pos]$. That means, if *elm* appears once in *list*, its position is returned. If *elm* appears several times in *list*, the position of the first occurrence is returned. If *elm* is not an element of *list*, the index where *elm* must be inserted to keep the list sorted is returned.

PositionSorted uses binary search, whereas Position (21.16.1) can in general use only linear search, see the remark at the beginning of 21.19. For sorting lists, see 21.18, for testing whether a list is sorted, see IsSortedList (21.17.3) and IsSSortedList (21.17.4).

Specialized functions for certain kinds of lists must be installed as methods for the operation PositionSortedOp.

Example

```
gap> PositionSorted( [1,4,5,5,6,7], 0 );
1
gap> PositionSorted( [1,4,5,5,6,7], 2 );
2
gap> PositionSorted( [1,4,5,5,6,7], 4 );
2
gap> PositionSorted( [1,4,5,5,6,7], 5 );
3
gap> PositionSorted( [1,4,5,5,6,7], 8 );
7
```

21.16.6 PositionSet

▷ PositionSet(*list, obj[, func]*) (function)

PositionSet is a slight variation of PositionSorted (21.16.5). The only difference to PositionSorted (21.16.5) is that PositionSet returns fail if *obj* is not in *list*.

Example

```
gap> PositionSet( [1,4,5,5,6,7], 0 );
fail
gap> PositionSet( [1,4,5,5,6,7], 2 );
fail
gap> PositionSet( [1,4,5,5,6,7], 4 );
2
gap> PositionSet( [1,4,5,5,6,7], 5 );
3
gap> PositionSet( [1,4,5,5,6,7], 8 );
fail
```

21.16.7 PositionProperty

▷ PositionProperty(*list, func[, from]*) (operation)

returns the position of the first entry in the list *list* for which the property tester function *func* returns true, or fail if no such entry exists. If a starting index *from* is given, it returns the position of the first entry satisfying *func*, starting the search *after* position *from*.

Example

```
gap> PositionProperty( [10^7..10^8], IsPrime );
20
gap> PositionProperty( [10^5..10^6],
>        n -> not IsPrime(n) and IsPrimePowerInt(n) );
490
```

First (21.20.22) allows you to extract the first element of a list that satisfies a certain property.

21.16.8 PositionsProperty

▷ PositionsProperty(*list, func*) (operation)

returns the list of all those positions in the dense list *list* for which the property tester function *func* returns true.

Example

```
gap> l:= [ -5 .. 5 ];;
gap> PositionsProperty( l, IsPosInt );
[ 7, 8, 9, 10, 11 ]
gap> PositionsProperty( l, IsPrimeInt );
[ 1, 3, 4, 8, 9, 11 ]
```

PositionProperty (21.16.7) allows you to extract the position of the first element in a list that satisfies a certain property.

21.16.9 PositionBound

▷ PositionBound(*list*) (operation)

returns the first index for which an element is bound in the list *list*. For the empty list it returns fail.

Example

```
gap> PositionBound([1,2,3]);
1
gap> PositionBound([,1,2,3]);
2
```

21.16.10 PositionNot

▷ PositionNot(*list, val[, from]*) (operation)

For a list *list* and an object *val*, PositionNot returns the smallest nonnegative integer n such that *list*$[n]$ is either unbound or not equal to *val*. If a starting index *from* is given, it returns the first position with this property starting the search *after* position *from*.

Example

```
gap> l:= [ 1, 1, 2, 3, 2 ];;  PositionNot( l, 1 );
3
gap> PositionNot( l, 1, 4 );  PositionNot( l, 2, 4 );
5
6
```

21.16.11 PositionNonZero

▷ PositionNonZero(*vec[, from]*) (operation)

For a row vector *vec*, PositionNonZero returns the position of the first non-zero element of *vec*, or Length(*vec*)+1 if all entries of *vec* are zero.

If a starting index *from* is given, it returns the position of the first occurrence starting the search *after* position *from*.

PositionNonZero implements a special case of PositionNot (21.16.10). Namely, the element to be avoided is the zero element, and the list must be (at least) homogeneous because otherwise the zero element cannot be specified implicitly.

Example

```
gap> PositionNonZero( [ 1, 1, 2, 3, 2 ] );
1
gap> PositionNonZero( [ 2, 3, 4, 5 ] * Z(2) );
2
```

21.16.12 PositionSublist

▷ PositionSublist(*list, sub[, from]*) (operation)

returns the smallest index in the list *list* at which a sublist equal to *sub* starts. If *sub* does not occur the operation returns fail. The version with given *from* starts searching *after* position *from*.

To determine whether *sub* matches *list* at a particular position, use IsMatchingSublist (21.17.1) instead.

21.16.13 PositionFirstComponent

▷ PositionFirstComponent(*list, obj*) (operation)

returns the index *i* in *list* such that *list*[*i*][1] = *obj* or the place where such an entry should be added (cf. PositionSorted (21.16.5)).

21.17 Properties and Attributes for Lists

A list that contains mutable objects (like lists or records) *cannot* store attribute values that depend on the values of its entries, such as whether it is homogeneous, sorted, or strictly sorted, as changes in any of its entries could change such property values, like the following example shows.

Example

```
gap> l:=[[1],[2]];
[ [ 1 ], [ 2 ] ]
gap> IsSSortedList(l);
true
gap> l[1][1]:=3;
3
gap> IsSSortedList(l);
false
```

For such lists these property values must be computed anew each time the property is asked for. For example, if *list* is a list of mutable row vectors then the call of Position (21.16.1) with *list* as first argument cannot take advantage of the fact that *list* is in fact sorted. One solution is to call explicitly PositionSorted (21.16.5) in such a situation, another solution is to replace *list* by an immutable copy using Immutable (12.6.3).

21.17.1 IsMatchingSublist

▷ IsMatchingSublist(*list, sub[, at]*) (operation)

returns true if *sub* matches a sublist of *list* from position 1 (or position *at*, in the case of three arguments), or false, otherwise. If *sub* is empty true is returned. If *list* is empty but *sub* is non-empty false is returned.

If you actually want to know whether there is an *at* for which IsMatchingSublist(*list*, *sub*, *at*) is true, use a construction like PositionSublist(*list*, *sub*) <> fail instead (see PositionSublist (21.16.12)); it's more efficient.

21.17.2 IsDuplicateFree

▷ IsDuplicateFree(*obj*) (property)
▷ IsDuplicateFreeList(*obj*) (property)

IsDuplicateFree returns true if *obj* is both a list or collection, and it is duplicate free; otherwise it returns false. IsDuplicateFreeList is a synonym for IsDuplicateFree and IsList.

A list is *duplicate free* if it is dense and does not contain equal entries in different positions. Every domain (see 12.4) is duplicate free.

Note that GAP cannot compare arbitrary objects (by equality). This can cause that IsDuplicateFree runs into an error, if *obj* is a list with some non-comparable entries.

21.17.3 IsSortedList

▷ IsSortedList(*obj*) (property)

returns true if *obj* is a list and it is sorted, and false otherwise.

A list *list* is *sorted* if it is dense (see IsDenseList (21.1.2)) and satisfies the relation $list[i] \leq list[j]$ whenever $i < j$. Note that a sorted list is not necessarily duplicate free (see IsDuplicateFree (21.17.2) and IsSSortedList (21.17.4)).

Many sorted lists are in fact homogeneous (see IsHomogeneousList (21.1.3)), but also non-homogeneous lists may be sorted (see 31.11).

In sorted lists, membership test and computing of positions can be done by binary search, see 21.19.

Note that GAP cannot compare (by less than) arbitrary objects. This can cause that IsSortedList runs into an error, if *obj* is a list with some non-comparable entries.

21.17.4 IsSSortedList

▷ IsSSortedList(*obj*) (property)
▷ IsSet(*obj*) (property)

returns `true` if *obj* is a list and it is strictly sorted, and `false` otherwise. `IsSSortedList` is short for "is strictly sorted list"; `IsSet` is just a synonym for `IsSSortedList`.

A list *list* is *strictly sorted* if it is sorted (see `IsSortedList` (21.17.3)) and satisfies the relation $list[i] < list[j]$ whenever $i < j$. In particular, such lists are duplicate free (see `IsDuplicateFree` (21.17.2)).

(Currently there is little special treatment of lists that are sorted but not strictly sorted. In particular, internally represented lists will *not* store that they are sorted but not strictly sorted.)

Note that GAP cannot compare (by less than) arbitrary objects. This can cause that `IsSSortedList` runs into an error, if *obj* is a list with some non-comparable entries.

21.17.5 Length

▷ `Length(`*list*`)` (attribute)

returns the *length* of the list *list*, which is defined to be the index of the last bound entry in *list*.

21.17.6 ConstantTimeAccessList

▷ `ConstantTimeAccessList(`*list*`)` (attribute)

`ConstantTimeAccessList` returns an immutable list containing the same elements as the list *list* (which may have holes) in the same order. If *list* is already a constant time access list, `ConstantTimeAccessList` returns an immutable copy of *list* directly. Otherwise it puts all elements and holes of *list* into a new list and makes that list immutable.

21.18 Sorting Lists

21.18.1 Sort

▷ `Sort(`*list*`[, `*func*`])` (operation)
▷ `SortBy(`*list*`, `*func*`)` (operation)

`Sort` sorts the list *list* in increasing order. In the one argument form `Sort` uses the operator `<` to compare the elements. (If the list is not homogeneous it is the users responsibility to ensure that `<` is defined for all element pairs, see 31.11) In the two argument form `Sort` uses the function *func* to compare elements. *func* must be a function taking two arguments that returns `true` if the first is regarded as strictly smaller than the second, and `false` otherwise.

Note that, in cases where it is applicable, `SortBy` is likely to be more efficient.

`Sort` does not return anything, it just changes the argument *list*. Use `ShallowCopy` (12.7.1) if you want to keep *list*. Use `Reversed` (21.20.7) if you want to get a new list that is sorted in decreasing order.

It is possible to sort lists that contain multiple elements which compare equal. It is not guaranteed that those elements keep their relative order, i.e., `Sort` is not stable.

Example

```
gap> list := [ 5, 4, 6, 1, 7, 5 ];; Sort( list ); list;
[ 1, 4, 5, 5, 6, 7 ]
```

```
gap> list := [ [0,6], [1,2], [1,3], [1,5], [0,4], [3,4] ];;
gap> Sort( list, function(v,w) return v*v < w*w; end );
gap> list;  # sorted according to the Euclidean distance from [0,0]
[ [ 1, 2 ], [ 1, 3 ], [ 0, 4 ], [ 3, 4 ], [ 1, 5 ], [ 0, 6 ] ]
gap> list := [ [0,6], [1,3], [3,4], [1,5], [1,2], [0,4], ];;
gap> Sort( list, function(v,w) return v[1] < w[1]; end );
gap> # note the random order of the elements with equal first component:
gap> list;
[ [ 0, 6 ], [ 0, 4 ], [ 1, 3 ], [ 1, 5 ], [ 1, 2 ], [ 3, 4 ] ]
```

SortBy sorts the list *list* into an order such that func(list[i]) <= func(list[i+1]) for all relevant *i*. *func* must thus be a function on one argument which returns values that can be compared. Each func(list[i]) is computed just once and stored, making this more efficient than using the two-argument version of Sort in many cases.

21.18.2 SortParallel

▷ SortParallel(*list1, list2[, func]*) (operation)

sorts the list *list1* in increasing order just as Sort (21.18.1) does. In parallel it applies the same exchanges that are necessary to sort *list1* to the list *list2*, which must of course have at least as many elements as *list1* does.

Example

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 2, 3, 5, 7, 8, 9 ];;
gap> SortParallel( list1, list2 );
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> list2;
[ 7, 3, 2, 9, 5, 8 ]
```

Note that [7, 3, 2, 9, 5, 8] or [7, 3, 9, 2, 5, 8] are possible results.

21.18.3 Sortex

▷ Sortex(*list[, func]*) (operation)

sorts the list *list* and returns a permutation that can be applied to *list* to obtain the sorted list. The one argument form sorts via the operator <, the two argument form sorts w.r.t. the function *func*. (If the list is not homogeneous it is the user's responsibility to ensure that < is defined for all element pairs, see 31.11)

Permuted (21.20.18) allows you to rearrange a list according to a given permutation.

Example

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := ShallowCopy( list1 );;
gap> perm := Sortex( list1 );
(1,3,5,6,4)
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
```

```
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

21.18.4 SortingPerm

▷ SortingPerm(*list*) (attribute)

SortingPerm returns the same as Sortex (21.18.3) but does *not* change the argument.

Example

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := ShallowCopy( list1 );;
gap> perm := SortingPerm( list1 );
(1,3,5,6,4)
gap> list1;
[ 5, 4, 6, 1, 7, 5 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

The default methods for all of these sorting operations currently use Shell sort as it has a comparable performance to Quicksort for lists of length at most a few thousands, and has better worst-case behaviour.

21.19 Sorted Lists and Sets

Searching objects in a list works much quicker if the list is known to be sorted. Currently GAP exploits the sortedness of a list automatically only if the list is *strictly sorted*, which is indicated by the property IsSSortedList (21.17.4).

Remember that a list of *mutable* objects cannot store that it is strictly sorted but has to test it anew whenever it is asked whether it is sorted, see the remark in 21.17. Therefore GAP cannot take advantage of the sortedness of a list if this list has mutable entries. Moreover, if a sorted list *list* with mutable elements is used as an argument of a function that *expects* this argument to be sorted, for example UniteSet (21.19.6) or RemoveSet (21.19.5), then it is checked whether *list* is in fact sorted; this check can have the effect actually to slow down the computations, compared to computations with sorted lists of immutable elements or computations that do not involve functions that do automatically check sortedness.

Strictly sorted lists are used to represent *sets* in GAP. More precisely, a strictly sorted list is called a *proper set* in the following, in order to avoid confusion with domains (see 12.4) which also represent sets.

In short proper sets are represented by sorted lists without holes and duplicates in GAP. Note that we guarantee this representation, so you may make use of the fact that a set is represented by a sorted list in your functions.

In some contexts (for example see 16), we also want to talk about multisets. A *multiset* is like a set, except that an element may appear several times in a multiset. Such multisets are represented by sorted lists without holes that may have duplicates.

This section lists only those functions that are defined exclusively for proper sets. Set theoretic functions for general collections, such as Intersection (30.5.2) and Union (30.5.3), are described

in Chapter 30. In particular, for the construction of proper sets, see SSortedList (30.3.7) and AsSSortedList (30.3.10). For finding positions in sorted lists, see PositionSorted (21.16.5).

There are nondestructive counterparts of the functions UniteSet (21.19.6), IntersectSet (21.19.7), and SubtractSet (21.19.8) available for proper sets. These are UnionSet, IntersectionSet, and Difference (30.5.4). The former two are methods for the more general operations Union (30.5.3) and Intersection (30.5.2), the latter is itself an operation (see Difference (30.5.4)).

The result of IntersectionSet and UnionSet is always a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the first argument *set*. If *set* is not a proper set it is not specified to which of a number of equal elements in *set* the element in the result is identical (see 21.6). The following functions, if not explicitly stated differently, take two arguments, *set* and *obj*, where *set* must be a proper set, otherwise an error is signalled; If the second argument *obj* is a list that is not a proper set then Set (30.3.7) is silently applied to it first.

21.19.1 \in (for strictly sorted lists)

▷ \in(*obj, list*) (method)

For a list *list* that stores that it is strictly sorted, the test with \in (21.19.1) whether the object *obj* is an entry of *list* uses binary search. This test can be entered also with the infix notation *obj* in *list*.

21.19.2 IsEqualSet

▷ IsEqualSet(*list1, list2*) (operation)

tests whether *list1* and *list2* are equal *when viewed as sets*, that is if every element of *list1* is an element of *list2* and vice versa. Either argument of IsEqualSet may also be a list that is not a proper set, in which case Set (30.3.7) is applied to it first.

If both lists are proper sets then they are of course equal if and only if they are also equal as lists. Thus IsEqualSet(*list1*, *list2*) is equivalent to Set(*list1*) = Set(*list2*) (see Set (30.3.7)), but the former is more efficient.

Example

```
gap> IsEqualSet( [2,3,5,7,11], [11,7,5,3,2] );
true
gap> IsEqualSet( [2,3,5,7,11], [2,3,5,7,11,13] );
false
```

21.19.3 IsSubsetSet

▷ IsSubsetSet(*list1, list2*) (operation)

tests whether every element of *list2* is contained in *list1*. Either argument of IsSubsetSet may also be a list that is not a proper set, in which case Set (30.3.7) is applied to it first.

21.19.4 AddSet

▷ AddSet(*set, obj*) (operation)

adds the element *obj* to the proper set *set*. If *obj* is already contained in *set* then *set* is not changed. Otherwise *obj* is inserted at the correct position such that *set* is again a proper set afterwards.

Note that *obj* must be in the same family as each element of *set*.

Example

```
gap> s := [2,3,7,11];;
gap> AddSet( s, 5 );  s;
[ 2, 3, 5, 7, 11 ]
gap> AddSet( s, 13 );  s;
[ 2, 3, 5, 7, 11, 13 ]
gap> AddSet( s, 3 );  s;
[ 2, 3, 5, 7, 11, 13 ]
```

21.19.5 RemoveSet

▷ RemoveSet(*set, obj*) (operation)

removes the element *obj* from the proper set *set*. If *obj* is not contained in *set* then *set* is not changed. If *obj* is an element of *set* it is removed and all the following elements in the list are moved one position forward.

Example

```
gap> s := [ 2, 3, 4, 5, 6, 7 ];;
gap> RemoveSet( s, 6 ); s;
[ 2, 3, 4, 5, 7 ]
gap> RemoveSet( s, 10 ); s;
[ 2, 3, 4, 5, 7 ]
```

21.19.6 UniteSet

▷ UniteSet(*set, list*) (operation)

unites the proper set *set* with *list*. This is equivalent to adding all elements of *list* to *set* (see AddSet (21.19.4)).

Example

```
gap> set := [ 2, 3, 5, 7, 11 ];;
gap> UniteSet( set, [ 4, 8, 9 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> UniteSet( set, [ 16, 9, 25, 13, 16 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 25 ]
```

21.19.7 IntersectSet

▷ IntersectSet(*set, list*) (operation)

intersects the proper set *set* with *list*. This is equivalent to removing from *set* all elements of *set* that are not contained in *list*.

Example

```
gap> set := [ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16 ];;
gap> IntersectSet( set, [ 3, 5, 7, 9, 11, 13, 15, 17 ] );  set;
[ 3, 5, 7, 9, 11, 13 ]
gap> IntersectSet( set, [ 9, 4, 6, 8 ] );  set;
[ 9 ]
```

21.19.8 SubtractSet

▷ SubtractSet(*set, list*) (operation)

subtracts *list* from the proper set *set*. This is equivalent to removing from *set* all elements of *list*.

Example

```
gap> set := [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ];;
gap> SubtractSet( set, [ 6, 10 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> SubtractSet( set, [ 9, 4, 6, 8 ] );  set;
[ 2, 3, 5, 7, 11 ]
```

21.20 Operations for Lists

Several of the following functions expect the first argument to be either a list or a collection (see 30), with possibly slightly different meaning for lists and non-list collections.

21.20.1 Concatenation (for several lists)

▷ Concatenation(*list1, list2, ...*) (function)
▷ Concatenation(*list*) (function)

In the first form Concatenation returns the concatenation of the lists *list1*, *list2*, etc. The *concatenation* is the list that begins with the elements of *list1*, followed by the elements of *list2*, and so on. Each list may also contain holes, in which case the concatenation also contains holes at the corresponding positions.

In the second form *list* must be a dense list of lists *list1*, *list2*, etc., and Concatenation returns the concatenation of those lists.

The result is a new mutable list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of *list1*, *list2*, etc. (see 21.6).

Note that Concatenation creates a new list and leaves its arguments unchanged, while Append (21.4.5) changes its first argument. For computing the union of proper sets, Union (30.5.3) can be used, see also 21.19.

Example

```
gap> Concatenation( [ 1, 2, 3 ], [ 4, 5 ] );
[ 1, 2, 3, 4, 5 ]
gap> Concatenation( [2,3,,5,,7], [11,,13,,,,17,,19] );
[ 2, 3,, 5,, 7, 11,, 13,,,, 17,, 19 ]
```

```
gap> Concatenation( [ [1,2,3], [2,3,4], [3,4,5] ] );
[ 1, 2, 3, 2, 3, 4, 3, 4, 5 ]
```

21.20.2 Compacted

▷ Compacted(*list*) (operation)

returns a new mutable list that contains the elements of *list* in the same order but omitting the holes.

Example

```
gap> l:=[,1,,,3,,,4,[5,,,6],7];;  Compacted( l );
[ 1, 3, 4, [ 5,,, 6 ], 7 ]
```

21.20.3 Collected

▷ Collected(*list*) (operation)

returns a new list *new* that contains for each element *elm* of the list *list* a list of length two, the first element of this is *elm* itself and the second element is the number of times *elm* appears in *list*. The order of those pairs in *new* corresponds to the ordering of the elements elm, so that the result is sorted.

For all pairs of elements in *list* the comparison via < must be defined.

Example

```
gap> Factors( Factorial( 10 ) );
[ 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7 ]
gap> Collected( last );
[ [ 2, 8 ], [ 3, 4 ], [ 5, 2 ], [ 7, 1 ] ]
gap> Collected( last );
[ [ [ 2, 8 ], 1 ], [ [ 3, 4 ], 1 ], [ [ 5, 2 ], 1 ], [ [ 7, 1 ], 1 ] ]
```

21.20.4 DuplicateFreeList

▷ DuplicateFreeList(*list*) (operation)
▷ Unique(*list*) (operation)

returns a new mutable list whose entries are the elements of the list *list* with duplicates removed. DuplicateFreeList only uses the = comparison and will not sort the result. Therefore DuplicateFreeList can be used even if the elements of *list* do not lie in the same family. Otherwise, if *list* contains objects that can be compared with \< (31.11.1) then it is much more efficient to use Set (30.3.7) instead of DuplicateFreeList.

Unique is a synonym for DuplicateFreeList.

Example

```
gap> l:=[1,Z(3),1,"abc",Group((1,2,3),(1,2)),Z(3),Group((1,2),(2,3))];;
gap> DuplicateFreeList( l );
[ 1, Z(3), "abc", Group([ (1,2,3), (1,2) ]) ]
```

21.20.5 AsDuplicateFreeList

▷ `AsDuplicateFreeList(`*list*`)` (attribute)

returns the same result as `DuplicateFreeList` (21.20.4), except that the result is immutable.

21.20.6 Flat

▷ `Flat(`*list*`)` (operation)

returns the list of all elements that are contained in the list *list* or its sublists. That is, `Flat` first makes a new empty list *new*. Then it loops over the elements *elm* of *list*. If *elm* is not a list it is added to *new*, otherwise `Flat` appends `Flat( `*elm*` )` to *new*.

Example

```
gap> Flat( [ 1, [ 2, 3 ], [ [ 1, 2 ], 3 ] ] );
[ 1, 2, 3, 1, 2, 3 ]
gap> Flat( [ ] );
[ ]
```

To reconstruct a matrix from the list obtained by applying `Flat` to the matrix, the sublist operator can be used, as follows.

Example

```
gap> l:=[9..14];;w:=2;; # w is the length of each row
gap> sub:=[1..w];;List([1..Length(l)/w],i->l{(i-1)*w+sub});
[ [ 9, 10 ], [ 11, 12 ], [ 13, 14 ] ]
```

21.20.7 Reversed

▷ `Reversed(`*list*`)` (function)

returns a new mutable list, containing the elements of the dense list *list* in reversed order.

The argument list is unchanged. The result list is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 21.6).

`Reversed` implements a special case of list assignment, which can also be formulated in terms of the `{}` operator (see 21.4).

Example

```
gap> Reversed( [ 1, 4, 9, 5, 6, 7 ] );
[ 7, 6, 5, 9, 4, 1 ]
```

21.20.8 Shuffle

▷ `Shuffle(`*list*`)` (operation)

The argument *list* must be a dense mutable list. This operation permutes the entries of *list* randomly (in place), and returns *list*.

Example

```
gap> Reset(GlobalMersenneTwister, 12345);; # make manual tester happy
gap> l := [1..20];
[ 1 .. 20 ]
gap> m := Shuffle(ShallowCopy(l));
[ 15, 13, 3, 19, 8, 11, 14, 7, 16, 4, 17, 18, 5, 1, 10, 6, 2, 9, 12,
  20 ]
gap> l;
[ 1 .. 20 ]
gap> Shuffle(l);;
gap> l;
[ 3, 4, 18, 13, 10, 7, 9, 8, 14, 17, 16, 6, 19, 12, 1, 11, 20, 2, 15,
  5 ]
```

21.20.9 IsLexicographicallyLess

▷ IsLexicographicallyLess(*list1, list2*) (function)

Let *list1* and *list2* be two dense, but not necessarily homogeneous lists (see IsDenseList (21.1.2), IsHomogeneousList (21.1.3)), such that for each i, the entries in both lists at position i can be compared via <. IsLexicographicallyLess returns true if *list1* is smaller than *list2* w.r.t. lexicographical ordering, and false otherwise.

21.20.10 Apply

▷ Apply(*list, func*) (function)

Apply applies the function *func* to every element of the dense and mutable list *list*, and replaces each element entry by the corresponding return value.

Apply changes its argument. The nondestructive counterpart of Apply is List (30.3.5).

Example

```
gap> l:= [ 1, 2, 3 ];;  Apply( l, i -> i^2 );  l;
[ 1, 4, 9 ]
```

21.20.11 Perform

▷ Perform(*list, func*) (operation)

Perform applies the function *func* to every element of the list *list*, discarding any return values. It does not return a value.

Example

```
gap> l := [1, 2, 3];; Perform(l,
> function(x) if IsPrimeInt(x) then Print(x,"\n"); fi; end);
2
3
```

21.20.12 PermListList

▷ PermListList(*list1, list2*) (function)

returns a permutation p of [1 .. Length(*list1*)] such that $list1[i\hat{}p] = list2[i]$. It returns fail if there is no such permutation.

Example

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 4, 1, 7, 5, 5, 6 ];;
gap> perm := PermListList(list1, list2);
(1,2,4)(3,5,6)
gap> Permuted( list2, perm );
[ 5, 4, 6, 1, 7, 5 ]
```

21.20.13 Maximum

▷ Maximum(*obj1, obj2, ...*) (function)
▷ Maximum(*list*) (function)

In the first form Maximum returns the *maximum* of its arguments, i.e., one argument *obj* for which $obj \geq obj1$, $obj \geq obj2$ etc.

In the second form Maximum takes a homogeneous list *list* and returns the maximum of the elements in this list.

Example

```
gap> Maximum( -123, 700, 123, 0, -1000 );
700
gap> Maximum( [ -123, 700, 123, 0, -1000 ] );
700
gap> # lists are compared elementwise:
gap> Maximum( [1,2], [0,15], [1,5], [2,-11] );
[ 2, -11 ]
```

21.20.14 Minimum

▷ Minimum(*obj1, obj2, ...*) (function)
▷ Minimum(*list*) (function)

In the first form Minimum returns the *minimum* of its arguments, i.e., one argument *obj* for which $obj \leq obj1$, $obj \leq obj2$ etc.

In the second form Minimum takes a homogeneous list *list* and returns the minimum of the elements in this list.

Note that for both Maximum (21.20.13) and Minimum the comparison of the objects *obj1*, *obj2* etc. must be defined; for that, usually they must lie in the same family (see 13.1).

Example

```
gap> Minimum( -123, 700, 123, 0, -1000 );
-1000
gap> Minimum( [ -123, 700, 123, 0, -1000 ] );
-1000
gap> Minimum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
[ 0, 15 ]
```

21.20.15 MaximumList and MinimumList

▷ MaximumList(*list[, seed]*) (operation)
▷ MinimumList(*list[, seed]*) (operation)

return the maximum resp. the minimum of the elements in the list *list*. They are the operations called by Maximum (21.20.13) resp. Minimum (21.20.14). Methods can be installed for special kinds of lists. For example, there are special methods to compute the maximum resp. the minimum of a range (see 21.22).

If a second argument *seed* is supplied, then the result is the maximum resp. minimum of the union of *list* and *seed*. In this manner, the operations may be applied to empty lists.

21.20.16 Cartesian

▷ Cartesian(*list1, list2, ...*) (function)
▷ Cartesian(*list*) (function)

In the first form Cartesian returns the cartesian product of the lists *list1*, *list2*, etc.

In the second form *list* must be a list of lists *list1*, *list2*, etc., and Cartesian returns the cartesian product of those lists.

The *cartesian product* is a list *cart* of lists *tup*, such that the first element of *tup* is an element of *list1*, the second element of *tup* is an element of *list2*, and so on. The total number of elements in *cart* is the product of the lengths of the argument lists. In particular *cart* is empty if and only if at least one of the argument lists is empty. Also *cart* contains duplicates if and only if no argument list is empty and at least one contains duplicates.

The last index runs fastest. That means that the first element *tup1* of *cart* contains the first element from *list1*, from *list2* and so on. The second element *tup2* of *cart* contains the first element from *list1*, the first from *list2*, an so on, but the last element of *tup2* is the second element of the last argument list. This implies that *cart* is a proper set if and only if all argument lists are proper sets (see 21.19).

The function Tuples (16.2.8) computes the *k*-fold cartesian product of a list.

Example

```
gap> Cartesian( [1,2], [3,4], [5,6] );
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ] ]
gap> Cartesian( [1,2,2], [1,1,2] );
[ [ 1, 1 ], [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 1 ], [ 2, 2 ],
  [ 2, 1 ], [ 2, 1 ], [ 2, 2 ] ]
```

21.20.17 IteratorOfCartesianProduct

▷ IteratorOfCartesianProduct(*list1, list2, ...*) (function)
▷ IteratorOfCartesianProduct(*list*) (function)

In the first form IteratorOfCartesianProduct returns an iterator (see 30.8) of all elements of the cartesian product (see Cartesian (21.20.16)) of the lists *list1*, *list2*, etc.

In the second form *list* must be a list of lists *list1*, *list2*, etc., and IteratorOfCartesianProduct returns an iterator of the cartesian product of those lists.

Resulting tuples will be returned in the lexicographic order. Usage of iterators of cartesian products is recommended in the case when the resulting cartesian product is big enough, so its generating and storage will require essential amount of runtime and memory. For smaller cartesian products it is faster to generate the full set of tuples using Cartesian (21.20.16) and then loop over its elements (with some minor overhead of needing more memory).

21.20.18 Permuted

▷ Permuted(*list, perm*) (operation)

returns a new list *new* that contains the elements of the list *list* permuted according to the permutation *perm*. That is *new*[*i*^*perm*] = *list*[*i*].

Sortex (21.18.3) allows you to compute a permutation that must be applied to a list in order to get the sorted list.

Example

```
gap> Permuted( [ 5, 4, 6, 1, 7, 5 ], (1,3,5,6,4) );
[ 1, 4, 5, 5, 6, 7 ]
```

21.20.19 List (for a list (and a function))

▷ List(*list[, func]*) (function)

This function returns a new mutable list new of the same length as the list *list* (which may have holes). The entry new[i] is unbound if *list*[i] is unbound. Otherwise new[i] = *func*(*list*[i]). If the argument *func* is omitted, its default is IdFunc (5.3.4), so this function does the same as ShallowCopy (12.7.1) (see also 21.7).

Example

```
gap> List( [1,2,3], i -> i^2 );
[ 1, 4, 9 ]
gap> List( [1..10], IsPrime );
[ false, true, true, false, true, false, true, false, false, false ]
gap> List([,1,,3,4], x-> x > 2);
[ , false,, true, true ]
```

(See also List (30.3.5).)

21.20.20 Filtered

▷ Filtered(*listorcoll, func*) (function)

returns a new list that contains those elements of the list or collection *listorcoll* (see 30), respectively, for which the unary function *func* returns true.

If the first argument is a list, the order of the elements in the result is the same as the order of the corresponding elements of this list. If an element for which *func* returns true appears several times in the list it will also appear the same number of times in the result. The argument list may contain holes, they are ignored by Filtered.

For each element of *listorcoll*, *func* must return either true or false, otherwise an error is signalled.

The result is a new list that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 21.6).

List assignment using the operator \{\} (21.3.1) (see 21.4) can be used to extract elements of a list according to indices given in another list.

Example

```
gap> Filtered( [1..20], IsPrime );
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
[ 3, 4, 4, 7 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>              n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
[ 3, 7 ]
gap> Filtered( Group( (1,2), (1,2,3) ), x -> Order( x ) = 2 );
[ (2,3), (1,2), (1,3) ]
```

21.20.21 Number

▷ Number(*listorcoll[, func]*) (function)

Called with a list *listorcoll*, Number returns the number of bound entries in this list. For dense lists Number, Length (21.17.5), and Size (30.4.6) return the same value; for lists with holes Number returns the number of bound entries, Length (21.17.5) returns the largest index of a bound entry, and Size (30.4.6) signals an error.

Called with two arguments, a list or collection *listorcoll* and a unary function *func*, Number returns the number of elements of *listorcoll* for which *func* returns true. If an element for which *func* returns true appears several times in *listorcoll* it will also be counted the same number of times.

For each element of *listorcoll*, *func* must return either true or false, otherwise an error is signalled.

Filtered (21.20.20) allows you to extract the elements of a list that have a certain property.

Example

```
gap> Number( [ 2, 3, 5, 7 ] );
4
gap> Number( [, 2, 3,, 5,, 7,,,, 11 ] );
5
gap> Number( [1..20], IsPrime );
8
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
4
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>            n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
2
gap> Number( Group( (1,2), (1,2,3) ), x -> Order( x ) = 2 );
3
```

21.20.22 First

▷ First(*list, func*) (function)

`First` returns the first element of the list *`list`* for which the unary function *`func`* returns `true`. *`list`* may contain holes. *`func`* must return either `true` or `false` for each element of *`list`*, otherwise an error is signalled. If *`func`* returns `false` for all elements of *`list`* then `First` returns `fail`.

`PositionProperty` (21.16.7) allows you to find the position of the first element in a list that satisfies a certain property.

Example

```
gap> First( [10^7..10^8], IsPrime );
10000019
gap> First( [10^5..10^6],
>      n -> not IsPrime(n) and IsPrimePowerInt(n) );
100489
gap> First( [ 1 .. 20 ], x -> x < 0 );
fail
gap> First( [ fail ], x -> x = fail );
fail
```

21.20.23 ForAll

▷ `ForAll(`*`listorcoll, func`*`)` (function)

tests whether the unary function *`func`* returns `true` for all elements in the list or collection *`listorcoll`*.

Example

```
gap> ForAll( [1..20], IsPrime );
false
gap> ForAll( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAll( [2..14], n -> IsPrimePowerInt(n) or n mod 2 = 0 );
true
gap> ForAll( Group( (1,2), (1,2,3) ), i -> SignPerm(i) = 1 );
false
```

21.20.24 ForAny

▷ `ForAny(`*`listorcoll, func`*`)` (function)

tests whether the unary function *`func`* returns `true` for at least one element in the list or collection *`listorcoll`*.

Example

```
gap> ForAny( [1..20], IsPrime );
true
gap> ForAny( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAny( [2..14],
>    n -> IsPrimePowerInt(n) and n mod 5 = 0 and not IsPrime(n) );
false
gap> ForAny( Integers, i ->     i > 0
>                           and ForAll( [0,2..4], j -> IsPrime(i+j) ) );
true
```

21.20.25 Product

▷ Product(*listorcoll[, func][, init]*) (function)

Called with one argument, a dense list or collection *listorcoll*, Product returns the product of the elements of *listorcoll* (see 30).

Called with a dense list or collection *listorcoll* and a function *func*, which must be a function taking one argument, Product applies the function *func* to the elements of *listorcoll*, and returns the product of the results. In either case Product returns 1 if the first argument is empty.

The general rules for arithmetic operations apply (see 21.15), so the result is immutable if and only if all summands are immutable.

If *listorcoll* contains exactly one element then this element (or its image under *func* if applicable) itself is returned, not a shallow copy of this element.

If an additional initial value *init* is given, Product returns the product of *init* and the elements of the first argument resp. of their images under the function *func*. This is useful for example if the first argument is empty and a different identity than 1 is desired, in which case *init* is returned.

Example

```
gap> Product( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
9699690
gap> Product( [1..10], x->x^2 );
13168189440000
gap> Product( [ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) ] );
(1,4)(2,3)
gap> Product( GF(8) );
0*Z(2)
```

21.20.26 Sum

▷ Sum(*listorcoll[, func][, init]*) (function)

Called with one argument, a dense list or collection *listorcoll*, Sum returns the sum of the elements of *listorcoll* (see 30).

Called with a dense list or collection *listorcoll* and a function *func*, which must be a function taking one argument, Sum applies the function *func* to the elements of *listorcoll*, and returns the sum of the results. In either case Sum returns 0 if the first argument is empty.

The general rules for arithmetic operations apply (see 21.15), so the result is immutable if and only if all summands are immutable.

If *listorcoll* contains exactly one element then this element (or its image under *func* if applicable) itself is returned, not a shallow copy of this element.

If an additional initial value *init* is given, Sum returns the sum of *init* and the elements of the first argument resp. of their images under the function *func*. This is useful for example if the first argument is empty and a different zero than 0 is desired, in which case *init* is returned.

Example

```
gap> Sum( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
77
gap> Sum( [1..10], x->x^2 );
385
gap> Sum( [ [1,2], [3,4], [5,6] ] );
[ 9, 12 ]
```

```
gap> Sum( GF(8) );
0*Z(2)
```

21.20.27 Iterated

▷ Iterated(*list, f*) (operation)

returns the result of the iterated application of the function *f*, which must take two arguments, to the elements of the list *list*. More precisely, if *list* has length n then Iterated returns the result of the following application, $f(\ldots f(f(list[1], list[2]), list[3]), \ldots, list[n])$.

Example

```
gap> Iterated( [ 126, 66, 105 ], Gcd );
3
```

21.20.28 ListN

▷ ListN(*list1, list2, ..., listn, f*) (function)

applies the n-argument function *f* to the lists. That is, ListN returns the list whose i-th entry is $f(list1[i], list2[i], \ldots, listn[i])$.

Example

```
gap> ListN( [1,2], [3,4], \+ );
[ 4, 6 ]
```

21.21 Advanced List Manipulations

The following functions are generalizations of List (30.3.5), Set (30.3.7), Sum (21.20.26), and Product (21.20.25).

21.21.1 ListX

▷ ListX(*arg1, arg2, ..., argn, func*) (operation)

ListX returns a new list constructed from the arguments.
Each of the arguments *arg1*, *arg2*, ... *argn* must be one of the following:

a list or collection
this introduces a new for-loop in the sequence of nested for-loops and if-statements;

a function returning a list or collection
this introduces a new for-loop in the sequence of nested for-loops and if-statements, where the loop-range depends on the values of the outer loop-variables; or

a function returning true **or** false
this introduces a new if-statement in the sequence of nested for-loops and if-statements.

The last argument *func* must be a function, it is applied to the values of the loop-variables and the results are collected.

Thus `ListX( list, func )` is the same as `List( list, func )`, and `ListX( list, func, x -> x )` is the same as `Filtered( list, func )`.

As a more elaborate example, assume *arg1* is a list or collection, *arg2* is a function returning `true` or `false`, *arg3* is a function returning a list or collection, and *arg4* is another function returning `true` or `false`, then

result `:= ListX(` *arg1*`,` *arg2*`,` *arg3*`,` *arg4*`,` *func* `);`

is equivalent to

```
result := [];
for v1 in arg1 do
  if arg2( v1 ) then
    for v2 in arg3( v1 ) do
      if arg4( v1, v2 ) then
        Add( result, func( v1, v2 ) );
      fi;
    od;
  fi;
od;
```

The following example shows how `ListX` can be used to compute all pairs and all strictly sorted pairs of elements in a list.

Example

```
gap> l:= [ 1, 2, 3, 4 ];;
gap> pair:= function( x, y ) return [ x, y ]; end;;
gap> ListX( l, l, pair );
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ],
  [ 2, 3 ], [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ],
  [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ] ]
```

In the following example, `\<` (31.11.1) is the comparison operation:

Example

```
gap> ListX( l, l, \<, pair );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

21.21.2 SetX

▷ `SetX(`*arg1, arg2, ..., func*`)` (operation)

The only difference between `SetX` and `ListX` (21.21.1) is that the result list of `SetX` is strictly sorted.

21.21.3 SumX

▷ `SumX(`*arg1, arg2, ..., func*`)` (operation)

`SumX` returns the sum of the elements in the list obtained by `ListX` (21.21.1) when this is called with the same arguments.

21.21.4 ProductX

▷ `ProductX(`*arg1, arg2, ..., func*`)` (operation)

`ProductX` returns the product of the elements in the list obtained by `ListX` (21.21.1) when this is called with the same arguments.

21.22 Ranges

A *range* is a dense list of integers in arithmetic progression (or degression). This is a list of integers such that the difference between consecutive elements is a nonzero constant. Ranges can be abbreviated with the syntactic construct

`[` *first*`,` *second* `..` *last* `]`

or, if the difference between consecutive elements is 1, as

`[` *first* `..` *last* `]`.

If *first* `>` *last*, `[` *first* `..` *last* `]` is the empty list, which by definition is also a range; also, if *second* `>` *first* `>` *last* or *second* `<` *first* `<` *last*, then `[` *first*`,` *second* `..` *last* `]` is the empty list. If *first* `=` *last*, `[` *first*`,` *second* `..` *last* `]` is a singleton list, which is a range, too. Note that *last* `-` *first* must be divisible by the increment *second* `-` *first*, otherwise an error is signalled.

Currently, the integers *first*, *second* and *last* and the length of a range must be small integers, that is at least -2^d and at most $2^d - 1$ with $d = 28$ on 32-bit architectures and $d = 60$ on 64-bit architectures.

Note also that a range is just a special case of a list. Thus you can access elements in a range (see 21.3), test for membership etc. You can even assign to such a range if it is mutable (see 21.4). Of course, unless you assign *last* `+` *second* `-` *first* to the entry *range*`[ Length(` *range* `) + 1 ]`, the resulting list will no longer be a range.

Example

```
gap> r := [10..20];
[ 10 .. 20 ]
gap> Length( r );
11
gap> r[3];
12
gap> 17 in r;
true
gap> r[12] := 25;; r;  # r is no longer a range
[ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25 ]
gap> r := [1,3..17];
[ 1, 3 .. 17 ]
gap> Length( r );
9
gap> r[4];
7
gap> r := [0,-1..-9];
[ 0, -1 .. -9 ]
gap> r[5];
-4
```

```
gap> r := [ 1, 4 .. 32 ];
Error, Range: <last>-<first> (31) must be divisible by <inc> (3)
```

Most often ranges are used in connection with the `for`-loop see 4.20). Here the construct
`for` *var* `in [` *first* `..` *last* `] do` *statements* `od`
replaces the
`for` *var* `from` *first* `to` *last* `do` *statements* `od`
which is more usual in other programming languages.

Example

```
gap> s := [];; for i in [10..20] do Add( s, i^2 ); od; s;
[ 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400 ]
```

Note that a range with *last* `>=` *first* is at the same time also a proper set (see 21.19), because it contains no holes or duplicates and is sorted, and also a row vector (see 23), because it contains no holes and all elements are integers.

21.22.1 IsRange

▷ `IsRange(`*obj*`)` (Category)

tests if the object *obj* is a range, i.e. is a dense list of integers that is also a range (see 21.22 for a definition of "range").

Example

```
gap> IsRange( [1,2,3] );  IsRange( [7,5,3,1] );
true
true
gap> IsRange( [1,2,4,5] );  IsRange( [1,,3,,5,,7] );
false
false
gap> IsRange( [] );  IsRange( [1] );
true
true
```

21.22.2 ConvertToRangeRep

▷ `ConvertToRangeRep(`*list*`)` (function)

For some lists the GAP kernel knows that they are in fact ranges. Those lists are represented internally in a compact way instead of the ordinary way.

If *list* is a range then `ConvertToRangeRep` changes the representation of *list* to this compact representation.

This is important since this representation needs only 12 bytes for the entire range while the ordinary representation needs 4*length* bytes.

Note that a list that is represented in the ordinary way might still be a range. It is just that GAP does not know this. The following rules tell you under which circumstances a range is represented in the compact way, so you can write your program in such a way that you make best use of this compact representation for ranges.

Lists created by the syntactic construct [*first*, *second* .. *last*] are of course known to be ranges and are represented in the compact way.

If you call ConvertToRangeRep for a list represented the ordinary way that is indeed a range, the representation is changed from the ordinary to the compact representation. A call of ConvertToRangeRep for a list that is not a range is ignored.

If you change a mutable range that is represented in the compact way, by assignment, Add (21.4.2) or Append (21.4.5), the range will be converted to the ordinary representation, even if the change is such that the resulting list is still a proper range.

Suppose you have built a proper range in such a way that it is represented in the ordinary way and that you now want to convert it to the compact representation to save space. Then you should call ConvertToRangeRep with that list as an argument. You can think of the call to ConvertToRangeRep as a hint to GAP that this list is a proper range.

Example

```
gap> r:= [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> ConvertToRangeRep( r );  r;
[ 1 .. 10 ]
gap> l:= [ 1, 2, 4, 5 ];;  ConvertToRangeRep( l );  l;
[ 1, 2, 4, 5 ]
```

21.23 Enumerators

An *enumerator* is an immutable list that need not store its elements explicitly but knows, from a set of basic data, how to determine the i-th element and the position of a given object. A typical example of this is a vector space over a finite field with q elements, say, for which it is very easy to enumerate all elements using q-adic expansions of integers.

Using this enumeration can be even quicker than a binary search in a sorted list of vectors, see IsQuickPositionList (21.23.1).

On the one hand, element access to an enumerator may take more time than element access to an internally represented list containing the same elements. On the other hand, an enumerator may save a vast amount of memory. Take for example a permutation group of size a few millions. Even for moderate degree it is unlikely that a list of all its elements will fit into memory whereas it is no problem to construct an enumerator from a stabilizer chain (see 43.6).

There are situations where one only wants to loop over the elements of a domain, without using the special facilities of an enumerator, namely the particular order of elements and the possibility to find the position of elements. For such cases, GAP provides iterators (see 30.8).

The functions Enumerator (30.3.2) and EnumeratorSorted (30.3.3) return enumerators of domains. Most of the special implementations of enumerators in the GAP library are based on the general interface that is provided by EnumeratorByFunctions (30.3.4); one generic example is EnumeratorByBasis (61.6.5), which can be used to get an enumerator of a finite dimensional free module.

Also enumerators for non-domains can be implemented via EnumeratorByFunctions (30.3.4); for a discussion, see 79.13.

21.23.1 IsQuickPositionList

▷ `IsQuickPositionList(`*`list`*`)` (filter)

This filter indicates that a position test in *`list`* is quicker than about 5 or 6 element comparisons for "smaller". If this is the case it can be beneficial to use `Position` (21.16.1) in *`list`* and a bit list than ordered lists to represent subsets of *`list`*.

Chapter 22

Boolean Lists

This chapter describes boolean lists. A *boolean list* is a list that has no holes and contains only the boolean values `true` and `false` (see Chapter 20). In function names we call boolean lists *blists* for brevity.

22.1 IsBlist (Filter)

22.1.1 IsBlist

▷ `IsBlist(`*obj*`)` (Category)

A boolean list ("blist") is a list that has no holes and contains only `true` and `false`. Boolean lists can be represented in an efficient compact form, see 22.5 for details.

Example

```
gap> IsBlist( [ true, true, false, false ] );
true
gap> IsBlist( [] );
true
gap> IsBlist( [false,,true] );  # has holes
false
gap> IsBlist( [1,1,0,0] );      # contains not only boolean values
false
gap> IsBlist( 17 );             # is not even a list
false
```

Boolean lists are lists and all operations for lists are therefore applicable to boolean lists.

Boolean lists can be used in various ways, but maybe the most important application is their use for the description of *subsets* of finite sets. Suppose *set* is a finite set, represented as a list. Then a subset *sub* of *set* is represented by a boolean list *blist* of the same length as *set* such that $blist[i]$ is `true` if $set[i]$ is in *sub*, and `false` otherwise.

22.2 Boolean Lists Representing Subsets

22.2.1 BlistList

▷ BlistList(*list*, *sub*) (function)

returns a new boolean list that describes the list *sub* as a sublist of the dense list *list*. That is BlistList returns a boolean list *blist* of the same length as *list* such that *blist*[*i*] is true if *list*[*i*] is in *sub* and false otherwise.

list need not be a proper set (see 21.19), even though in this case BlistList is most efficient. In particular *list* may contain duplicates. *sub* need not be a proper sublist of *list*, i.e., *sub* may contain elements that are not in *list*. Those elements of course have no influence on the result of BlistList.

Example

```
gap> BlistList( [1..10], [2,3,5,7] );
[ false, true, true, false, true, false, true, false, false, false ]
gap> BlistList( [1,2,3,4,5,2,8,6,4,10], [4,8,9,16] );
[ false, false, false, true, false, false, true, false, true, false ]
```

See also UniteBlistList (22.4.2).

22.2.2 ListBlist

▷ ListBlist(*list*, *blist*) (operation)

returns the sublist *sub* of the list *list*, which must have no holes, represented by the boolean list *blist*, which must have the same length as *list*.

sub contains the element *list*[*i*] if *blist*[*i*] is true and does not contain the element if *blist*[*i*] is false. The order of the elements in *sub* is the same as the order of the corresponding elements in *list*.

Example

```
gap> ListBlist([1..8],[false,true,true,true,true,false,true,true]);
[ 2, 3, 4, 5, 7, 8 ]
gap> ListBlist( [1,2,3,4,5,2,8,6,4,10],
> [false,false,false,true,false,false,true,false,true,false] );
[ 4, 8, 4 ]
```

22.2.3 SizeBlist

▷ SizeBlist(*blist*) (function)

returns the number of entries of the boolean list *blist* that are true. This is the size of the subset represented by the boolean list *blist*.

Example

```
gap> SizeBlist( [ false, true, false, true, false ] );
2
```

22.2.4 IsSubsetBlist

▷ IsSubsetBlist(*blist1, blist2*) (function)

returns `true` if the boolean list *blist2* is a subset of the boolean list *blist1*, which must have equal length, and `false` otherwise. *blist2* is a subset of *blist1* if *blist1*$[i]$ = *blist1*$[i]$ `or` *blist2*$[i]$ for all *i*.

Example

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IsSubsetBlist( blist1, blist2 );
false
gap> blist2 := [ true, false, false, false ];;
gap> IsSubsetBlist( blist1, blist2 );
true
```

22.3 Set Operations via Boolean Lists

22.3.1 UnionBlist

▷ UnionBlist(*blist1, blist2[, ...]*) (function)
▷ UnionBlist(*list*) (function)

In the first form `UnionBlist` returns the union of the boolean lists *blist1*, *blist2*, etc., which must have equal length. The *union* is a new boolean list that contains at position *i* the value *blist1*$[i]$ `or` *blist2*$[i]$ `or` ....

The second form takes the union of all blists (which as for the first form must have equal length) in the list *list*.

22.3.2 IntersectionBlist

▷ IntersectionBlist(*blist1, blist2[, ...]*) (function)
▷ IntersectionBlist(*list*) (function)

In the first form `IntersectionBlist` returns the intersection of the boolean lists *blist1*, *blist2*, etc., which must have equal length. The *intersection* is a new blist that contains at position *i* the value *blist1*$[i]$ `and` *blist2*$[i]$ `and` ....

In the second form *list* must be a list of boolean lists *blist1*, *blist2*, etc., which must have equal length, and `IntersectionBlist` returns the intersection of those boolean lists.

22.3.3 DifferenceBlist

▷ DifferenceBlist(*blist1, blist2*) (function)

returns the asymmetric set difference of the two boolean lists *blist1* and *blist2*, which must have equal length. The *asymmetric set difference* is a new boolean list that contains at position *i* the value *blist1*$[i]$ `and not` *blist2*$[i]$.

Example
```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> UnionBlist( blist1, blist2 );
[ true, true, true, false ]
gap> IntersectionBlist( blist1, blist2 );
[ true, false, false, false ]
gap> DifferenceBlist( blist1, blist2 );
[ false, true, false, false ]
```

22.4 Function that Modify Boolean Lists

22.4.1 UniteBlist

▷ UniteBlist(*blist1, blist2*) (function)

UniteBlist unites the boolean list *blist1* with the boolean list *blist2*, which must have the same length. This is equivalent to assigning *blist1*$[i]$:= *blist1*$[i]$ or *blist2*$[i]$ for all *i*.

UniteBlist returns nothing, it is only called to change *blist1*.

Example
```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> UniteBlist( blist1, blist2 );
gap> blist1;
[ true, true, true, false ]
```

The function UnionBlist (22.3.1) is the nondestructive counterpart to UniteBlist.

22.4.2 UniteBlistList

▷ UniteBlistList(*list, blist, sub*) (function)

works like UniteBlist(*blist*,BlistList(*list*,*sub*)). As no intermediate blist is created, the performance is better than the separate function calls.

22.4.3 IntersectBlist

▷ IntersectBlist(*blist1, blist2*) (function)

intersects the boolean list *blist1* with the boolean list *blist2*, which must have the same length. This is equivalent to assigning *blist1*$[i]$:= *blist1*$[i]$ and *blist2*$[i]$ for all *i*.

IntersectBlist returns nothing, it is only called to change *blist1*.

Example
```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IntersectBlist( blist1, blist2 );
gap> blist1;
[ true, false, false, false ]
```

The function IntersectionBlist (22.3.2) is the nondestructive counterpart to IntersectBlist.

22.4.4 SubtractBlist

▷ SubtractBlist(*blist1, blist2*) (function)

subtracts the boolean list *blist2* from the boolean list *blist1*, which must have equal length. This is equivalent to assigning *blist1*$[i]$:= *blist1*$[i]$ and not *blist2*$[i]$ for all i.

SubtractBlist returns nothing, it is only called to change *blist1*.

Example

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> SubtractBlist( blist1, blist2 );
gap> blist1;
[ false, true, false, false ]
```

The function DifferenceBlist (22.3.3) is the nondestructive counterpart to SubtractBlist.

22.5 More about Boolean Lists

We defined a boolean list as a list that has no holes and contains only true and false. There is a special internal representation for boolean lists that needs only 1 bit for each entry. This bit is set if the entry is true and reset if the entry is false. This representation is of course much more compact than the ordinary representation of lists, which needs 32 or 64 bits per entry.

Not every boolean list is represented in this compact representation. It would be too much work to test every time a list is changed, whether this list has become a boolean list. This section tells you under which circumstances a boolean list is represented in the compact representation, so you can write your functions in such a way that you make best use of the compact representation.

If a dense list containing only true and false is read, it is stored in the compact representation. Furthermore, the results of BlistList (22.2.1), UnionBlist (22.3.1), IntersectionBlist (22.3.2) and DifferenceBlist (22.3.3) are known to be boolean lists by construction, and thus are represented in the compact representation upon creation.

If an argument of IsSubsetBlist (22.2.4), ListBlist (22.2.2), UnionBlist (22.3.1), IntersectionBlist (22.3.2), DifferenceBlist (22.3.3), UniteBlist (22.4.1), IntersectBlist (22.4.3) and SubtractBlist (22.4.4) is a list represented in the ordinary representation, it is tested to see if it is in fact a boolean list. If it is not, an error is signalled. If it is, the representation of the list is changed to the compact representation.

If you change a boolean list that is represented in the compact representation by assignment (see 21.4) or Add (21.4.2) in such a way that the list remains a boolean list it will remain represented in the compact representation. Note that changing a list that is not represented in the compact representation, whether it is a boolean list or not, in such a way that the resulting list becomes a boolean list, will never change the representation of the list.

22.5.1 IsBlistRep

▷ IsBlistRep(*obj*) (Representation)

▷ ConvertToBlistRep(*blist*) (function)

Returns: `true` or `false`

The first function is a filter that returns `true` if the object *`obj`* is a boolean list in compact representation and `false` otherwise, see 22.5.

The second function converts the object *`blist`* to a boolean list in compact representation and returns `true` if this is possible. Otherwise *`blist`* is unchanged and `false` is returned.

Example

```
gap> l := [true, false, true];
[ true, false, true ]
gap> IsBlistRep(l);
true
gap> l := [true, false, 1];
[ true, false, 1 ]
gap> l[3] := false;
false
gap> IsBlistRep(l);
false
gap> ConvertToBlistRep(l);
true
```

Chapter 23

Row Vectors

Just as in mathematics, a vector in GAP is any object which supports appropriate addition and scalar multiplication operations (see Chapter 61). As in mathematics, an especially important class of vectors are those represented by a list of coefficients with respect to some basis. These correspond roughly to the GAP concept of *row vectors*.

23.1 IsRowVector (Filter)

23.1.1 IsRowVector

▷ `IsRowVector(`*obj*`)` (Category)

A *row vector* is a vector (see `IsVector` (31.14.14)) that is also a homogeneous list of odd additive nesting depth (see 21.12). Typical examples are lists of integers and rationals, lists of finite field elements of the same characteristic, and lists of polynomials from a common polynomial ring. Note that matrices are *not* regarded as row vectors, because they have even additive nesting depth.

The additive operations of the vector must thus be compatible with that for lists, implying that the list entries are the coefficients of the vector with respect to some basis.

Note that not all row vectors admit a multiplication via `*` (which is to be understood as a scalar product); for example, class functions are row vectors but the product of two class functions is defined in a different way. For the installation of a scalar product of row vectors, the entries of the vector must be ring elements; note that the default method expects the row vectors to lie in `IsRingElementList`, and this category may not be implied by `IsRingElement` (31.14.16) for all entries of the row vector (see the comment in `IsVector` (31.14.14)).

Note that methods for special types of row vectors really must be installed with the requirement `IsRowVector`, since `IsVector` (31.14.14) may lead to a rank of the method below that of the default method for row vectors (see file `lib/vecmat.gi`).

Example

```
gap> IsRowVector([1,2,3]);
true
```

Because row vectors are just a special case of lists, all operations and functions for lists are applicable to row vectors as well (see Chapter 21). This especially includes accessing elements of a row vector (see 21.3), changing elements of a mutable row vector (see 21.4), and comparing row vectors (see 21.10).

Note that, unless your algorithms specifically require you to be able to change entries of your vectors, it is generally better and faster to work with immutable row vectors. See Section 12.6 for more details.

23.2 Operators for Row Vectors

The rules for arithmetic operations involving row vectors are in fact special cases of those for the arithmetic of lists, as given in Section 21.11 and the following sections, here we reiterate that definition, in the language of vectors.

Note that the additive behaviour sketched below is defined only for lists in the category `IsGeneralizedRowVector` (21.12.1), and the multiplicative behaviour is defined only for lists in the category `IsMultiplicativeGeneralizedRowVector` (21.12.2).

vec1 `+` *vec2*

returns the sum of the two row vectors *vec1* and *vec2*. Probably the most usual situation is that *vec1* and *vec2* have the same length and are defined over a common field; in this case the sum is a new row vector over the same field where each entry is the sum of the corresponding entries of the vectors.

In more general situations, the sum of two row vectors need not be a row vector, for example adding an integer vector *vec1* and a vector *vec2* over a finite field yields the list of pointwise sums, which will be a mixture of finite field elements and integers if *vec1* is longer than *vec2*.

scalar `+` *vec*

vec `+` *scalar*

returns the sum of the scalar *scalar* and the row vector *vec*. Probably the most usual situation is that the elements of *vec* lie in a common field with *scalar*; in this case the sum is a new row vector over the same field where each entry is the sum of the scalar and the corresponding entry of the vector.

More general situations are for example the sum of an integer scalar and a vector over a finite field, or the sum of a finite field element and an integer vector.

Example

```
gap> [ 1, 2, 3 ] + [ 1/2, 1/3, 1/4 ];
[ 3/2, 7/3, 13/4 ]
gap>  [ 1/2, 3/2, 1/2 ] + 1/2;
[ 1, 2, 1 ]
```

vec1 `-` *vec2*

scalar `-` *vec*

vec `-` *scalar*

Subtracting a vector or scalar is defined as adding its additive inverse, so the statements for the addition hold likewise.

Example

```
gap> [ 1, 2, 3 ] - [ 1/2, 1/3, 1/4 ];
[ 1/2, 5/3, 11/4 ]
gap> [ 1/2, 3/2, 1/2 ] - 1/2;
[ 0, 1, 0 ]
```

scalar `*` *vec*

vec `*` *scalar*

returns the product of the scalar *scalar* and the row vector *vec*. Probably the most usual situation is that the elements of *vec* lie in a common field with *scalar*; in this case the product is a new row vector over the same field where each entry is the product of the scalar and the corresponding entry of the vector.

More general situations are for example the product of an integer scalar and a vector over a finite field, or the product of a finite field element and an integer vector.

Example

```
gap> [ 1/2, 3/2, 1/2 ] * 2;
[ 1, 3, 1 ]
```

vec1 * *vec2*

returns the standard scalar product of *vec1* and *vec2*, i.e., the sum of the products of the corresponding entries of the vectors. Probably the most usual situation is that *vec1* and *vec2* have the same length and are defined over a common field; in this case the sum is an element of this field.

More general situations are for example the inner product of an integer vector and a vector over a finite field, or the inner product of two row vectors of different lengths.

Example

```
gap> [ 1, 2, 3 ] * [ 1/2, 1/3, 1/4 ];
23/12
```

For the mutability of results of arithmetic operations, see 12.6.

Further operations with vectors as operands are defined by the matrix operations, see 24.3.

23.2.1 NormedRowVector

▷ NormedRowVector(*v*) (attribute)

returns a scalar multiple `w = c * v` of the row vector *v* with the property that the first nonzero entry of *w* is an identity element in the sense of IsOne (31.10.5).

Example

```
gap> NormedRowVector( [ 5, 2, 3 ] );
[ 1, 2/5, 3/5 ]
```

23.3 Row Vectors over Finite Fields

GAP can use compact formats to store row vectors over fields of order at most 256, based on those used by the Meat-Axe [Rin93]. This format also permits extremely efficient vector arithmetic. On the other hand element access and assignment is significantly slower than for plain lists.

The function ConvertToVectorRep (23.3.1) is used to convert a list into a compressed vector, or to rewrite a compressed vector over another field. Note that this function is *much* faster when it is given a field (or field size) as an argument, rather than having to scan the vector and try to decide the field. Supplying the field can also avoid errors and/or loss of performance, when one vector from some collection happens to have all of its entries over a smaller field than the "natural" field of the problem.

23.3.1 ConvertToVectorRep

▷ ConvertToVectorRep(*list[, field]*) (function)
▷ ConvertToVectorRep(*list[, fieldsize]*) (function)
▷ ConvertToVectorRepNC(*list[, field]*) (function)
▷ ConvertToVectorRepNC(*list[, fieldsize]*) (function)

Called with one argument *list*, ConvertToVectorRep converts *list* to an internal row vector representation if possible.

Called with a list *list* and a finite field *field*, ConvertToVectorRep converts *list* to an internal row vector representation appropriate for a row vector over *field*.

Instead of a *field* also its size *fieldsize* may be given.

It is forbidden to call this function unless *list* is a plain list or a row vector, *field* is a field, and all elements of *list* lie in *field*. Violation of this condition can lead to unpredictable behaviour or a system crash. (Setting the assertion level to at least 2 might catch some violations before a crash, see SetAssertionLevel (7.5.1).)

list may already be a compressed vector. In this case, if no *field* or *fieldsize* is given, then nothing happens. If one is given then the vector is rewritten as a compressed vector over the given *field* unless it has the filter IsLockedRepresentationVector, in which case it is not changed.

The return value is the size of the field over which the vector ends up written, if it is written in a compressed representation.

In this example, we first create a row vector and then ask GAP to rewrite it, first over GF(2) and then over GF(4).

Example

```
gap> v := [Z(2)^0,Z(2),Z(2),0*Z(2)];
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(v);
[ "IsPlistRep", "IsInternalRep" ]
gap> ConvertToVectorRep(v);
2
gap> v;
<a GF2 vector of length 4>
gap> ConvertToVectorRep(v,4);
4
gap> v;
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(v);
[ "IsDataObjectRep", "Is8BitVectorRep" ]
```

A vector in the special representation over GF(2) is always viewed as <a GF2 vector of length ...>. Over fields of orders 3 to 256, a vector of length 10 or less is viewed as the list of its coefficients, but a longer one is abbreviated.

Arithmetic operations (see 21.11 and the following sections) preserve the compression status of row vectors in the sense that if all arguments are compressed row vectors written over the same field and the result is a row vector then also the result is a compressed row vector written over this field.

23.3.2 NumberFFVector

▷ `NumberFFVector(vec, sz)` (operation)

returns an integer that gives the position of the finite field row vector *vec* in the sorted list of all row vectors over the field with *sz* elements in the same dimension as *vec*. `NumberFFVector` returns `fail` if the vector cannot be represented over the field with *sz* elements.

23.4 Coefficient List Arithmetic

The following operations all perform arithmetic on row vectors. given as homogeneous lists of the same length, containing elements of a commutative ring.

There are two reasons for using `AddRowVector` (23.4.1) in preference to arithmetic operators. Firstly, the three argument form has no single-step equivalent. Secondly `AddRowVector` (23.4.1) changes its first argument in-place, rather than allocating a new vector to hold the result, and may thus produce less garbage.

23.4.1 AddRowVector

▷ `AddRowVector(dst, src[, mul[, from, to]])` (operation)

Adds the product of *src* and *mul* to *dst*, changing *dst*. If *from* and *to* are given then only the index range `[ from .. to ]` is guaranteed to be affected. Other indices *may* be affected, if it is more convenient to do so. Even when *from* and *to* are given, *dst* and *src* must be row vectors of the *same* length.

If *mul* is not given either then this operation simply adds *src* to *dst*.

23.4.2 AddCoeffs

▷ `AddCoeffs(list1[, poss1], list2[, poss2[, mul]])` (operation)

`AddCoeffs` adds the entries of *list2*{*poss2*}, multiplied by the scalar *mul*, to *list1*{*poss1*}. Unbound entries in *list1* are assumed to be zero. The position of the right-most non-zero element is returned.

If the ranges *poss1* and *poss2* are not given, they are assumed to span the whole vectors. If the scalar *mul* is omitted, one is used as a default.

Note that it is the responsibility of the caller to ensure that *list2* has elements at position *poss2* and that the result (in *list1*) will be a dense list.

The function is free to remove trailing (right-most) zeros.

Example

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;AddCoeffs(l,m);
4
gap> l;
[ 6, 8, 10, 4 ]
```

23.4.3 MultRowVector

▷ MultRowVector(*list1[, poss1, list2, poss2], mul*) (operation)

The five argument version of this operation replaces *list1*[*poss1*[*i*]] by *mul***list2*[*poss2*[*i*]] for *i* between 1 and Length(*poss1*).

The two-argument version simply multiplies each element of *list1*, in-place, by *mul*.

23.4.4 CoeffsMod

▷ CoeffsMod(*list1[, len1], modulus*) (operation)

returns the coefficient list obtained by reducing the entries in *list1* modulo *modulus*. After reducing it shrinks the list to remove trailing zeroes. If the optional argument *len1* is used, it reduces only first *len1* elements of the list.

Example

```
gap> l:=[1,2,3,4];;CoeffsMod(l,2);
[ 1, 0, 1 ]
```

23.5 Shifting and Trimming Coefficient Lists

The following functions change coefficient lists by shifting or trimming.

23.5.1 LeftShiftRowVector

▷ LeftShiftRowVector(*list, shift*) (operation)

changes *list* by assigning *list*[*i*]:= *list*[*i*+*shift*] and removing the last *shift* entries of the result.

23.5.2 RightShiftRowVector

▷ RightShiftRowVector(*list, shift, fill*) (operation)

changes *list* by assigning *list*[*i*+*shift*]:= *list*[*i*] and filling each of the *shift* first entries with *fill*.

23.5.3 ShrinkRowVector

▷ ShrinkRowVector(*list*) (operation)

removes trailing zeroes from the list *list*.

Example

```
gap> l:=[1,0,0];;ShrinkRowVector(l);l;
[ 1 ]
```

23.5.4 RemoveOuterCoeffs

▷ RemoveOuterCoeffs(*list, coef*) (operation)

removes *coef* at the beginning and at the end of *list* and returns the number of elements removed at the beginning.

Example

```
gap> l:=[1,1,2,1,2,1,1,2,1];; RemoveOuterCoeffs(l,1);
2
gap> l;
[ 2, 1, 2, 1, 1, 2 ]
```

23.6 Functions for Coding Theory

The following functions perform operations on finite fields vectors considered as code words in a linear code.

23.6.1 WeightVecFFE

▷ WeightVecFFE(*vec*) (operation)

returns the weight of the finite field vector *vec*, i.e. the number of nonzero entries.

23.6.2 DistanceVecFFE

▷ DistanceVecFFE(*vec1, vec2*) (operation)

returns the distance between the two vectors *vec1* and *vec2*, which must have the same length and whose elements must lie in a common field. The distance is the number of places where *vec1* and *vec2* differ.

23.6.3 DistancesDistributionVecFFEsVecFFE

▷ DistancesDistributionVecFFEsVecFFE(*vecs, vec*) (operation)

returns the distances distribution of the vector *vec* to the vectors in the list *vecs*. All vectors must have the same length, and all elements must lie in a common field. The distances distribution is a list d of length Length(*vec*)+1, such that the value $d[i]$ is the number of vectors in *vecs* that have distance $i+1$ to *vec*.

23.6.4 DistancesDistributionMatFFEVecFFE

▷ DistancesDistributionMatFFEVecFFE(*mat, F, vec*) (operation)

returns the distances distribution of the vector *vec* to the vectors in the vector space generated by the rows of the matrix *mat* over the finite field *F*. The length of the rows of *mat* and the length of *vec* must be equal, and all entries must lie in *F*. The rows of *mat* must be linearly independent. The

distances distribution is a list d of length `Length(vec)+1`, such that the value $d[i]$ is the number of vectors in the vector space generated by the rows of *mat* that have distance $i+1$ to *vec*.

23.6.5 AClosestVectorCombinationsMatFFEVecFFE

▷ `AClosestVectorCombinationsMatFFEVecFFE(`*mat, f, vec, l, stop*`)` (operation)
▷ `AClosestVectorCombinationsMatFFEVecFFECoords(`*mat, f, vec, l, stop*`)` (operation)

These functions run through the *f*-linear combinations of the vectors in the rows of the matrix *mat* that can be written as linear combinations of exactly *l* rows (that is without using zero as a coefficient). The length of the rows of *mat* and the length of *vec* must be equal, and all elements must lie in the field *f*. The rows of *mat* must be linearly independent. `AClosestVectorCombinationsMatFFEVecFFE` returns a vector from these that is closest to the vector *vec*. If it finds a vector of distance at most *stop*, which must be a nonnegative integer, then it stops immediately and returns this vector.

`AClosestVectorCombinationsMatFFEVecFFECoords` returns a length 2 list containing the same closest vector and also a vector *v* with exactly *l* non-zero entries, such that *v* times *mat* is the closest vector.

23.6.6 CosetLeadersMatFFE

▷ `CosetLeadersMatFFE(`*mat, f*`)` (operation)

returns a list of representatives of minimal weight for the cosets of a code. *mat* must be a *check matrix* for the code, the code is defined over the finite field *f*. All rows of *mat* must have the same length, and all elements must lie in the field *f*. The rows of *mat* must be linearly independent.

23.7 Vectors as coefficients of polynomials

A list of ring elements can be interpreted as a row vector or the list of coefficients of a polynomial. There are a couple of functions that implement arithmetic operations based on these interpretations. GAP contains proper support for polynomials (see 66), the operations described in this section are on a lower level.

The following operations all perform arithmetic on univariate polynomials given by their coefficient lists. These lists can have different lengths but must be dense homogeneous lists containing elements of a commutative ring. Not all input lists may be empty.

In the following descriptions we will always assume that *list1* is the coefficient list of the polynomial *pol1* and so forth. If length parameter *leni* is not given, it is set to the length of *listi* by default.

23.7.1 ValuePol

▷ `ValuePol(`*coeff, x*`)` (function)

Let *coeff* be the coefficients list of a univariate polynomial f, and *x* a ring element. Then `ValuePol` returns the value $f(x)$.

The coefficient of x^i is assumed to be stored at position $i+1$ in the coefficients list.

Example

```
gap> ValuePol([1,2,3],4);
57
```

23.7.2 ProductCoeffs

▷ ProductCoeffs(*list1[, len1], list2[, len2]*) (operation)

Let $p1$ (and $p2$) be polynomials given by the first *len1* (*len2*) entries of the coefficient list *list2* (*list2*). If *len1* and *len2* are omitted, they default to the lengths of *list1* and *list2*. This operation returns the coefficient list of the product of $p1$ and $p2$.

Example

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;ProductCoeffs(l,m);
[ 5, 16, 34, 52, 45, 28 ]
```

23.7.3 ReduceCoeffs

▷ ReduceCoeffs(*list1[, len1], list2[, len2]*) (operation)

Let $p1$ (and $p2$) be polynomials given by the first *len1* (*len2*) entries of the coefficient list *list1* (*list2*). If *len1* and *len2* are omitted, they default to the lengths of *list1* and *list2*. ReduceCoeffs changes *list1* to the coefficient list of the remainder when dividing *p1* by *p2*. This operation changes *list1* which therefore must be a mutable list. The operation returns the position of the last non-zero entry of the result but is not guaranteed to remove trailing zeroes.

Example

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;ReduceCoeffs(l,m);
2
gap> l;
[ 64/49, -24/49, 0, 0 ]
```

23.7.4 ReduceCoeffsMod

▷ ReduceCoeffsMod(*list1[, len1], list2[, len2], modulus*) (operation)

Let $p1$ (and $p2$) be polynomials given by the first *len1* (*len2*) entries of the coefficient list *list1* (*list2*). If *len1* and *len2* are omitted, they default to the lengths of *list1* and *list2*. ReduceCoeffsMod changes *list1* to the coefficient list of the remainder when dividing *p1* by *p2* modulo *modulus*, which must be a positive integer. This operation changes *list1* which therefore must be a mutable list. The operations returns the position of the last non-zero entry of the result but is not guaranteed to remove trailing zeroes.

Example

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;ReduceCoeffsMod(l,m,3);
1
gap> l;
[ 1, 0, 0, 0 ]
```

23.7.5 PowerModCoeffs

▷ PowerModCoeffs(*list1[, len1], exp, list2[, len2]*) (operation)

Let $p1$ and $p2$ be polynomials whose coefficients are given by the first *len1* resp. *len2* entries of the lists *list1* and *list2*, respectively. If *len1* and *len2* are omitted, they default to the lengths of *list1* and *list2*. Let *exp* be a positive integer. PowerModCoeffs returns the coefficient list of the remainder when dividing the *exp*-th power of $p1$ by $p2$. The coefficients are reduced already while powers are computed, therefore avoiding an explosion in list length.

Example

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;PowerModCoeffs(l,5,m);
[ -839462813696/678223072849, -7807439437824/678223072849 ]
```

23.7.6 ShiftedCoeffs

▷ ShiftedCoeffs(*list, shift*) (operation)

produces a new coefficient list new obtained by the rule new[i+*shift*]:= *list*[i] and filling initial holes by the appropriate zero.

Example

```
gap> l:=[1,2,3];;ShiftedCoeffs(l,2);ShiftedCoeffs(l,-2);
[ 0, 0, 1, 2, 3 ]
[ 3 ]
```

Chapter 24

Matrices

Matrices are represented in GAP by lists of row vectors (see 23) (for future changes to this policy see Chapter 26). The vectors must all have the same length, and their elements must lie in a common ring. However, since checking rectangularness can be expensive functions and methods of operations for matrices often will not give an error message for non-rectangular lists of lists –in such cases the result is undefined.

Because matrices are just a special case of lists, all operations and functions for lists are applicable to matrices also (see chapter 21). This especially includes accessing elements of a matrix (see 21.3), changing elements of a matrix (see 21.4), and comparing matrices (see 21.10).

Note that, since a matrix is a list of lists, the behaviour of `ShallowCopy` (12.7.1) for matrices is just a special case of `ShallowCopy` (12.7.1) for lists (see 21.7); called with an immutable matrix *mat*, `ShallowCopy` (12.7.1) returns a mutable matrix whose rows are identical to the rows of *mat*. In particular the rows are still immutable. To get a matrix whose rows are mutable, one can use `List(` *mat*`, ShallowCopy )`.

24.1 InfoMatrix (Info Class)

24.1.1 InfoMatrix

▷ `InfoMatrix` (info class)

The info class for matrix operations is `InfoMatrix`.

24.2 Categories of Matrices

24.2.1 IsMatrix

▷ `IsMatrix(`*obj*`)` (Category)

A *matrix* is a list of lists of equal length whose entries lie in a common ring.

Note that matrices may have different multiplications, besides the usual matrix product there is for example the Lie product. So there are categories such as `IsOrdinaryMatrix` (24.2.2) and `IsLieMatrix` (24.2.3) that describe the matrix multiplication. One can form the product of two matrices only if they support the same multiplication.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
gap> IsMatrix(mat);
true
```

Note also the filter `IsTable` (21.1.4) which may be more appropriate than `IsMatrix` for some purposes.

Note that the empty list `[]` and more complex "empty" structures such as `[[]]` are *not* matrices, although special methods allow them be used in place of matrices in some situations. See `EmptyMatrix` (24.5.3) below.

Example

```
gap> [[0]]*[[]];
[ [  ] ]
gap> IsMatrix([[]]);
false
```

24.2.2 IsOrdinaryMatrix

▷ `IsOrdinaryMatrix(`*obj*`)` (Category)

An *ordinary matrix* is a matrix whose multiplication is the ordinary matrix multiplication.

Each matrix in internal representation is in the category `IsOrdinaryMatrix`, and arithmetic operations with objects in `IsOrdinaryMatrix` produce again matrices in `IsOrdinaryMatrix`.

Note that we want that Lie matrices shall be matrices that behave in the same way as ordinary matrices, except that they have a different multiplication. So we must distinguish the different matrix multiplications, in order to be able to describe the applicability of multiplication, and also in order to form a matrix of the appropriate type as the sum, difference etc. of two matrices which have the same multiplication.

24.2.3 IsLieMatrix

▷ `IsLieMatrix(`*mat*`)` (Category)

A *Lie matrix* is a matrix whose multiplication is given by the Lie bracket. (Note that a matrix with ordinary matrix multiplication is in the category `IsOrdinaryMatrix` (24.2.2).)

Each matrix created by `LieObject` (64.1.1) is in the category `IsLieMatrix`, and arithmetic operations with objects in `IsLieMatrix` produce again matrices in `IsLieMatrix`.

24.3 Operators for Matrices

The rules for arithmetic operations involving matrices are in fact special cases of those for the arithmetic of lists, given in Section 21.11 and the following sections, here we reiterate that definition, in the language of vectors and matrices.

Note that the additive behaviour sketched below is defined only for lists in the category `IsGeneralizedRowVector` (21.12.1), and the multiplicative behaviour is defined only for lists in the category `IsMultiplicativeGeneralizedRowVector` (21.12.2) (see 21.12).

`mat1 + mat2`

returns the sum of the two matrices `mat1` and `mat2`, Probably the most usual situation is that `mat1` and `mat2` have the same dimensions and are defined over a common field; in this case the sum is a new matrix over the same field where each entry is the sum of the corresponding entries of the matrices.

In more general situations, the sum of two matrices need not be a matrix, for example adding an integer matrix `mat1` and a matrix `mat2` over a finite field yields the table of pointwise sums, which will be a mixture of finite field elements and integers if `mat1` has bigger dimensions than `mat2`.

`scalar + mat`

`mat + scalar`

returns the sum of the scalar `scalar` and the matrix `mat`. Probably the most usual situation is that the entries of `mat` lie in a common field with `scalar`; in this case the sum is a new matrix over the same field where each entry is the sum of the scalar and the corresponding entry of the matrix.

More general situations are for example the sum of an integer scalar and a matrix over a finite field, or the sum of a finite field element and an integer matrix.

`mat1 - mat2`

`scalar - mat`

`mat - scalar`

Subtracting a matrix or scalar is defined as adding its additive inverse, so the statements for the addition hold likewise.

`scalar * mat`

`mat * scalar`

returns the product of the scalar `scalar` and the matrix `mat`. Probably the most usual situation is that the elements of `mat` lie in a common field with `scalar`; in this case the product is a new matrix over the same field where each entry is the product of the scalar and the corresponding entry of the matrix.

More general situations are for example the product of an integer scalar and a matrix over a finite field, or the product of a finite field element and an integer matrix.

`vec * mat`

returns the product of the row vector `vec` and the matrix `mat`. Probably the most usual situation is that `vec` and `mat` have the same lengths and are defined over a common field, and that all rows of `mat` have the same length m, say; in this case the product is a new row vector of length m over the same field which is the sum of the scalar multiples of the rows of `mat` with the corresponding entries of `vec`.

More general situations are for example the product of an integer vector and a matrix over a finite field, or the product of a vector over a finite field and an integer matrix.

`mat * vec`

returns the product of the matrix `mat` and the row vector `vec`. (This is the standard product of a matrix with a *column* vector.) Probably the most usual situation is that the length of `vec` and of all rows of `mat` are equal, and that the elements of `mat` and `vec` lie in a common field; in this case the product is a new row vector of the same length as `mat` and over the same field which is the sum of the scalar multiples of the columns of `mat` with the corresponding entries of `vec`.

More general situations are for example the product of an integer matrix and a vector over a finite field, or the product of a matrix over a finite field and an integer vector.

`mat1 * mat2`

This form evaluates to the (Cauchy) product of the two matrices `mat1` and `mat2`. Probably the most usual situation is that the number of columns of `mat1` equals the number of rows of `mat2`, and

that the elements of *mat* and *vec* lie in a common field; if *mat1* is a matrix with *m* rows and *n* columns, say, and *mat2* is a matrix with *n* rows and *o* columns, the result is a new matrix with *m* rows and *o* columns. The element in row *i* at position *j* of the product is the sum of *mat1*[*i*][*l*] ∗ *mat2*[*l*][*j*], with *l* running from 1 to *n*.

`Inverse( mat )`

returns the inverse of the matrix *mat*, which must be an invertible square matrix. If *mat* is not invertible then `fail` is returned.

mat1 / mat2

scalar / mat

mat / scalar

vec / mat

In general, *left / right* is defined as *left * right*`^-1`. Thus in the above forms the right operand must always be invertible.

mat ^ int

mat1 ^ mat2

vec ^ mat

Powering a square matrix *mat* by an integer *int* yields the *int*-th power of *mat*; if *int* is negative then *mat* must be invertible, if *int* is `0` then the result is the identity matrix `One( mat )`, even if *mat* is not invertible.

Powering a square matrix *mat1* by an invertible square matrix *mat2* of the same dimensions yields the conjugate of *mat1* by *mat2*, i.e., the matrix *mat2*`^-1 * mat1 * mat2`.

Powering a row vector *vec* by a matrix *mat* is in every respect equivalent to *vec * mat*. This operations reflects the fact that matrices act naturally on row vectors by multiplication from the right, and that the powering operator is GAP's standard for group actions.

`Comm( mat1, mat2 )`

returns the commutator of the square invertible matrices *mat1* and *mat2* of the same dimensions and over a common field, which is the matrix *mat1*`^-1 * mat2^-1 * mat1 * mat2`.

The following cases are still special cases of the general list arithmetic defined in 21.11.

scalar + matlist

matlist + scalar

scalar - matlist

matlist - scalar

*scalar * matlist*

*matlist * scalar*

matlist / scalar

A scalar *scalar* may also be added, subtracted, multiplied with, or divided into a list *matlist* of matrices. The result is a new list of matrices where each matrix is the result of performing the operation with the corresponding matrix in *matlist*.

*mat * matlist*

*matlist * mat*

A matrix *mat* may also be multiplied with a list *matlist* of matrices. The result is a new list of matrices, where each entry is the product of *mat* and the corresponding entry in *matlist*.

matlist / mat

Dividing a list *matlist* of matrices by an invertible matrix *mat* evaluates to *matlist * mat*`^-1`.

*vec * matlist*

returns the product of the vector *vec* and the list of matrices *mat*. The lengths *l* of *vec* and *matlist* must be equal. All matrices in *matlist* must have the same dimensions. The elements of

vec and the elements of the matrices in *matlist* must lie in a common ring. The product is the sum over *vec*[*i*] * *matlist*[*i*] with *i* running from 1 to *l*.

For the mutability of results of arithmetic operations, see 12.6.

24.4 Properties and Attributes of Matrices

24.4.1 DimensionsMat

▷ DimensionsMat(*mat*) (attribute)

is a list of length 2, the first being the number of rows, the second being the number of columns of the matrix *mat*. If *mat* is malformed, that is, it is not a IsRectangularTable (21.1.5), returns fail.

Example

```
gap> DimensionsMat([[1,2,3],[4,5,6]]);
[ 2, 3 ]
gap> DimensionsMat([[1,2,3],[4,5]]);
fail
```

24.4.2 DefaultFieldOfMatrix

▷ DefaultFieldOfMatrix(*mat*) (attribute)

For a matrix *mat*, DefaultFieldOfMatrix returns either a field (not necessarily the smallest one) containing all entries of *mat*, or fail.

If *mat* is a matrix of finite field elements or a matrix of cyclotomics, DefaultFieldOfMatrix returns the default field generated by the matrix entries (see 59.3 and 18.1).

Example

```
gap> DefaultFieldOfMatrix([[Z(4),Z(8)]]);
GF(2^6)
```

24.4.3 TraceMat

▷ TraceMat(*mat*) (function)

▷ Trace(*mat*) (function)

The trace of a square matrix is the sum of its diagonal entries.

Example

```
gap> TraceMat([[1,2,3],[4,5,6],[7,8,9]]);
15
```

24.4.4 DeterminantMat

▷ DeterminantMat(*mat*) (attribute)

▷ Determinant(*mat*) (attribute)

returns the determinant of the square matrix *mat*.

These methods assume implicitly that *mat* is defined over an integral domain whose quotient field is implemented in GAP. For matrices defined over an arbitrary commutative ring with one see DeterminantMatDivFree (24.4.6).

24.4.5 DeterminantMatDestructive

▷ DeterminantMatDestructive(*mat*) (operation)

Does the same as DeterminantMat (24.4.4), with the difference that it may destroy its argument. The matrix *mat* must be mutable.

Example

```
gap> DeterminantMat([[1,2],[2,1]]);
-3
gap> mm:= [[1,2],[2,1]];;
gap> DeterminantMatDestructive( mm );
-3
gap> mm;
[ [ 1, 2 ], [ 0, -3 ] ]
```

24.4.6 DeterminantMatDivFree

▷ DeterminantMatDivFree(*mat*) (operation)

returns the determinant of a square matrix *mat* over an arbitrary commutative ring with one using the division free method of Mahajan and Vinay [MV97].

24.4.7 IsMonomialMatrix

▷ IsMonomialMatrix(*mat*) (property)

A matrix is monomial if and only if it has exactly one nonzero entry in every row and every column.

Example

```
gap> IsMonomialMatrix([[0,1],[1,0]]);
true
```

24.4.8 IsDiagonalMat

▷ IsDiagonalMat(*mat*) (operation)

returns true if mat has only zero entries off the main diagonal, false otherwise.

24.4.9 IsUpperTriangularMat

▷ IsUpperTriangularMat(*mat*) (operation)

returns true if mat has only zero entries below the main diagonal, false otherwise.

24.4.10 IsLowerTriangularMat

▷ IsLowerTriangularMat(*mat*) (operation)

returns true if mat has only zero entries below the main diagonal, false otherwise.

24.5 Matrix Constructions

24.5.1 IdentityMat

▷ IdentityMat(*m[, R]*) (function)

returns a (mutable) $m \times m$ identity matrix over the ring given by *R*. Here, *R* can be either a ring, or an element of a ring. By default, an integer matrix is created.

Example

```
gap> IdentityMat(3);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> IdentityMat(2,Integers mod 15);
[ [ ZmodnZObj( 1, 15 ), ZmodnZObj( 0, 15 ) ],
  [ ZmodnZObj( 0, 15 ), ZmodnZObj( 1, 15 ) ] ]
gap> IdentityMat(2,Z(3));
[ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ]
```

24.5.2 NullMat

▷ NullMat(*m, n[, R]*) (function)

returns a (mutable) $m \times n$ null matrix over the ring given by by *R*. Here, *R* can be either a ring, or an element of a ring. By default, an integer matrix is created.

Example

```
gap> NullMat(3,2);
[ [ 0, 0 ], [ 0, 0 ], [ 0, 0 ] ]
gap> NullMat(2,2,Integers mod 15);
[ [ ZmodnZObj( 0, 15 ), ZmodnZObj( 0, 15 ) ],
  [ ZmodnZObj( 0, 15 ), ZmodnZObj( 0, 15 ) ] ]
gap> NullMat(3,2,Z(3));
[ [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ] ]
```

24.5.3 EmptyMatrix

▷ EmptyMatrix(*char*) (function)

is an empty (ordinary) matrix in characteristic *char* that can be added to or multiplied with empty lists (representing zero-dimensional row vectors). It also acts (via the operation \^ (31.12.1)) on empty lists.

Example

```
gap> EmptyMatrix(5);
EmptyMatrix( 5 )
gap> AsList(last);
[ ]
```

24.5.4 DiagonalMat

▷ DiagonalMat(*vector*) (function)

returns a diagonal matrix *mat* with the diagonal entries given by *vector*.

Example

```
gap> DiagonalMat([1,2,3]);
[ [ 1, 0, 0 ], [ 0, 2, 0 ], [ 0, 0, 3 ] ]
```

24.5.5 PermutationMat

▷ PermutationMat(*perm, dim[, F]*) (function)

returns a matrix in dimension *dim* over the field given by *F* (i.e. the smallest field containing the element *F* or *F* itself if it is a field) that represents the permutation *perm* acting by permuting the basis vectors as it permutes points.

Example

```
gap> PermutationMat((1,2,3),4,1);
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ] ]
```

24.5.6 TransposedMatImmutable

▷ TransposedMatImmutable(*mat*) (attribute)
▷ TransposedMatAttr(*mat*) (attribute)
▷ TransposedMat(*mat*) (attribute)
▷ TransposedMatMutable(*mat*) (operation)
▷ TransposedMatOp(*mat*) (operation)

These functions all return the transposed of the matrix *mat*, i.e., a matrix *trans* such that *trans*[*i*][*k*] = *mat*[*k*][*i*] holds.

They differ only w.r.t. the mutability of the result.

TransposedMat is an attribute and hence returns an immutable result. TransposedMatMutable is guaranteed to return a new *mutable* matrix.

TransposedMatImmutable and TransposedMatAttr are synonyms of TransposedMat, and TransposedMatOp is a synonym of TransposedMatMutable, in analogy to operations such as Zero (31.10.3).

24.5.7 TransposedMatDestructive

▷ TransposedMatDestructive(*mat*) (operation)

If *mat* is a mutable matrix, then the transposed is computed by swapping the entries in *mat*. In this way *mat* gets changed. In all other cases the transposed is computed by TransposedMat (24.5.6).

Example

```
gap> TransposedMat([[1,2,3],[4,5,6],[7,8,9]]);
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> mm:= [[1,2,3],[4,5,6],[7,8,9]];;
gap> TransposedMatDestructive( mm );
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> mm;
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
```

24.5.8 KroneckerProduct

▷ KroneckerProduct(*mat1, mat2*) (operation)

The Kronecker product of two matrices is the matrix obtained when replacing each entry `a` of *mat1* by the product `a*mat2` in one matrix.

Example

```
gap> KroneckerProduct([[1,2]],[[5,7],[9,2]]);
[ [ 5, 7, 10, 14 ], [ 9, 2, 18, 4 ] ]
```

24.5.9 ReflectionMat

▷ ReflectionMat(*coeffs[, conj][, root]*) (function)

Let *coeffs* be a row vector. `ReflectionMat` returns the matrix of the reflection in this vector.

More precisely, if *coeffs* is the coefficients list of a vector v w.r.t. a basis B (see `Basis` (61.5.2)), say, then the returned matrix describes the reflection in v w.r.t. B as a map on a row space, with action from the right.

The optional argument *root* is a root of unity that determines the order of the reflection. The default is a reflection of order 2. For triflections one should choose a third root of unity etc. (see `E` (18.1.1)).

conj is a function of one argument that conjugates a ring element. The default is `ComplexConjugate` (18.5.2).

The matrix of the reflection in v is defined as

$$M = I_n + \overline{v^{tr}} \cdot (w-1)/(v\overline{v^{tr}}) \cdot v$$

where w equals *root*, n is the length of the coefficient list, and denotes the conjugation.

So v is mapped to wv, with default $-v$, and any vector x with the property $x\overline{v^{tr}} = 0$ is fixed by the reflection.

24.5.10 PrintArray

▷ PrintArray(*array*) (function)

pretty-prints the array *array*.

24.6 Random Matrices

24.6.1 RandomMat

▷ RandomMat(*m, n[, R]*) (function)

`RandomMat` returns a new mutable random matrix with *m* rows and *n* columns with elements taken from the ring *R*, which defaults to `Integers` (14).

Example

```
gap> RandomMat(2,3,GF(3));
[ [ Z(3), Z(3), 0*Z(3) ], [ Z(3), Z(3)^0, Z(3) ] ]
```

24.6.2 RandomInvertibleMat

▷ RandomInvertibleMat(*m[, R]*) (function)

RandomInvertibleMat returns a new mutable invertible random matrix with *m* rows and columns with elements taken from the ring *R*, which defaults to Integers (14).

Example

```
gap> m := RandomInvertibleMat(4);
[ [ 1, -2, -1, 0 ], [ 1, 0, 1, -1 ], [ 0, 2, 0, 4 ],
  [ -1, -3, 1, -4 ] ]
gap> m^-1;
[ [ 1/4, 1/2, -1/8, -1/4 ], [ -1/3, 0, -1/3, -1/3 ],
  [ -1/12, 1/2, 13/24, 5/12 ], [ 1/6, 0, 5/12, 1/6 ] ]
```

24.6.3 RandomUnimodularMat

▷ RandomUnimodularMat(*m*) (function)

returns a new random mutable $m \times m$ matrix with integer entries that is invertible over the integers.

Example

```
gap> m := RandomUnimodularMat(3);
[ [ 1, 0, 0 ], [ 156, -39, -25 ], [ -100, 25, 16 ] ]
gap> m^-1;
[ [ 1, 0, 0 ], [ 4, 16, 25 ], [ 0, -25, -39 ] ]
```

24.7 Matrices Representing Linear Equations and the Gaussian Algorithm

24.7.1 RankMat

▷ RankMat(*mat*) (attribute)

If *mat* is a matrix whose rows span a free module over the ring generated by the matrix entries and their inverses then RankMat returns the dimension of this free module. Otherwise fail is returned.

Note that RankMat may perform a Gaussian elimination. For large rational matrices this may take very long, because the entries may become very large.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> RankMat(mat);
2
```

24.7.2 TriangulizedMat

▷ TriangulizedMat(*mat*) (operation)

▷ RREF(*mat*) (operation)

Computes an upper triangular form of the matrix *mat* via the Gaussian Algorithm. It returns a immutable matrix in upper triangular form. This is sometimes also called “Hermite normal form” or “Reduced Row Echelon Form”. RREF is a synonym for TriangulizedMat.

24.7.3 TriangulizeMat

▷ `TriangulizeMat(`*mat*`)` (operation)

Applies the Gaussian Algorithm to the mutable matrix *mat* and changes *mat* such that it is in upper triangular normal form (sometimes called "Hermite normal form" or "Reduced Row Echelon Form").

Example

```
gap> m:=TransposedMatMutable(mat);
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> TriangulizeMat(m);m;
[ [ 1, 0, -1 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
gap> m:=TransposedMatMutable(mat);
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> TriangulizedMat(m);m;
[ [ 1, 0, -1 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
```

24.7.4 NullspaceMat

▷ `NullspaceMat(`*mat*`)` (attribute)
▷ `TriangulizedNullspaceMat(`*mat*`)` (attribute)

returns a list of row vectors that form a basis of the vector space of solutions to the equation *vec***mat*=0. The result is an immutable matrix. This basis is not guaranteed to be in any specific form.

The variant `TriangulizedNullspaceMat` returns a basis of the nullspace in triangulized form as is often needed for algorithms.

24.7.5 NullspaceMatDestructive

▷ `NullspaceMatDestructive(`*mat*`)` (operation)
▷ `TriangulizedNullspaceMatDestructive(`*mat*`)` (operation)

This function does the same as `NullspaceMat` (24.7.4). However, the latter function makes a copy of *mat* to avoid having to change it. This function does not do that; it returns the nullspace and may destroy *mat*; this saves a lot of memory in case *mat* is big. The matrix *mat* must be mutable.

The variant `TriangulizedNullspaceMatDestructive` returns a basis of the nullspace in triangulized form. It may destroy the matrix *mat*.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> NullspaceMat(mat);
[ [ 1, -2, 1 ] ]
gap> mm:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> NullspaceMatDestructive( mm );
[ [ 1, -2, 1 ] ]
gap> mm;
[ [ 1, 2, 3 ], [ 0, -3, -6 ], [ 0, 0, 0 ] ]
```

24.7.6 SolutionMat

▷ SolutionMat(*mat, vec*) (operation)

returns a row vector *x* that is a solution of the equation *x * mat = vec*. It returns fail if no such vector exists.

24.7.7 SolutionMatDestructive

▷ SolutionMatDestructive(*mat, vec*) (operation)

Does the same as SolutionMat(*mat*, *vec*) except that it may destroy the matrix *mat* and the vector *vec*. The matrix *mat* must be mutable.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> SolutionMat(mat,[3,5,7]);
[ 5/3, 1/3, 0 ]
gap> mm:= [[1,2,3],[4,5,6],[7,8,9]];;
gap> v:= [3,5,7];;
gap> SolutionMatDestructive( mm, v );
[ 5/3, 1/3, 0 ]
gap> mm;
[ [ 1, 2, 3 ], [ 0, -3, -6 ], [ 0, 0, 0 ] ]
gap> v;
[ 0, 0, 0 ]
```

24.7.8 BaseFixedSpace

▷ BaseFixedSpace(*mats*) (function)

BaseFixedSpace returns a list of row vectors that form a base of the vector space V such that $vM = v$ for all v in V and all matrices M in the list *mats*. (This is the common eigenspace of all matrices in *mats* for the eigenvalue 1.)

Example

```
gap> BaseFixedSpace([[[1,2],[0,1]]]);
[ [ 0, 1 ] ]
```

24.8 Eigenvectors and eigenvalues

24.8.1 GeneralisedEigenvalues

▷ GeneralisedEigenvalues(*F, A*) (operation)
▷ GeneralizedEigenvalues(*F, A*) (operation)

The generalised eigenvalues of the matrix *A* over the field *F*.

24.8.2 GeneralisedEigenspaces

▷ `GeneralisedEigenspaces(F, A)` (operation)
▷ `GeneralizedEigenspaces(F, A)` (operation)

The generalised eigenspaces of the matrix `A` over the field `F`.

24.8.3 Eigenvalues

▷ `Eigenvalues(F, A)` (operation)

The eigenvalues of the matrix `A` over the field `F`.

24.8.4 Eigenspaces

▷ `Eigenspaces(F, A)` (operation)

The eigenspaces of the matrix `A` over the field `F`.

24.8.5 Eigenvectors

▷ `Eigenvectors(F, A)` (operation)

The eigenvectors of the matrix `A` over the field `F`.

24.9 Elementary Divisors

See also chapter 25.

24.9.1 ElementaryDivisorsMat

▷ `ElementaryDivisorsMat([ring, ]mat)` (operation)
▷ `ElementaryDivisorsMatDestructive(ring, mat)` (function)

returns a list of the elementary divisors, i.e., the unique d with $d[i]$ divides $d[i+1]$ and `mat` is equivalent to a diagonal matrix with the elements $d[i]$ on the diagonal. The operations are performed over the euclidean ring `ring`, which must contain all matrix entries. For compatibility reasons it can be omitted and defaults to the `DefaultRing` (56.1.3) of the matrix entries.

The function `ElementaryDivisorsMatDestructive` produces the same result but in the process may destroy the contents of `mat`.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> ElementaryDivisorsMat(mat);
[ 1, 3, 0 ]
gap> x:=Indeterminate(Rationals,"x");;
gap> mat:=mat*One(x)-x*mat^0;
[ [ -x+1, 2, 3 ], [ 4, -x+5, 6 ], [ 7, 8, -x+9 ] ]
gap> ElementaryDivisorsMat(PolynomialRing(Rationals,1),mat);
[ 1, 1, x^3-15*x^2-18*x ]
```

```
gap> mat:=KroneckerProduct(CompanionMat((x-1)^2),
>                          CompanionMat((x^3-1)*(x-1)));;
gap> mat:=mat*One(x)-x*mat^0;
[ [ -x, 0, 0, 0, 0, 0, 0, 1 ], [ 0, -x, 0, 0, -1, 0, 0, -1 ],
  [ 0, 0, -x, 0, 0, -1, 0, 0 ], [ 0, 0, 0, -x, 0, 0, -1, -1 ],
  [ 0, 0, 0, -1, -x, 0, 0, -2 ], [ 1, 0, 0, 1, 2, -x, 0, 2 ],
  [ 0, 1, 0, 0, 0, 2, -x, 0 ], [ 0, 0, 1, 1, 0, 0, 2, -x+2 ] ]
gap> ElementaryDivisorsMat(PolynomialRing(Rationals,1),mat);
[ 1, 1, 1, 1, 1, 1, x-1, x^7-x^6-2*x^4+2*x^3+x-1 ]
```

24.9.2 ElementaryDivisorsTransformationsMat

▷ `ElementaryDivisorsTransformationsMat(`*[ring,]mat*`)` (operation)
▷ `ElementaryDivisorsTransformationsMatDestructive(`*ring, mat*`)` (function)

`ElementaryDivisorsTransformations`, in addition to the tasks done by `ElementaryDivisorsMat`, also calculates transforming matrices. It returns a record with components `normal` (a matrix S), `rowtrans` (a matrix P), and `coltrans` (a matrix Q) such that $PAQ = S$. The operations are performed over the euclidean ring *ring*, which must contain all matrix entries. For compatibility reasons it can be omitted and defaults to the `DefaultRing` (56.1.3) of the matrix entries.

The function `ElementaryDivisorsTransformationsMatDestructive` produces the same result but in the process destroys the contents of *mat*.

Example

```
gap> mat:=KroneckerProduct(CompanionMat((x-1)^2),CompanionMat((x^3-1)*(x-1)));;
gap> mat:=mat*One(x)-x*mat^0;
[ [ -x, 0, 0, 0, 0, 0, 0, 1 ], [ 0, -x, 0, 0, -1, 0, 0, -1 ],
  [ 0, 0, -x, 0, 0, -1, 0, 0 ], [ 0, 0, 0, -x, 0, 0, -1, -1 ],
  [ 0, 0, 0, -1, -x, 0, 0, -2 ], [ 1, 0, 0, 1, 2, -x, 0, 2 ],
  [ 0, 1, 0, 0, 0, 2, -x, 0 ], [ 0, 0, 1, 1, 0, 0, 2, -x+2 ] ]
gap> t:=ElementaryDivisorsTransformationsMat(PolynomialRing(Rationals,1),mat);
rec( coltrans := [ [ 0, 0, 0, 0, 0, 0, 1/6*x^2-7/9*x-1/18, -3*x^3-x^2-x-1 ],
      [ 0, 0, 0, 0, 0, 0, -1/6*x^2+x-1, 3*x^3-3*x^2 ],
      [ 0, 0, 0, 0, 0, 1, -1/18*x^4+1/3*x^3-1/3*x^2-1/9*x, x^5-x^4+2*x^2-2*x
        ], [ 0, 0, 0, 0, -1, 0, -1/9*x^3+1/2*x^2+1/9*x, 2*x^4+x^3+x^2+2*x ],
      [ 0, -1, 0, 0, 0, 0, -2/9*x^2+19/18*x, 4*x^3+x^2+x ],
      [ 0, 0, -1, 0, 0, -x, 1/18*x^5-1/3*x^4+1/3*x^3+1/9*x^2,
          -x^6+x^5-2*x^3+2*x^2 ],
      [ 0, 0, 0, -1, x, 0, 1/9*x^4-2/3*x^3+2/3*x^2+1/18*x,
          -2*x^5+2*x^4-x^2+x ],
      [ 1, 0, 0, 0, 0, 0, 1/6*x^3-7/9*x^2-1/18*x, -3*x^4-x^3-x^2-x ] ],
  normal := [ [ 1, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 1, 0, 0, 0, 0 ],
      [ 0, 0, 0, 0, 1, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 1, 0, 0 ],
      [ 0, 0, 0, 0, 0, 0, x-1, 0 ],
      [ 0, 0, 0, 0, 0, 0, 0, x^7-x^6-2*x^4+2*x^3+x-1 ] ],
  rowtrans := [ [ 1, 0, 0, 0, 0, 0, 0, 0 ], [ 1, 1, 0, 0, 0, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0, 0, 0, 0 ], [ 1, 0, 0, 1, 0, 0, 0, 0 ],
      [ -x+2, -x, 0, 0, 1, 0, 0, 0 ],
      [ 2*x^2-4*x+2, 2*x^2-x, 0, 2, -2*x+1, 0, 0, 1 ],
      [ 3*x^3-6*x^2+3*x, 3*x^3-2*x^2, 2, 3*x, -3*x^2+2*x, 0, 1, 2*x ],
```

```
      [ 1/6*x^8-7/6*x^7+2*x^6-4/3*x^5+7/3*x^4-4*x^3+13/6*x^2-7/6*x+2,
          1/6*x^8-17/18*x^7+13/18*x^6-5/18*x^5+35/18*x^4-31/18*x^3+1/9*x^2-x+\
2, 1/9*x^5-5/9*x^4+1/9*x^3-1/9*x^2+14/9*x-1/9,
          1/6*x^6-5/6*x^5+1/6*x^4-1/6*x^3+11/6*x^2-1/6*x,
          -1/6*x^7+17/18*x^6-13/18*x^5+5/18*x^4-35/18*x^3+31/18*x^2-1/9*x+1,
          1, 1/18*x^5-5/18*x^4+1/18*x^3-1/18*x^2+23/18*x-1/18,
          1/9*x^6-5/9*x^5+1/9*x^4-1/9*x^3+14/9*x^2-1/9*x ] ] )
gap> t.rowtrans*mat*t.coltrans;
[ [ 1, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 1, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, x-1, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, x^7-x^6-2*x^4+2*x^3+x-1 ] ]
```

24.9.3 DiagonalizeMat

▷ `DiagonalizeMat(`*ring*`, `*mat*`)` (operation)

brings the mutable matrix *mat*, considered as a matrix over *ring*, into diagonal form by elementary row and column operations.

Example

```
gap> m:=[[1,2],[2,1]];;
gap> DiagonalizeMat(Integers,m);m;
[ [ 1, 0 ], [ 0, 3 ] ]
```

24.10 Echelonized Matrices

24.10.1 SemiEchelonMat

▷ `SemiEchelonMat(`*mat*`)` (attribute)

A matrix over a field F is in semi-echelon form if the first nonzero element in each row is the identity of F, and all values exactly below these pivots are the zero of F.

`SemiEchelonMat` returns a record that contains information about a semi-echelonized form of the matrix *mat*.

The components of this record are

`vectors`
: list of row vectors, each with pivot element the identity of F,

`heads`
: list that contains at position *i*, if nonzero, the number of the row for that the pivot element is in column *i*.

24.10.2 SemiEchelonMatDestructive

▷ `SemiEchelonMatDestructive(`*mat*`)` (operation)

This does the same as SemiEchelonMat(*mat*), except that it may (and probably will) destroy the matrix *mat*.

Example

```
gap> mm:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> SemiEchelonMatDestructive( mm );
rec( heads := [ 1, 2, 0 ], vectors := [ [ 1, 2, 3 ], [ 0, 1, 2 ] ] )
gap> mm;
[ [ 1, 2, 3 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
```

24.10.3 SemiEchelonMatTransformation

▷ SemiEchelonMatTransformation(*mat*) (attribute)

does the same as SemiEchelonMat (24.10.1) but additionally stores the linear transformation T performed on the matrix. The additional components of the result are

coeffs
: a list of coefficients vectors of the vectors component, with respect to the rows of *mat*, that is, coeffs * mat is the vectors component.

relations
: a list of basis vectors for the (left) null space of *mat*.

Example

```
gap> SemiEchelonMatTransformation([[1,2,3],[0,0,1]]);
rec( coeffs := [ [ 1, 0 ], [ 0, 1 ] ], heads := [ 1, 0, 2 ],
  relations := [  ], vectors := [ [ 1, 2, 3 ], [ 0, 0, 1 ] ] )
```

24.10.4 SemiEchelonMats

▷ SemiEchelonMats(*mats*) (operation)

A list of matrices over a field F is in semi-echelon form if the list of row vectors obtained on concatenating the rows of each matrix is a semi-echelonized matrix (see SemiEchelonMat (24.10.1)).

SemiEchelonMats returns a record that contains information about a semi-echelonized form of the list *mats* of matrices.

The components of this record are

vectors
: list of matrices, each with pivot element the identity of F,

heads
: matrix that contains at position [*i*,*j*], if nonzero, the number of the matrix that has the pivot element in this position

24.10.5 SemiEchelonMatsDestructive

▷ SemiEchelonMatsDestructive(*mats*) (operation)

Does the same as SemiEchelonmats, except that it may destroy its argument. Therefore the argument must be a list of matrices that re mutable.

24.11 Matrices as Basis of a Row Space

See also chapter 25

24.11.1 BaseMat

▷ `BaseMat(mat)` (attribute)

returns a basis for the row space generated by the rows of *mat* in the form of an immutable matrix.

24.11.2 BaseMatDestructive

▷ `BaseMatDestructive(mat)` (operation)

Does the same as `BaseMat` (24.11.1), with the difference that it may destroy the matrix *mat*. The matrix *mat* must be mutable.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> BaseMat(mat);
[ [ 1, 2, 3 ], [ 0, 1, 2 ] ]
gap> mm:= [[1,2,3],[4,5,6],[5,7,9]];;
gap> BaseMatDestructive( mm );
[ [ 1, 2, 3 ], [ 0, 1, 2 ] ]
gap> mm;
[ [ 1, 2, 3 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
```

24.11.3 BaseOrthogonalSpaceMat

▷ `BaseOrthogonalSpaceMat(mat)` (attribute)

Let V be the row space generated by the rows of *mat* (over any field that contains all entries of *mat*). `BaseOrthogonalSpaceMat( mat )` computes a base of the orthogonal space of V.

The rows of *mat* need not be linearly independent.

24.11.4 SumIntersectionMat

▷ `SumIntersectionMat(M1, M2)` (operation)

performs Zassenhaus' algorithm to compute bases for the sum and the intersection of spaces generated by the rows of the matrices *M1*, *M2*.

returns a list of length 2, at first position a base of the sum, at second position a base of the intersection. Both bases are in semi-echelon form (see 24.10).

Example

```
gap> SumIntersectionMat(mat,[[2,7,6],[5,9,4]]);
[ [ [ 1, 2, 3 ], [ 0, 1, 2 ], [ 0, 0, 1 ] ], [ [ 1, -3/4, -5/2 ] ] ]
```

24.11.5 BaseSteinitzVectors

▷ BaseSteinitzVectors(*bas, mat*) (function)

find vectors extending mat to a basis spanning the span of *bas*. Both *bas* and *mat* must be matrices of full (row) rank. It returns a record with the following components:

subspace
: s a basis of the space spanned by *mat* in upper triangular form with leading ones at all echelon steps and zeroes above these ones.

factorspace
: is a list of extending vectors in upper triangular form.

factorzero
: is a zero vector.

heads
: is a list of integers which can be used to decompose vectors in the basis vectors. The *i*th entry indicating the vector that gives an echelon step at position *i*. A negative number indicates an echelon step in the subspace, a positive number an echelon step in the complement, the absolute value gives the position of the vector in the lists subspace and factorspace.

Example

```
gap> BaseSteinitzVectors(IdentityMat(3,1),[[11,13,15]]);
rec( factorspace := [ [ 0, 1, 15/13 ], [ 0, 0, 1 ] ],
  factorzero := [ 0, 0, 0 ], heads := [ -1, 1, 2 ],
  subspace := [ [ 1, 13/11, 15/11 ] ] )
```

24.12 Triangular Matrices

24.12.1 DiagonalOfMat

▷ DiagonalOfMat(*mat*) (operation)

returns the diagonal of the matrix *mat*. If *mat* is not a square matrix, then the result has the same length as the rows of *mat*, and is padded with zeros if *mat* has fewer rows than columns.

Example

```
gap> DiagonalOfMat([[1,2,3],[4,5,6]]);
[ 1, 5, 0 ]
```

24.12.2 UpperSubdiagonal

▷ UpperSubdiagonal(*mat, pos*) (operation)

returns a mutable list containing the entries of the *pos*th upper subdiagonal of *mat*.

Example

```
gap> UpperSubdiagonal(mat,1);
[ 2, 6 ]
```

24.12.3 DepthOfUpperTriangularMatrix

▷ DepthOfUpperTriangularMatrix(*mat*) (attribute)

If *mat* is an upper triangular matrix this attribute returns the index of the first nonzero diagonal.

Example

```
gap> DepthOfUpperTriangularMatrix([[0,1,2],[0,0,1],[0,0,0]]);
1
gap> DepthOfUpperTriangularMatrix([[0,0,2],[0,0,0],[0,0,0]]);
2
```

24.13 Matrices as Linear Mappings

24.13.1 CharacteristicPolynomial

▷ CharacteristicPolynomial(*[F, E,]mat[, ind]*) (attribute)

For a square matrix *mat*, CharacteristicPolynomial returns the *characteristic polynomial* of *mat*, that is, the StandardAssociate (56.5.5) of the determinant of the matrix $mat - X \cdot I$, where X is an indeterminate and I is the appropriate identity matrix.

If fields *F* and *E* are given, then *F* must be a subfield of *E*, and *mat* must have entries in *E*. Then CharacteristicPolynomial returns the characteristic polynomial of the *F*-linear mapping induced by *mat* on the underlying *E*-vector space of *mat*. In this case, the characteristic polynomial is computed using BlownUpMat (24.13.3) for the field extension of E/F generated by the default field. Thus, if $F = E$, the result is the same as for the one argument version.

The returned polynomials are expressed in the indeterminate number *ind*. If *ind* is not given, it defaults to 1.

CharacteristicPolynomial(*F*, *E*, *mat*) is a multiple of the minimal polynomial MinimalPolynomial(*F*, *mat*) (see MinimalPolynomial (66.8.1)).

Note that, up to GAP version 4.4.6, CharacteristicPolynomial only allowed to specify one field (corresponding to *F*) as an argument. That usage has been disabled because its definition turned out to be ambiguous and may have lead to unexpected results. (To ensure backward compatibility, it is still possible to use the old form if *F* contains the default field of the matrix, see DefaultFieldOfMatrix (24.4.2), but this feature will disappear in future versions of GAP.)

Example

```
gap> CharacteristicPolynomial( [ [ 1, 1 ], [ 0, 1 ] ] );
x^2-2*x+1
gap> mat := [[0,1],[E(4)-1,E(4)]];;
gap> CharacteristicPolynomial( mat );
x^2+(-E(4))*x+(1-E(4))
gap> CharacteristicPolynomial( Rationals, CF(4), mat );
x^4+3*x^2+2*x+2
gap> mat:= [ [ E(4), 1 ], [ 0, -E(4) ] ];;
gap> CharacteristicPolynomial( mat );
x^2+1
gap> CharacteristicPolynomial( Rationals, CF(4), mat );
x^4+2*x^2+1
```

24.13.2 JordanDecomposition

▷ JordanDecomposition(*mat*) (attribute)

JordanDecomposition(*mat*) returns a list [S,N] such that S is a semisimple matrix and N is nilpotent. Furthermore, S and N commute and *mat*=S+N.

Example

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> JordanDecomposition(mat);
[ [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
  [ [ 0, 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ] ]
```

24.13.3 BlownUpMat

▷ BlownUpMat(*B, mat*) (function)

Let *B* be a basis of a field extension F/K, and *mat* a matrix whose entries are all in F. (This is not checked.) BlownUpMat returns a matrix over K that is obtained by replacing each entry of *mat* by its regular representation w.r.t. *B*.

More precisely, regard *mat* as the matrix of a linear transformation on the row space F^n w.r.t. the F-basis with vectors $(v_1, ldots, v_n)$, say, and suppose that the basis *B* consists of the vectors $(b_1, \ldots, b_m)$; then the returned matrix is the matrix of the linear transformation on the row space K^{mn} w.r.t. the K-basis whose vectors are $(b_1v_1, \ldots b_mv_1, \ldots, b_mv_n)$.

Note that the linear transformations act on *row* vectors, i.e., each row of the matrix is a concatenation of vectors of *B*-coefficients.

24.13.4 BlownUpVector

▷ BlownUpVector(*B, vector*) (function)

Let *B* be a basis of a field extension F/K, and *vector* a row vector whose entries are all in F. BlownUpVector returns a row vector over K that is obtained by replacing each entry of *vector* by its coefficients w.r.t. *B*.

So BlownUpVector and BlownUpMat (24.13.3) are compatible in the sense that for a matrix *mat* over F, BlownUpVector(*B*, *mat* * *vector*) is equal to BlownUpMat(*B*, *mat*) * BlownUpVector(*B*, *vector*).

Example

```
gap> B:= Basis( CF(4), [ 1, E(4) ] );;
gap> mat:= [ [ 1, E(4) ], [ 0, 1 ] ];;  vec:= [ 1, E(4) ];;
gap> bmat:= BlownUpMat( B, mat );;  bvec:= BlownUpVector( B, vec );;
gap> Display( bmat );  bvec;
[ [   1,   0,   0,   1 ],
  [   0,   1,  -1,   0 ],
  [   0,   0,   1,   0 ],
  [   0,   0,   0,   1 ] ]
[ 1, 0, 0, 1 ]
gap> bvec * bmat = BlownUpVector( B, vec * mat );
true
```

24.13.5 CompanionMat

▷ `CompanionMat(`*`poly`*`)` (function)

computes a companion matrix of the polynomial *poly*. This matrix has *poly* as its minimal polynomial.

24.14 Matrices over Finite Fields

Just as for row vectors, (see section 23.3), GAP has a special representation for matrices over small finite fields.

To be eligible to be represented in this way, each row of a matrix must be able to be represented as a compact row vector of the same length over *the same* finite field.

Example

```
gap> v := Z(2)*[1,0,0,1,1];
[ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> ConvertToVectorRep(v,2);
2
gap> v;
<a GF2 vector of length 5>
gap> m := [v];; ConvertToMatrixRep(m,GF(2));; m;
<a 1x5 matrix over GF2>
gap> m := [v,v];; ConvertToMatrixRep(m,GF(2));; m;
<a 2x5 matrix over GF2>
gap> m := [v,v,v];; ConvertToMatrixRep(m,GF(2));; m;
<a 3x5 matrix over GF2>
gap> v := Z(3)*[1..8];
[ Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ]
gap> ConvertToVectorRep(v);
3
gap> m := [v];; ConvertToMatrixRep(m,GF(3));; m;
[ [ Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ] ]
gap> RepresentationsOfObject(m);
[ "IsPositionalObjectRep", "Is8BitMatrixRep" ]
gap> m := [v,v,v,v];; ConvertToMatrixRep(m,GF(3));; m;
< mutable compressed matrix 4x8 over GF(3) >
```

All compressed matrices over GF(2) are viewed as `<a nxm matrix over GF2>`, while over fields GF(q) for q between 3 and 256, matrices with 25 or more entries are viewed in this way, and smaller ones as lists of lists.

Matrices can be converted to this special representation via the following functions.

Note that the main advantage of this special representation of matrices is in low dimensions, where various overheads can be reduced. In higher dimensions, a list of compressed vectors will be almost as fast. Note also that list access and assignment will be somewhat slower for compressed matrices than for plain lists.

In order to form a row of a compressed matrix a vector must accept certain restrictions. Specifically, it cannot change its length or change the field over which it is compressed. The main consequences of this are: that only elements of the appropriate field can be assigned to entries of the vector,

and only to positions between 1 and the original length; that the vector cannot be shared between two matrices compressed over different fields.

This is enforced by the filter `IsLockedRepresentationVector`. When a vector becomes part of a compressed matrix, this filter is set for it. Assignment, `Unbind` (21.5.2), `ConvertToVectorRep` (23.3.1) and `ConvertToMatrixRep` (24.14.2) are all prevented from altering a vector with this filter.

Example

```
gap> v := [Z(2),Z(2)];; ConvertToVectorRep(v,GF(2));; v;
<a GF2 vector of length 2>
gap> m := [v,v];
[ <a GF2 vector of length 2>, <a GF2 vector of length 2> ]
gap> ConvertToMatrixRep(m,GF(2));
2
gap> m2 := [m[1], [Z(4),Z(4)]]; # now try and mix in some GF(4)
[ <a GF2 vector of length 2>, [ Z(2^2), Z(2^2) ] ]
gap> ConvertToMatrixRep(m2); # but m2[1] is locked
#I  ConvertToVectorRep: locked vector not converted to different field
fail
gap> m2 := [ShallowCopy(m[1]), [Z(4),Z(4)]]; # a fresh copy of row 1
[ <a GF2 vector of length 2>, [ Z(2^2), Z(2^2) ] ]
gap> ConvertToMatrixRep(m2); # now it works
4
gap> m2;
[ [ Z(2)^0, Z(2)^0 ], [ Z(2^2), Z(2^2) ] ]
gap> RepresentationsOfObject(m2);
[ "IsPositionalObjectRep", "Is8BitMatrixRep" ]
```

Arithmetic operations (see 21.11 and the following sections) preserve the compression status of matrices in the sense that if all arguments are compressed matrices written over the same field and the result is a matrix then also the result is a compressed matrix written over this field.

There are also two operations that are only available for matrices written over finite fields.

24.14.1 ImmutableMatrix

▷ `ImmutableMatrix(`*field, matrix[, change]*`)` (function)

returns an immutable matrix equal to *matrix* which is in the optimal (concerning space and runtime) representation for matrices defined over *field*. This means that matrices obtained by several calls of `ImmutableMatrix` for the same *field* are compatible for fast arithmetic without need for field conversion.

The input matrix *matrix* or its rows might change the representation, however the result of `ImmutableMatrix` is not necessarily *identical* to *matrix* if a conversion is not possible.

If *change* is `true`, the rows of *matrix* (or *matrix* itself) may be changed to become immutable; otherwise they are copied first).

24.14.2 ConvertToMatrixRep (for a list (and a field))

▷ `ConvertToMatrixRep(`*list[, field]*`)` (function)
▷ `ConvertToMatrixRep(`*list[, fieldsize]*`)` (function)
▷ `ConvertToMatrixRepNC(`*list[, field]*`)` (function)

▷ ConvertToMatrixRepNC(*list[, fieldsize]*) (function)

This function is more technical version of ImmutableMatrix (24.14.1), which will never copy a matrix (or any rows of it) but may fail if it encounters rows locked in the wrong representation, or various other more technical problems. Most users should use ImmutableMatrix (24.14.1) instead. The NC versions of the function do less checking of the argument and may cause unpredictable results or crashes if given unsuitable arguments. Called with one argument *list*, ConvertToMatrixRep converts *list* to an internal matrix representation if possible.

Called with a list *list* and a finite field *field*, ConvertToMatrixRep converts *list* to an internal matrix representation appropriate for a matrix over *field*.

Instead of a *field* also its size *fieldsize* may be given.

It is forbidden to call this function unless all elements of *list* are row vectors with entries in the field *field*. Violation of this condition can lead to unpredictable behaviour or a system crash. (Setting the assertion level to at least 2 might catch some violations before a crash, see SetAssertionLevel (7.5.1).)

list may already be a compressed matrix. In this case, if no *field* or *fieldsize* is given, then nothing happens.

The return value is the size of the field over which the matrix ends up written, if it is written in a compressed representation.

24.14.3 ProjectiveOrder

▷ ProjectiveOrder(*mat*) (attribute)

Returns an integer n and a finite field element e such that *A*^n = eI. *mat* must be a matrix defined over a finite field.

Example

```
gap> ProjectiveOrder([[1,4],[5,2]]*Z(11)^0);
[ 5, Z(11)^5 ]
```

24.14.4 SimultaneousEigenvalues

▷ SimultaneousEigenvalues(*matlist, expo*) (function)

The matrices in *matlist* must be matrices over GF(q) for some prime q. Together, they must generate an abelian p-group of exponent *expo*. Then the eigenvalues of *mat* in the splitting field GF(q^*r*) for some r are powers of an element ξ in the splitting field, which is of order *expo*. SimultaneousEigenvalues returns a matrix of integers mod *expo*, say $(a_{i,j})$, such that the power $\xi^{a_{i,j}}$ is an eigenvalue of the i-th matrix in *matlist* and the eigenspaces of the different matrices to the eigenvalues $\xi^{a_{i,j}}$ for fixed j are equal.

24.15 Inverse and Nullspace of an Integer Matrix Modulo an Ideal

The following two operations deal with matrices over a ring, but only care about the residues of their entries modulo some ring element. In the case of the integers and a prime number p, say, this is effectively computation in a matrix over the prime field in characteristic p.

24.15.1 InverseMatMod

▷ `InverseMatMod(mat, obj)` (operation)

For a square matrix *mat*, `InverseMatMod` returns a matrix *inv* such that *inv * mat* is congruent to the identity matrix modulo *obj*, if such a matrix exists, and `fail` otherwise.

Example

```
gap> mat:= [ [ 1, 2 ], [ 3, 4 ] ];;  inv:= InverseMatMod( mat, 5 );
[ [ 3, 1 ], [ 4, 2 ] ]
gap> mat * inv;
[ [ 11, 5 ], [ 25, 11 ] ]
```

24.15.2 NullspaceModQ

▷ `NullspaceModQ(E, q)` (function)

E must be a matrix of integers and *q* a prime power. Then `NullspaceModQ` returns the set of all vectors of integers modulo *q*, which solve the homogeneous equation system given by *E* modulo *q*.

Example

```
gap> mat:= [ [ 1, 3 ], [ 1, 2 ], [ 1, 1 ] ];;  NullspaceModQ( mat, 5 );
[ [ 0, 0, 0 ], [ 1, 3, 1 ], [ 2, 1, 2 ], [ 4, 2, 4 ], [ 3, 4, 3 ] ]
```

24.16 Special Multiplication Algorithms for Matrices over GF(2)

When multiplying two compressed matrices M and N over GF(2) of dimensions $a \times b$ and $b \times c$, say, where a, b and c are all greater than or equal to 128, GAP by default uses a more sophisticated matrix multiplication algorithm, in which linear combinations of groups of 8 rows of M are remembered and re-used in constructing various rows of the product. This is called level 8 grease. To optimise memory access patterns, these combinations are stored for $(b+255)/256$ sets of 8 rows at once. This number is called the blocking level.

These levels of grease and blocking are found experimentally to give good performance across a range of processors and matrix sizes, but other levels may do even better in some cases. You can control the levels exactly using the functions below.

We plan to include greased blocked matrix multiplication for other finite fields, and greased blocked algorithms for inversion and other matrix operations in a future release.

24.16.1 PROD_GF2MAT_GF2MAT_SIMPLE

▷ `PROD_GF2MAT_GF2MAT_SIMPLE(m1, m2)` (function)

This function performs the standard unblocked and ungreased matrix multiplication for matrices of any size.

24.16.2 PROD_GF2MAT_GF2MAT_ADVANCED

▷ `PROD_GF2MAT_GF2MAT_ADVANCED(m1, m2, g, b)` (function)

This function computes the product of *m1* and *m2*, which must be compressed matrices over GF(2) of compatible dimensions, using level *g* grease and level *b* blocking.

24.17 Block Matrices

Block matrices are a special representation of matrices which can save a lot of memory if large matrices have a block structure with lots of zero blocks. GAP uses the representation IsBlockMatrixRep to store block matrices.

24.17.1 AsBlockMatrix

▷ AsBlockMatrix(*m, nrb, ncb*) (function)

returns a block matrix with *nrb* row blocks and *ncb* column blocks which is equal to the ordinary matrix *m*.

24.17.2 BlockMatrix

▷ BlockMatrix(*blocks, nrb, ncb[, rpb, cpb, zero]*) (function)

BlockMatrix returns an immutable matrix in the sparse representation IsBlockMatrixRep. The nonzero blocks are described by the list *blocks* of triples $[i, j, M(i, j)]$ each consisting of a matrix $M(i, j)$ and its block coordinates in the block matrix to be constructed. All matrices $M(i, j)$ must have the same dimensions. As usual the first coordinate specifies the row and the second one the column. The resulting matrix has *nrb* row blocks and *ncb* column blocks.

If *blocks* is empty (i.e., if the matrix is a zero matrix) then the dimensions of the blocks must be entered as *rpb* and *cpb*, and the zero element as *zero*.

Note that all blocks must be ordinary matrices (see IsOrdinaryMatrix (24.2.2)), and also the block matrix is an ordinary matrix.

Example

```
gap> M := BlockMatrix([[1,1,[[1, 2],[ 3, 4]]],
>                      [1,2,[[9,10],[11,12]]],
>                      [2,2,[[5, 6],[ 7, 8]]]],2,2);
<block matrix of dimensions (2*2)x(2*2)>
gap> Display(M);
[ [   1,   2,   9,  10 ],
  [   3,   4,  11,  12 ],
  [   0,   0,   5,   6 ],
  [   0,   0,   7,   8 ] ]
```

24.17.3 MatrixByBlockMatrix

▷ MatrixByBlockMatrix(*blockmat*) (attribute)

returns a plain ordinary matrix that is equal to the block matrix *blockmat*.

Chapter 25

Integral matrices and lattices

25.1 Linear equations over the integers and Integral Matrices

25.1.1 NullspaceIntMat

▷ `NullspaceIntMat(`*mat*`)` (attribute)

If *mat* is a matrix with integral entries, this function returns a list of vectors that forms a basis of the integral nullspace of *mat*, i.e., of those vectors in the nullspace of *mat* that have integral entries.

Example

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> NullspaceMat(mat);
[ [ -7/4, 9/2, -15/4, 1, 0 ], [ -3/4, -3/2, 1/4, 0, 1 ] ]
gap> NullspaceIntMat(mat);
[ [ 1, 18, -9, 2, -6 ], [ 0, 24, -13, 3, -7 ] ]
```

25.1.2 SolutionIntMat

▷ `SolutionIntMat(`*mat, vec*`)` (operation)

If *mat* is a matrix with integral entries and *vec* a vector with integral entries, this function returns a vector *x* with integer entries that is a solution of the equation *x* `*` *mat* `=` *vec*. It returns `fail` if no such vector exists.

Example

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> SolutionMat(mat,[95,115,182]);
[ 47/4, -17/2, 67/4, 0, 0 ]
gap> SolutionIntMat(mat,[95,115,182]);
[ 2285, -5854, 4888, -1299, 0 ]
```

25.1.3 SolutionNullspaceIntMat

▷ `SolutionNullspaceIntMat(`*mat, vec*`)` (operation)

This function returns a list of length two, its first entry being the result of a call to `SolutionIntMat` (25.1.2) with same arguments, the second the result of `NullspaceIntMat` (25.1.1)

applied to the matrix *mat*. The calculation is performed faster than if two separate calls would be used.

Example

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> SolutionNullspaceIntMat(mat,[95,115,182]);
[ [ 2285, -5854, 4888, -1299, 0 ],
  [ [ 1, 18, -9, 2, -6 ], [ 0, 24, -13, 3, -7 ] ] ]
```

25.1.4 BaseIntMat

▷ `BaseIntMat(`*mat*`)` (attribute)

If *mat* is a matrix with integral entries, this function returns a list of vectors that forms a basis of the integral row space of *mat*, i.e. of the set of integral linear combinations of the rows of *mat*.

Example

```
gap> mat:=[[1,2,7],[4,5,6],[10,11,19]];;
gap> BaseIntMat(mat);
[ [ 1, 2, 7 ], [ 0, 3, 7 ], [ 0, 0, 15 ] ]
```

25.1.5 BaseIntersectionIntMats

▷ `BaseIntersectionIntMats(`*m, n*`)` (attribute)

If *m* and *n* are matrices with integral entries, this function returns a list of vectors that forms a basis of the intersection of the integral row spaces of *m* and *n*.

Example

```
gap> nat:=[[5,7,2],[4,2,5],[7,1,4]];;
gap> BaseIntMat(nat);
[ [ 1, 1, 15 ], [ 0, 2, 55 ], [ 0, 0, 64 ] ]
gap> BaseIntersectionIntMats(mat,nat);
[ [ 1, 5, 509 ], [ 0, 6, 869 ], [ 0, 0, 960 ] ]
```

25.1.6 ComplementIntMat

▷ `ComplementIntMat(`*full, sub*`)` (attribute)

Let *full* be a list of integer vectors generating an integral row module M and *sub* a list of vectors defining a submodule S of M. This function computes a free basis for M that extends S. I.e., if the dimension of S is n it determines a basis $B = \{b_1, \ldots, b_m\}$ for M, as well as n integers x_i such that the n vectors $s_i := x_i \cdot b_i$ form a basis for S.

It returns a record with the following components:

`complement`
: the vectors b_{n+1} up to b_m (they generate a complement to S).

`sub` the vectors s_i (a basis for S).

`moduli`
: the factors x_i.

Example

```
gap> m:=IdentityMat(3);;
gap> n:=[[1,2,3],[4,5,6]];;
gap> ComplementIntMat(m,n);
rec( complement := [ [ 0, 0, 1 ] ], moduli := [ 1, 3 ],
  sub := [ [ 1, 2, 3 ], [ 0, 3, 6 ] ] )
```

25.2 Normal Forms over the Integers

This section describes the computation of the Hermite and Smith normal form of integer matrices.

The Hermite Normal Form (HNF) H of an integer matrix A is a row equivalent upper triangular form such that all off-diagonal entries are reduced modulo the diagonal entry of the column they are in. There exists a unique unimodular matrix Q such that $QA = H$.

The Smith Normal Form S of an integer matrix A is the unique equivalent diagonal form with S_i dividing S_j for $i < j$. There exist unimodular integer matrices P, Q such that $PAQ = S$.

All routines described in this section build on the "workhorse" routine `NormalFormIntMat` (25.2.9).

25.2.1 TriangulizedIntegerMat

▷ `TriangulizedIntegerMat(`*mat*`)` (operation)

Computes an upper triangular form of a matrix with integer entries. It returns a immutable matrix in upper triangular form.

25.2.2 TriangulizedIntegerMatTransform

▷ `TriangulizedIntegerMatTransform(`*mat*`)` (operation)

Computes an upper triangular form of a matrix with integer entries. It returns a record with a component `normal` (an immutable matrix in upper triangular form) and a component `rowtrans` that gives the transformations done to the original matrix to bring it into upper triangular form.

25.2.3 TriangulizeIntegerMat

▷ `TriangulizeIntegerMat(`*mat*`)` (operation)

Changes *mat* to be in upper triangular form. (The result is the same as that of `TriangulizedIntegerMat` (25.2.1), but *mat* will be modified, thus using less memory.) If *mat* is immutable an error will be triggered.

Example

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> TriangulizedIntegerMat(m);
[ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
gap> n:=TriangulizedIntegerMatTransform(m);
rec( normal := [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rank := 3, rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ 1, 0, 0 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
```

```
    rowtrans := [ [ 1, 0, 0 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
    signdet := 1 )
  gap> n.rowtrans*m=n.normal;
  true
  gap> TriangulizeIntegerMat(m); m;
  [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
```

25.2.4 HermiteNormalFormIntegerMat

▷ `HermiteNormalFormIntegerMat(`*mat*`)` (operation)

This operation computes the Hermite normal form of a matrix *mat* with integer entries. It returns a immutable matrix in HNF.

25.2.5 HermiteNormalFormIntegerMatTransform

▷ `HermiteNormalFormIntegerMatTransform(`*mat*`)` (operation)

This operation computes the Hermite normal form of a matrix *mat* with integer entries. It returns a record with components `normal` (a matrix H) and `rowtrans` (a matrix Q) such that $QA = H$.

Example

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> HermiteNormalFormIntegerMat(m);
[ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
gap> n:=HermiteNormalFormIntegerMatTransform(m);
rec( normal := [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ], rank := 3,
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  signdet := 1 )
gap> n.rowtrans*m=n.normal;
true
```

25.2.6 SmithNormalFormIntegerMat

▷ `SmithNormalFormIntegerMat(`*mat*`)` (operation)

This operation computes the Smith normal form of a matrix *mat* with integer entries. It returns a new immutable matrix in the Smith normal form.

25.2.7 SmithNormalFormIntegerMatTransforms

▷ `SmithNormalFormIntegerMatTransforms(`*mat*`)` (operation)

This operation computes the Smith normal form of a matrix *mat* with integer entries. It returns a record with components `normal` (a matrix S), `rowtrans` (a matrix P), and `coltrans` (a matrix Q) such that $PAQ = S$.

25.2.8 DiagonalizeIntMat

▷ DiagonalizeIntMat(*mat*) (operation)

This function changes *mat* to its SNF. (The result is the same as that of SmithNormalFormIntegerMat (25.2.6), but *mat* will be modified, thus using less memory.) If *mat* is immutable an error will be triggered.

Example

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> SmithNormalFormIntegerMat(m);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
gap> n:=SmithNormalFormIntegerMatTransforms(m);
rec( colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  colQ := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ],
  coltrans := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ],
  normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ], rank := 3,
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  signdet := 1 )
gap> n.rowtrans*m*n.coltrans=n.normal;
true
gap> DiagonalizeIntMat(m);m;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

25.2.9 NormalFormIntMat

▷ NormalFormIntMat(*mat, options*) (operation)

This general operation for computation of various Normal Forms is probably the most efficient. Options bit values:

0/1 Triangular Form / Smith Normal Form.

2 Reduce off diagonal entries.

4 Row Transformations.

8 Col Transformations.

16 Destructive (the original matrix may be destroyed)

Compute a Triangular, Hermite or Smith form of the $n \times m$ integer input matrix A. Optionally, compute $n \times n$ and $m \times m$ unimodular transforming matrices Q, P which satisfy $QA = H$ or $QAP = S$.

Note option is a value ranging from 0 to 15 but not all options make sense (e.g., reducing off diagonal entries with SNF option selected already). If an option makes no sense it is ignored.

Returns a record with component `normal` containing the computed normal form and optional components `rowtrans` and/or `coltrans` which hold the respective transformation matrix. Also in the record are components holding the sign of the determinant, `signdet`, and the rank of the matrix, `rank`.

Example

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> NormalFormIntMat(m,0);  # Triangular, no transforms
rec( normal := [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rank := 3, signdet := 1 )
gap> NormalFormIntMat(m,6);  # Hermite Normal Form with row transforms
rec( normal := [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ], rank := 3,
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  signdet := 1 )
gap> NormalFormIntMat(m,13); # Smith Normal Form with both transforms
rec( colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  colQ := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ],
  coltrans := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ],
  normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ], rank := 3,
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  signdet := 1 )
gap> last.rowtrans*m*last.coltrans;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

25.2.10 AbelianInvariantsOfList

▷ AbelianInvariantsOfList(*list*) (attribute)

Given a list of nonnegative integers, this routine returns a sorted list containing the prime power factors of the positive entries in the original list, as well as all zeroes of the original list.

Example

```
gap> AbelianInvariantsOfList([4,6,2,0,12]);
[ 0, 2, 2, 3, 3, 4, 4 ]
```

25.3 Determinant of an integer matrix

25.3.1 DeterminantIntMat

▷ DeterminantIntMat(*mat*) (operation)

Computes the determinant of an integer matrix using the same strategy as NormalFormIntMat (25.2.9). This method is faster in general for matrices greater than 20×20 but quite a lot slower for smaller matrices. It therefore passes the work to the more general DeterminantMat (24.4.4) for these smaller matrices.

25.4 Decompositions

For computing the decomposition of a vector of integers into the rows of a matrix of integers, with integral coefficients, one can use p-adic approximations, as follows.

Let A be a square integral matrix, and p an odd prime. The reduction of A modulo p is $\overline{A}$, its entries are chosen in the interval $[-(p-1)/2,(p-1)/2]$. If $\overline{A}$ is regular over the field with p elements, we can form $A' = \overline{A}^{-1}$. Now we consider the integral linear equation system $xA = b$, i.e., we look for an integral solution x. Define $b_0 = b$, and then iteratively compute

$$x_i = (b_i A') \bmod p, b_{i+1} = (b_i - x_i A)/p, i = 0,1,2,\ldots.$$

By induction, we get

$$p^{i+1} b_{i+1} + \left(\sum_{j=0}^{i} p^j x_j \right) A = b.$$

If there is an integral solution x then it is unique, and there is an index l such that b_{l+1} is zero and $x = \sum_{j=0}^{l} p^j x_j$.

There are two useful generalizations of this idea. First, A need not be square; it is only necessary that there is a square regular matrix formed by a subset of columns of A. Second, A does not need to be integral; the entries may be cyclotomic integers as well, in this case one can replace each column of A by the columns formed by the coefficients w.r.t. an integral basis (which are integers). Note that this preprocessing must be performed compatibly for A and b.

GAP provides the following functions for this purpose (see also `InverseMatMod` (24.15.1)).

25.4.1 Decomposition

▷ `Decomposition(`*`A, B, depth`*`)` (function)

For a $m \times n$ matrix `A` of cyclotomics that has rank $m \leq n$, and a list `B` of cyclotomic vectors, each of length n, `Decomposition` tries to find integral solutions of the linear equation systems `x * A = B[i]`, by computing the p-adic series of hypothetical solutions.

`Decomposition( A, B, depth )`, where *`depth`* is a nonnegative integer, computes for each vector `B[i]` the initial part $\sum_{k=0}^{depth} x_k p^k$, with all x_k vectors of integers with entries bounded by $\pm(p-1)/2$. The prime p is set to 83 first; if the reduction of `A` modulo p is singular, the next prime is chosen automatically.

A list `X` is returned. If the computed initial part for `x * A = B[i]` *is* a solution, we have `X[i] = x`, otherwise `X[i] = fail`.

If *`depth`* is not an integer then it must be the string `"nonnegative"`. `Decomposition( A, B, "nonnegative" )` assumes that the solutions have only nonnegative entries, and that the first column of `A` consists of positive integers. This is satisfied, e.g., for the decomposition of ordinary characters into Brauer characters. In this case the necessary number *`depth`* of iterations can be computed; the `i`-th entry of the returned list is `fail` if there *exists* no nonnegative integral solution of the system `x * A = B[i]`, and it is the solution otherwise.

Note that the result is a list of `fail` if `A` has not full rank, even if there might be a unique integral solution for some equation system.

25.4.2 LinearIndependentColumns

▷ `LinearIndependentColumns(`*`mat`*`)` (function)

Called with a matrix *`mat`*, `LinearIndependentColumns` returns a maximal list of column positions such that the restriction of *`mat`* to these columns has the same rank as *`mat`*.

25.4.3 PadicCoefficients

▷ PadicCoefficients(*A, Amodpinv, b, prime, depth*) (function)

Let *A* be an integral matrix, *prime* a prime integer, *Amodpinv* an inverse of *A* modulo *prime*, *b* an integral vector, and *depth* a nonnegative integer. PadicCoefficients returns the list $[x_0, x_1, \ldots, x_l, b_{l+1}]$ describing the *prime*-adic approximation of *b* (see above), where $l =$ *depth* or l is minimal with the property that $b_{l+1} = 0$.

25.4.4 IntegralizedMat

▷ IntegralizedMat(*A[, inforec]*) (function)

IntegralizedMat returns, for a matrix *A* of cyclotomics, a record intmat with components mat and inforec. Each family of algebraic conjugate columns of *A* is encoded in a set of columns of the rational matrix intmat.mat by replacing cyclotomics in *A* by their coefficients w.r.t. an integral basis. intmat.inforec is a record containing the information how to encode the columns.

If the only argument is *A*, the value of the component inforec is computed that can be entered as second argument *inforec* in a later call of IntegralizedMat with a matrix *B* that shall be encoded compatibly with *A*.

25.4.5 DecompositionInt

▷ DecompositionInt(*A, B, depth*) (function)

DecompositionInt does the same as Decomposition (25.4.1), except that *A* and *B* must be integral matrices, and *depth* must be a nonnegative integer.

25.5 Lattice Reduction

25.5.1 LLLReducedBasis

▷ LLLReducedBasis(*[L,]vectors[, y][, "linearcomb"][, lllout]*) (function)

provides an implementation of the *LLL algorithm* by Lenstra, Lenstra and Lovász (see [LLJL82], [Poh87]). The implementation follows the description in [Coh93, p. 94f.].

LLLReducedBasis returns a record whose component basis is a list of LLL reduced linearly independent vectors spanning the same lattice as the list *vectors*. *L* must be a lattice, with scalar product of the vectors *v* and *w* given by ScalarProduct(*L*, *v*, *w*). If no lattice is specified then the scalar product of vectors given by ScalarProduct(*v*, *w*) is used.

In the case of the option "linearcomb", the result record contains also the components relations and transformation, with the following meaning. relations is a basis of the relation space of *vectors*, i.e., of vectors *x* such that *x* * *vectors* is zero. transformation gives the expression of the new lattice basis in terms of the old, i.e., transformation * *vectors* equals the basis component of the result.

Another optional argument is *y*, the "sensitivity" of the algorithm, a rational number between $1/4$ and 1 (the default value is $3/4$).

The optional argument *lllout* is a record with the components mue and B, both lists of length k, with the meaning that if *lllout* is present then the first k vectors in *vectors* form an LLL reduced basis of the lattice they generate, and *lllout*.mue and *lllout*.B contain their scalar products and norms used internally in the algorithm, which are also present in the output of LLLReducedBasis. So *lllout* can be used for "incremental" calls of LLLReducedBasis.

The function LLLReducedGramMat (25.5.2) computes an LLL reduced Gram matrix.

Example

```
gap> vectors:= [ [ 9, 1, 0, -1, -1 ], [ 15, -1, 0, 0, 0 ],
>                [ 16, 0, 1, 1, 1 ], [ 20, 0, -1, 0, 0 ],
>                [ 25, 1, 1, 0, 0 ] ];;
gap> LLLReducedBasis( vectors, "linearcomb" );
rec( B := [ 5, 36/5, 12, 50/3 ],
  basis := [ [ 1, 1, 1, 1, 1 ], [ 1, 1, -2, 1, 1 ],
      [ -1, 3, -1, -1, -1 ], [ -3, 1, 0, 2, 2 ] ],
  mue := [ [  ], [ 2/5 ], [ -1/5, 1/3 ], [ 2/5, 1/6, 1/6 ] ],
  relations := [ [ -1, 0, -1, 0, 1 ] ],
  transformation := [ [ 0, -1, 1, 0, 0 ], [ -1, -2, 0, 2, 0 ],
      [ 1, -2, 0, 1, 0 ], [ -1, -2, 1, 1, 0 ] ] )
```

25.5.2 LLLReducedGramMat

▷ LLLReducedGramMat(*G[, y]*) (function)

LLLReducedGramMat provides an implementation of the *LLL algorithm* by Lenstra, Lenstra and Lovász (see [LLJL82], [Poh87]). The implementation follows the description in [Coh93, p. 94f.].

Let *G* the Gram matrix of the vectors $(b_1, b_2, \ldots, b_n)$; this means *G* is either a square symmetric matrix or lower triangular matrix (only the entries in the lower triangular half are used by the program).

LLLReducedGramMat returns a record whose component remainder is the Gram matrix of the LLL reduced basis corresponding to $(b_1, b_2, \ldots, b_n)$. If *G* is a lower triangular matrix then also the remainder component of the result record is a lower triangular matrix.

The result record contains also the components relations and transformation, which have the following meaning.

relations is a basis of the space of vectors $(x_1, x_2, \ldots, x_n)$ such that $\sum_{i=1}^{n} x_i b_i$ is zero, and transformation gives the expression of the new lattice basis in terms of the old, i.e., transformation is the matrix T such that $T \cdot G \cdot T^{tr}$ is the remainder component of the result.

The optional argument y denotes the "sensitivity" of the algorithm, it must be a rational number between $1/4$ and 1; the default value is $y = 3/4$.

The function LLLReducedBasis (25.5.1) computes an LLL reduced basis.

Example

```
gap> g:= [ [ 4, 6, 5, 2, 2 ], [ 6, 13, 7, 4, 4 ],
>    [ 5, 7, 11, 2, 0 ], [ 2, 4, 2, 8, 4 ], [ 2, 4, 0, 4, 8 ] ];;
gap> LLLReducedGramMat( g );
rec( B := [ 4, 4, 75/16, 168/25, 32/7 ],
  mue := [ [  ], [ 1/2 ], [ 1/4, -1/8 ], [ 1/2, 1/4, -2/25 ],
      [ -1/4, 1/8, 37/75, 8/21 ] ], relations := [  ],
  remainder := [ [ 4, 2, 1, 2, -1 ], [ 2, 5, 0, 2, 0 ],
      [ 1, 0, 5, 0, 2 ], [ 2, 2, 0, 8, 2 ], [ -1, 0, 2, 2, 7 ] ],
  transformation := [ [ 1, 0, 0, 0, 0 ], [ -1, 1, 0, 0, 0 ],
      [ -1, 0, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ], [ -2, 0, 1, 0, 1 ] ] )
```

25.6 Orthogonal Embeddings

25.6.1 OrthogonalEmbeddings

▷ `OrthogonalEmbeddings(`*gram*`[, "positive"][, `*maxdim*`])` (function)

computes all possible orthogonal embeddings of a lattice given by its Gram matrix *gram*, which must be a regular symmetric matrix of integers. In other words, all integral solutions X of the equation $X^{tr} \cdot X =$*gram* are calculated. The implementation follows the description in [Ple95].

Usually there are many solutions X but all their rows belong to a small set of vectors, so `OrthogonalEmbeddings` returns the solutions encoded by a record with the following components.

`vectors`
: the list $L = [x_1, x_2, \ldots, x_n]$ of vectors that may be rows of a solution, up to sign; these are exactly the vectors with the property $x_i \cdot$*gram*$^{-1} \cdot x_i^{tr} \leq 1$, see `ShortestVectors` (25.6.2),

`norms`
: the list of values $x_i \cdot$*gram*$^{-1} \cdot x_i^{tr}$, and

`solutions`
: a list S of index lists; the i-th solution matrix is L`{ }`$S[i]$`{ }`, so the dimension of the i-th solution is the length of $S[i]$, and we have *gram*$= \sum_{j \in S[i]} x_j^{tr} \cdot x_j$,

The optional argument `"positive"` will cause `OrthogonalEmbeddings` to compute only vectors x_i with nonnegative entries. In the context of characters this is allowed (and useful) if *gram* is the matrix of scalar products of ordinary characters.

When `OrthogonalEmbeddings` is called with the optional argument *maxdim* (a positive integer), only solutions up to dimension *maxdim* are computed; this may accelerate the algorithm.

Example

```
gap> b:= [ [ 3, -1, -1 ], [ -1, 3, -1 ], [ -1, -1, 3 ] ];;
gap> c:=OrthogonalEmbeddings( b );
rec( norms := [ 1, 1, 1, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2 ],
  solutions := [ [ 1, 2, 3 ], [ 1, 6, 6, 7, 7 ], [ 2, 5, 5, 8, 8 ],
      [ 3, 4, 4, 9, 9 ], [ 4, 5, 6, 7, 8, 9 ] ],
  vectors := [ [ -1, 1, 1 ], [ 1, -1, 1 ], [ -1, -1, 1 ],
      [ -1, 1, 0 ], [ -1, 0, 1 ], [ 1, 0, 0 ], [ 0, -1, 1 ],
      [ 0, 1, 0 ], [ 0, 0, 1 ] ] )
gap> c.vectors{ c.solutions[1] };
[ [ -1, 1, 1 ], [ 1, -1, 1 ], [ -1, -1, 1 ] ]
```

gram may be the matrix of scalar products of some virtual characters. From the characters and the embedding given by the matrix X, `Decreased` (72.10.7) may be able to compute irreducibles.

25.6.2 ShortestVectors

▷ `ShortestVectors(`*G*`, `*m*`[, "positive"])` (function)

Let *G* be a regular matrix of a symmetric bilinear form, and *m* a nonnegative integer. `ShortestVectors` computes the vectors x that satisfy $x \cdot G \cdot x^{tr} \leq$ *m*, and returns a record describing these vectors. The result record has the components

`vectors`
: list of the nonzero vectors x, but only one of each pair $(x, -x)$,

`norms`
: list of norms of the vectors according to the Gram matrix `G`.

If the optional argument `"positive"` is entered, only those vectors x with nonnegative entries are computed.

Example

```
gap> g:= [ [ 2, 1, 1 ], [ 1, 2, 1 ], [ 1, 1, 2 ] ];;
gap> ShortestVectors(g,4);
rec( norms := [ 4, 2, 2, 4, 2, 4, 2, 2, 2 ],
  vectors := [ [ -1, 1, 1 ], [ 0, 0, 1 ], [ -1, 0, 1 ], [ 1, -1, 1 ],
      [ 0, -1, 1 ], [ -1, -1, 1 ], [ 0, 1, 0 ], [ -1, 1, 0 ],
      [ 1, 0, 0 ] ] )
```

Chapter 26

Vector and matrix objects

This chapter is work in progress. It will eventually describe the new interface to vector and matrix objects.

Traditionally, vectors in GAP have been lists and matrices have been lists of lists (of equal length). Unfortunately, such lists cannot store their type and so it is impossible to use the full advantages of GAP's method selection on them. This situation is unsustainable in the long run since more special representations (compressed, sparse, etc.) have already been and even more will be implemented. To eventually solve this problem, this chapter describes a new programming interface to vectors and matrices.

26.1 Fundamental ideas and rules

The whole idea of this interface is that (row-) vectors and matrices must be proper objects with a stored type (i.e. created by Objectify allowing inheritance) to benefit from method selection. We therefore refer to the new style vectors and matrices as "vector objects" and "matrix objects" respectively.

It should be possible to write (efficient) code that is independent of the actual representation (in the sense of GAP's representation filters) and preserves it.

This latter requirement makes it necessary to distinguish between (at least) two classes of matrices:

- "RowList"-Matrices which behave basically like lists of rows, in particular are the rows individual GAP objects that can be shared between different matrix objects.
- "Flat" matrices which behave basically like one GAP object that cannot be split up further. In particular a row is only a part of a matrix and no GAP object in itself.

For various reasons these two classes have to be distinguished already with respect to the definition of the operations for them.

In particular vectors and matrices know their BaseDomain and their dimensions. Note that the basic condition is that the elements of vectors and matrices must either lie in the BaseDomain or naturally embed in the sense that + and * and = automatically work with all elements of the base domain (example: integers in polynomials over integers).

Vectors are equal with respect to "=" if they have the same length and the same entries. It is not necessary that they have the same BaseDomain. Matrices are equal with respect to "=" if they have the same dimensions and the same entries. It is possible that not for all pairs of representations methods exist.

It is not guaranteed that all rows of a matrix have the same vector type! It is for example thinkable that a matrix stores some of its rows in a sparse representation and some in a dense one! However, it is guaranteed that the rows of matrices in the same representation are compatible in the sense that all vector operations defined in this interface can be applied to them and that new matrices in the same representation as the original matrix can be formed out of them.

Note that there is neither a default mapping from the set of matrix representations to the set of vector representations nor one in the reverse direction! There is nothing like an "associated" vector representation to a matrix representation or vice versa.

The way to write code that preserves the representation basically works by using constructing operations that take template objects to decide about the actual representation of the new object.

Vectors do not have to be lists in the sense that they do not have to support all list operations. The same holds for matrices. However, RowList matrices behave nearly like lists of row vectors that insist on being dense and containing only vectors of the same length and with the same BaseDomain.

There are some rules embedded in the comments to the following code. They are marked with the word "Rule". FIXME: Collect all rules here.

26.2 Categories of vectors and matrices

26.3 Constructing vector and matrix objects

26.4 Operations for row vector objects

26.5 Operations for row list matrix objects

26.6 Operations for flat matrix objects

Chapter 27

Strings and Characters

27.1 IsChar and IsString

27.1.1 IsChar

▷ IsChar(*obj*) (Category)
▷ IsCharCollection(*obj*) (Category)

A *character* is simply an object in GAP that represents an arbitrary character from the character set of the operating system. Character literals can be entered in GAP by enclosing the character in *singlequotes* '.

Example

```
gap> x:= 'a';  IsChar( x );
'a'
true
gap> '*';
'*'
```

27.1.2 IsString

▷ IsString(*obj*) (filter)

A *string* is a dense list (see IsList (21.1.1), IsDenseList (21.1.2)) of characters (see IsChar (27.1.1)); thus strings are always homogeneous (see IsHomogeneousList (21.1.3)).

A string literal can either be entered as the list of characters or by writing the characters between *doublequotes* ". GAP will always output strings in the latter format. However, the input via the double quote syntax enables GAP to store the string in an efficient compact internal representation. See IsStringRep (27.3.1) below for more details.

Each character, in particular those which cannot be typed directly from the keyboard, can also be typed in three digit octal notation. And for some special characters (like the newline character) there is a further possibility to type them, see section 27.2.

Example

```
gap> s1 := ['H','e','l','l','o',' ','w','o','r','l','d','.'];
"Hello world."
gap> IsString( s1 );
true
```

```
gap> s2 := "Hello world.";
"Hello world."
gap> s1 = s2;
true
gap> s3 := "";  # the empty string
""
gap> s3 = [];
true
gap> IsString( [] );
true
gap> IsString( "123" );  IsString( 123 );
true
false
gap> IsString( [ '1', '2', '3' ] );
true
gap> IsString( [ '1', '2', , '4' ] );  # strings must be dense
false
gap> IsString( [ '1', '2', 3 ] );  # strings must only contain characters
false
```

27.1.3 Strings As Lists

Note that a string is just a special case of a list. So everything that is possible for lists (see 21) is also possible for strings. Thus you can access the characters in such a string (see 21.3), test for membership (see 30.6), ask for the length, concatenate strings (see `Concatenation` (21.20.1)), form substrings etc. You can even assign to a mutable string (see 21.4). Of course unless you assign a character in such a way that the list stays dense, the resulting list will no longer be a string.

Example

```
gap> Length( s2 );
12
gap> s2[2];
'e'
gap> 'a' in s2;
false
gap> s2[2] := 'a';;  s2;
"Hallo world."
gap> s1{ [1..4] };
"Hell"
gap> Concatenation( s1{ [ 1 .. 6 ] }, s1{ [ 1 .. 4 ] } );
"Hello Hell"
```

27.1.4 Printing Strings

▷ `ViewObj(`*str*`)` (method)
▷ `PrintObj(`*str*`)` (method)

If a string is displayed by `View` (6.3.3), for example as result of an evaluation (see 6.1), or by `ViewObj` (6.3.5) and `PrintObj` (6.3.5), it is displayed with enclosing doublequotes. (But note that there is an ambiguity for the empty string which is also an empty list of arbitrary GAP ob-

jects; it is only printed like a string if it was input as empty string or converted to a string with ConvertToStringRep (27.3.2).) The output of PrintObj can be read back into GAP.

Strings behave differently from other GAP objects with respect to Print (6.3.4), PrintTo (9.7.3), or AppendTo (9.7.3). These commands *interpret* a string in the sense that they essentially send the characters of the string directly to the output stream/file. (But depending on the type of the stream and the presence of some special characters used as hints for line breaks there may be sent some additional newline (or backslash and newline) characters.

Example

```
gap> s4:= "abc\"def\nghi";;
gap> View( s4 );  Print( "\n" );
"abc\"def\nghi"
gap> ViewObj( s4 );  Print( "\n" );
"abc\"def\nghi"
gap> PrintObj( s4 );  Print( "\n" );
"abc\"def\nghi"
gap> Print( s4 );  Print( "\n" );
abc"def
ghi
gap> s := "German uses strange characters: äöüß\n";
"German uses strange characters: äöüß\n"
gap> Print(s);
German uses strange characters: äöüß
gap> PrintObj(s);  Print( "\n" );
"German uses strange characters: \303\244\303\266\303\274\303\237\n"
```

Example

```
gap> s := "\007";
"\007"
gap> Print(s); # rings bell in many terminals
```

Note that only those line breaks are printed by Print (6.3.4) that are contained in the string (\n characters, see 27.2), as is shown in the example below.

Example

```
gap> s1;
"Hello world."
gap> Print( s1 );
Hello world.gap> Print( s1, "\n" );
Hello world.
gap> Print( s1, "\nnext line\n" );
Hello world.
next line
```

27.2 Special Characters

There are a number of *special character sequences* that can be used between the singlequotes of a character literal or between the doublequotes of a string literal to specify characters. They consist of two characters. The first is a backslash \. The second may be any character. If it is an octal digit (from 0 to 7) there must be two more such digits. The meaning is given in the following list

`\n` *newline character*. This is the character that, at least on UNIX systems, separates lines in a text file. Printing of this character in a string has the effect of moving the cursor down one line and back to the beginning of the line.

`\"` *doublequote character*. Inside a string a doublequote must be escaped by the backslash, because it is otherwise interpreted as end of the string.

`\'` *singlequote character*. Inside a character a singlequote must escaped by the backslash, because it is otherwise interpreted as end of the character.

`\\` *backslash character*. Inside a string a backslash must be escaped by another backslash, because it is otherwise interpreted as first character of an escape sequence.

`\b` *backspace character*. Printing this character should have the effect of moving the cursor back one character. Whether it works or not is system dependent and should not be relied upon.

`\r` *carriage return character*. Printing this character should have the effect of moving the cursor back to the beginning of the same line. Whether this works or not is again system dependent.

`\c` *flush character*. This character is not printed. Its purpose is to flush the output queue. Usually GAP waits until it sees a `newline` before it prints a string. If you want to display a string that does not include this character use `\c`.

`\XYZ`
with `X`, `Y`, `Z` three octal digits. This is translated to the character corresponding to the number `X * 64 + Y * 8 + Z modulo 256`. This can be used to specify and store arbitrary binary data as a string in GAP.

other
For any other character the backslash is simply ignored.

Again, if the line is displayed as result of an evaluation, those escape sequences are displayed in the same way that they are input.

Only `Print` (6.3.4), `PrintTo` (9.7.3), or `AppendTo` (9.7.3) send the characters directly to the output stream.

Example

```
gap> "This is one line.\nThis is another line.\n";
"This is one line.\nThis is another line.\n"
gap> Print( last );
This is one line.
This is another line.
```

Note in particular that it is not allowed to enclose a *`newline`* inside the string. You can use the special character sequence `\n` to write strings that include *`newline`* characters. If, however, an input string is too long to fit on a single line it is possible to *continue* it over several lines. In this case the last character of each input line, except the last line must be a backslash. Both backslash and *`newline`* are thrown away by GAP while reading the string. Note that the same continuation mechanism is available for identifiers and integers, see 6.2.

27.3 Internally Represented Strings

27.3.1 IsStringRep

▷ IsStringRep(*obj*) (Representation)

IsStringRep is a special (internal) representation of dense lists of characters. Dense lists of characters can be converted into this representation using ConvertToStringRep (27.3.2). Note that calling IsString (27.1.2) does *not* change the representation.

27.3.2 ConvertToStringRep

▷ ConvertToStringRep(*obj*) (function)

If *obj* is a dense internally represented list of characters then ConvertToStringRep changes the representation to IsStringRep (27.3.1). This is useful in particular for converting the empty list [], which usually is in IsPlistRep, to IsStringRep (27.3.1). If *obj* is not a string then ConvertToStringRep signals an error.

27.3.3 CopyToStringRep

▷ CopyToStringRep(*obj*) (function)

If *obj* is a dense internally represented list of characters then CopyToStringRep copies *obj* to a new object with representation IsStringRep (27.3.1). If *obj* is not a string then CopyToStringRep signals an error.

27.3.4 IsEmptyString

▷ IsEmptyString(*str*) (function)

IsEmptyString returns true if *str* is the empty string in the representation IsStringRep (27.3.1), and false otherwise. Note that the empty list [] and the empty string "" have the same type, the recommended way to distinguish them is via IsEmptyString. For formatted printing, this distinction is sometimes necessary.

Example

```
gap> l:= [];;  IsString( l );  IsEmptyString( l );  IsEmpty( l );
true
false
true
gap> l;  ConvertToStringRep( l );  l;
[ ]
""
gap> IsEmptyString( l );  IsEmptyString( "" );  IsEmptyString( "abc" );
true
true
false
gap> ll:= [ 'a', 'b' ];  IsStringRep( ll );  ConvertToStringRep( ll );
"ab"
false
```

```
gap> ll;  IsStringRep( ll );
"ab"
true
```

27.3.5 EmptyString

▷ `EmptyString(`*`len`*`)` (function)

Returns: a string

▷ `ShrinkAllocationString(`*`str`*`)` (function)

Returns: nothing

The function `EmptyString` returns an empty string in internal representation which has enough memory allocated for *`len`* characters. This can be useful for creating and filling a string with a known number of entries.

The function `ShrinkAllocationString` gives back to GAPs memory manager the physical memory which is allocated for the string *`str`* in internal representation but not needed by its current number of characters.

These functions are intended for saving some of GAPs memory in certain situations, see the explanations and the example for the analogeous functions `EmptyPlist` (21.9.1) and `ShrinkAllocationPlist` (21.9.1) for plain lists.

27.3.6 CharsFamily

▷ `CharsFamily` (global variable)

Each character lies in the family `CharsFamily`, each nonempty string lies in the collections family of this family. Note the subtle differences between the empty list `[]` and the empty string `""` when both are printed.

27.4 Recognizing Characters

27.4.1 IsDigitChar

▷ `IsDigitChar(`*`c`*`)` (function)

checks whether the character *`c`* is a digit, i.e., occurs in the string `"0123456789"`.

27.4.2 IsLowerAlphaChar

▷ `IsLowerAlphaChar(`*`c`*`)` (function)

checks whether the character *`c`* is a lowercase alphabet letter, i.e., occurs in the string `"abcdefghijklmnopqrstuvwxyz"`.

27.4.3 IsUpperAlphaChar

▷ `IsUpperAlphaChar(`*`c`*`)` (function)

checks whether the character *c* is an uppercase alphabet letter, i.e., occurs in the string `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`.

27.4.4 IsAlphaChar

▷ `IsAlphaChar(`*c*`)` (function)

checks whether the character *c* is either a lowercase or an uppercase alphabet letter.

27.5 Comparisons of Strings

27.5.1 \= (for two strings)

▷ `\=(`*string1, string2*`)` (method)

The equality operator = returns `true` if the two strings *string1* and *string2* are equal and `false` otherwise. The inequality operator <> returns `true` if the two strings *string1* and *string2* are not equal and `false` otherwise.

Example

```
gap> "Hello world.\n" = "Hello world.\n";
true
gap> "Hello World.\n" = "Hello world.\n"; # comparison is case sensitive
false
gap> "Hello world." = "Hello world.\n";  # first string has no <newline>
false
gap> "Goodbye world.\n" = "Hello world.\n";
false
gap> [ 'a', 'b' ] = "ab";
true
```

27.5.2 \< (for two strings)

▷ `\<(`*string1, string2*`)` (method)

The ordering of strings is lexicographically according to the order implied by the underlying, system dependent, character set.

Example

```
gap> "Hello world.\n" < "Hello world.\n";  # the strings are equal
false
gap> # in ASCII capitals range before small letters:
gap> "Hello World." < "Hello world.";
true
gap> "Hello world." < "Hello world.\n";  # prefixes are always smaller
true
gap> # G comes before H, in ASCII at least:
gap> "Goodbye world.\n" < "Hello world.\n";
true
```

Strings can be compared via < with certain GAP objects that are not strings, see 4.12 for the details.

27.6 Operations to Produce or Manipulate Strings

For the possibility to print GAP objects to strings, see 10.7.

27.6.1 DisplayString

▷ `DisplayString(obj)` (operation)

Returns a string which could be used to display the object *obj* in a nice, formatted way which is easy to read (but might be difficult for machines to understand). The actual format used for this depends on the type of *obj*. Each method should include a newline character as last character. Note that no method for `DisplayString` may delegate to any of the operations `Display` (6.3.6), `ViewObj` (6.3.5) or `PrintObj` (6.3.5) to avoid circular delegations.

27.6.2 DEFAULTDISPLAYSTRING

▷ `DEFAULTDISPLAYSTRING` (global variable)

This is the default value for `DisplayString` (27.6.1).

27.6.3 ViewString

▷ `ViewString(obj)` (operation)

`ViewString` returns a string which would be displayed by `ViewObj` (6.3.5) for an object. Note that no method for `ViewString` may delegate to any of the operations `Display` (6.3.6), `ViewObj` (6.3.5), `DisplayString` (27.6.1) or `PrintObj` (6.3.5) to avoid circular delegations.

27.6.4 DEFAULTVIEWSTRING

▷ `DEFAULTVIEWSTRING` (global variable)

This is the default value for `ViewString` (27.6.3).

27.6.5 PrintString

▷ `PrintString(obj[, length])` (operation)

`PrintString` returns a representation of *obj*, which may be an object of arbitrary type, as a string. This string should approximate as closely as possible the character sequence you see if you print *obj* using `PrintObj` (6.3.5).

If *length* is given it must be an integer. The absolute value gives the minimal length of the result. If the string representation of *obj* takes less than that many characters it is filled with blanks. If *length* is positive it is filled on the left, if *length* is negative it is filled on the right.

In the two argument case, the string returned is a new mutable string (in particular not a part of any other object); it can be modified safely, and `MakeImmutable` (12.6.4) may be safely applied to it.

Example

```
gap> PrintString(123);PrintString([1,2,3]);
"123"
"[ 1, 2, 3 ]"
```

PrintString is entitled to put in additional control characters \< (ASCII 1) and \> (ASCII 2) that allow proper line breaks. See StripLineBreakCharacters (27.6.7) for a function to get rid of these control characters.

27.6.6 String

▷ String(*obj[, length]*) (attribute)

String returns a representation of *obj*, which may be an object of arbitrary type, as a string. This string should approximate as closely as possible the character sequence you see if you print *obj*.

If *length* is given it must be an integer. The absolute value gives the minimal length of the result. If the string representation of *obj* takes less than that many characters it is filled with blanks. If *length* is positive it is filled on the left, if *length* is negative it is filled on the right.

In the two argument case, the string returned is a new mutable string (in particular not a part of any other object); it can be modified safely, and MakeImmutable (12.6.4) may be safely applied to it.

Example

```
gap> String(123);String([1,2,3]);
"123"
"[ 1, 2, 3 ]"
```

String must not put in additional control characters \< (ASCII 1) and \> (ASCII 2) that allow proper line breaks.

27.6.7 StripLineBreakCharacters

▷ StripLineBreakCharacters(*st*) (function)

This function takes a string *st* as an argument and removes all control characters \< (ASCII 1) and \> (ASCII 2) which are used by PrintString (27.6.5) and PrintObj (6.3.5) to ensure proper line breaking. A new string with these characters removed is returned.

27.6.8 HexStringInt

▷ HexStringInt(*int*) (function)

returns a string which represents the integer *int* with hexa-decimal digits (using A to F as digits 10 to 15). The inverse translation can be achieved with IntHexString (27.8.1).

27.6.9 StringPP

▷ StringPP(*int*) (function)

returns a string representing the prime factor decomposition of the integer *int*.

Example

```
gap> StringPP(40320);
"2^7*3^2*5*7"
```

27.6.10 WordAlp

▷ WordAlp(*alpha, nr*) (function)

returns a string that is the *nr*-th word over the alphabet list *alpha*, w.r.t. word length and lexicographical order. The empty word is WordAlp(*alpha* , 0).

Example

```
gap> List([0..5],i->WordAlp("abc",i));
[ "", "a", "b", "c", "aa", "ab" ]
```

27.6.11 LowercaseString

▷ LowercaseString(*string*) (function)

returns a lowercase version of the string *string*, that is, a string in which each uppercase alphabet character is replaced by the corresponding lowercase character.

Example

```
gap> LowercaseString("This Is UpperCase");
"this is uppercase"
```

27.6.12 SplitString

▷ SplitString(*string, seps[, wspace]*) (operation)

This function accepts a string *string* and lists *seps* and, optionally, *wspace* of characters. Now *string* is split into substrings at each occurrence of a character in *seps* or *wspace*. The characters in *wspace* are interpreted as white space characters. Substrings of characters in *wspace* are treated as one white space character and they are ignored at the beginning and end of a string.

Both arguments *seps* and *wspace* can be single characters.

Each string in the resulting list of substring does not contain any characters in *seps* or *wspace*.

A character that occurs both in *seps* and *wspace* is treated as a white space character.

A separator at the end of a string is interpreted as a terminator; in this case, the separator does not produce a trailing empty string. Also see Chomp (27.6.18).

Example

```
gap> SplitString( "substr1:substr2::substr4", ":" );
[ "substr1", "substr2", "", "substr4" ]
gap> SplitString( "a;b;c;d;", ";" );
[ "a", "b", "c", "d" ]
gap> SplitString( "/home//user//dir/", "", "/" );
[ "home", "user", "dir" ]
```

27.6.13 ReplacedString

▷ ReplacedString(*string, old, new*) (function)

replaces occurrences of the string *old* in *string* by *new*, starting from the left and always replacing the first occurrence. To avoid infinite recursion, characters which have been replaced already, are not subject to renewed replacement.

Example

```
gap> ReplacedString("abacab","a","zl");
"zlbzlczlb"
gap> ReplacedString("ababa", "aba","c");
"cba"
gap> ReplacedString("abacab","a","ba");
"babbacbab"
```

27.6.14 NormalizeWhitespace

▷ NormalizeWhitespace(*string*) (function)

This function changes the string *string* in place. The characters (space), \n, \r and \t are considered as *white space*. Leading and trailing white space characters in *string* are removed. Sequences of white space characters between other characters are replaced by a single space character.

See NormalizedWhitespace (27.6.15) for a non-destructive version.

Example

```
gap> s := "   x y \n\n\t\r  z\n   \n";
"   x y \n\n\t\r  z\n   \n"
gap> NormalizeWhitespace(s);
gap> s;
"x y z"
```

27.6.15 NormalizedWhitespace

▷ NormalizedWhitespace(*str*) (function)

This function returns a copy of string *str* to which NormalizeWhitespace (27.6.14) was applied.

27.6.16 RemoveCharacters

▷ RemoveCharacters(*string, chars*) (function)

Both arguments must be strings. This function efficiently removes all characters given in *chars* from *string*.

Example

```
gap> s := "ab c\ndef\n\ng    h i .\n";
"ab c\ndef\n\ng    h i .\n"
gap> RemoveCharacters(s, " \n\t\r"); # remove all whitespace characters
gap> s;
"abcdefghi."
```

27.6.17 JoinStringsWithSeparator

▷ JoinStringsWithSeparator(*list[, sep]*) (function)

joins *list* (a list of strings) after interpolating *sep* (or "," if the second argument is omitted) between each adjacent pair of strings; *sep* should be a string.

Example

```
gap> list := List([1..10], String);
[ "1", "2", "3", "4", "5", "6", "7", "8", "9", "10" ]
gap> JoinStringsWithSeparator(list);
"1,2,3,4,5,6,7,8,9,10"
gap> JoinStringsWithSeparator(["The", "quick", "brown", "fox"], " ");
"The quick brown fox"
gap> new:= JoinStringsWithSeparator(["a", "b", "c", "d"], ",\n    ");
"a,\n    b,\n    c,\n    d"
gap> Print("    ", new, "\n");
    a,
    b,
    c,
    d
```

27.6.18 Chomp

▷ Chomp(*str*) (function)

Like the similarly named Perl function, Chomp removes a trailing newline character (or carriage-return line-feed couplet) from a string argument *str* if present and returns the result. If *str* is not a string or does not have such trailing character(s) it is returned unchanged. This latter property means that Chomp is safe to use in cases where one is manipulating the result of another function which might sometimes return fail.

Example

```
gap> Chomp("The quick brown fox jumps over the lazy dog.\n");
"The quick brown fox jumps over the lazy dog."
gap> Chomp("The quick brown fox jumps over the lazy dog.\r\n");
"The quick brown fox jumps over the lazy dog."
gap> Chomp("The quick brown fox jumps over the lazy dog.");
"The quick brown fox jumps over the lazy dog."
gap> Chomp(fail);
fail
gap> Chomp(32);
32
```

Note: Chomp only removes a trailing newline character from *str*. If your string contains several newline characters and you really want to split *str* into lines at the newline characters (and remove those newline characters) then you should use SplitString (27.6.12), e.g.

Example

```
gap> str := "The quick brown fox\njumps over the lazy dog.\n";
"The quick brown fox\njumps over the lazy dog.\n"
gap> SplitString(str, "", "\n");
[ "The quick brown fox", "jumps over the lazy dog." ]
```

```
gap> Chomp(str);
"The quick brown fox\njumps over the lazy dog."
```

The following two functions convert basic strings to lists of numbers and vice versa. They are useful for examples of text encryption.

27.6.19 NumbersString

▷ NumbersString(*s*, *m*[, *table*]) (function)

NumbersString takes a string message *s* and returns a list of integers, each not exceeding the integer *m* that encode the message using the scheme $A = 11$, $B = 12$ and so on (and converting lower case to upper case). If a list of characters is given in *table*, it is used instead for encoding).

Example

```
gap> l:=NumbersString("Twas brillig and the slithy toves",1000000);
[ 303311, 291012, 281922, 221917, 101124, 141030, 181510, 292219,
  301835, 103025, 321529 ]
```

27.6.20 StringNumbers

▷ StringNumbers(*l*, *m*[, *table*]) (function)

StringNumbers takes a list *l* of integers that was encoded using NumbersString (27.6.19) and the size integer *m*, and returns a message string, using the scheme $A = 11$, $B = 12$ and so on. If a list of characters is given in *table*, it is used instead for decoding).

Example

```
gap> StringNumbers(l,1000000);
"TWAS BRILLIG AND THE SLITHY TOVES"
```

27.7 Character Conversion

The following functions convert characters in their internal integer values and vice versa. Note that the number corresponding to a particular character might depend on the system used. While most systems use an extension of ASCII, in particular character values outside the range [32 .. 126] might differ between architectures.

27.7.1 IntChar

▷ IntChar(*char*) (function)

returns an integer value in the range [0 .. 255] that corresponds to *char*.

27.7.2 CharInt

▷ CharInt(*int*) (function)

returns a character that corresponds to the integer value *int*, which must be in the range [0 .. 255].

Example

```
gap> c:=CharInt(65);
'A'
gap> IntChar(c);
65
```

27.7.3 SIntChar

▷ SIntChar(*char*) (function)

returns a signed integer value in the range [-128 .. 127] that corresponds to *char*.

27.7.4 CharSInt

▷ CharSInt(*int*) (function)

returns a character which corresponds to the signed integer value *int*, which must be in the range [-128 .. 127].

The signed and unsigned integer functions behave the same for values in the range [0 .. 127].

Example

```
gap> SIntChar(c);
65
gap> c:=CharSInt(-20);;
gap> SIntChar(c);
-20
gap> IntChar(c);
236
gap> SIntChar(CharInt(255));
-1
```

27.8 Operations to Evaluate Strings

27.8.1 Int (for strings)

▷ Int(*str*) (attribute)
▷ Rat(*str*) (attribute)
▷ IntHexString(*str*) (function)

return either an integer (Int and IntHexString), or a rational (Rat) as represented by the string *str*. Int returns fail if non-digit characters occur in *str*. For Rat, the argument string may start with the sign character -, followed by either a sequence of digits or by two sequences of digits that are separated by one of the characters / or ., where the latter stands for a decimal dot. (The methods only evaluate numbers but do *not* perform arithmetic!)

IntHexString evaluates an integer written with hexa-decimal digits. Here the letters a-f or A-F are used as *digits* 10 to 15. An error occurs when a wrong character is found in the string. This function can be used (together with HexStringInt (27.6.8)) for efficiently storing and reading

large integers from respectively into GAP. Note that the translation between integers and their hexadecimal representation costs linear computation time in terms of the number of digits, while translation from and into decimal representation needs substantial computations. If *str* is not in compact string representation then ConvertToStringRep (27.3.2) is applied to it as side effect.

Example

```
gap> Int("12345")+1;
12346
gap> Int("123/45");
fail
gap> Int("1+2");
fail
gap> Int("-12");
-12
gap> Rat("123/45");
41/15
gap> Rat( "123.45" );
2469/20
gap> IntHexString("-abcdef0123456789");
-12379813738877118345
gap> HexStringInt(last);
"-ABCDEF0123456789"
```

27.8.2 Ordinal

▷ Ordinal(*n*) (function)

returns the ordinal of the integer *n* as a string.

Example

```
gap> Ordinal(2);  Ordinal(21);  Ordinal(33);  Ordinal(-33);
"2nd"
"21st"
"33rd"
"-33rd"
```

27.8.3 EvalString

▷ EvalString(*expr*) (function)

passes the string *expr* through an input text stream so that GAP interprets it, and returns the result.

Example

```
gap> a:=10;
10
gap> EvalString("a^2");
100
```

EvalString is intended for *single* expressions. A sequence of commands may be interpreted by using the functions InputTextString (10.7.1) and ReadAsFunction (10.3.2) together; see 10.3 for an example.

If EvalString is used inside a function, then it doesn't know about the local variables and the arguments of the function. A possible workaround is to define global variables in advance, and then to assign the values of the local variables to the global ones, like in the example below.

Example

```
gap> global_a := 0;;
gap> global_b := 0;;
gap> example := function ( local_a )
>     local  local_b;
>     local_b := 5;
>     global_a := local_a;
>     global_b := local_b;
>     return EvalString( "global_a * global_b" );
> end;;
gap> example( 2 );
10
```

27.8.4 CrcString

▷ CrcString(*str*) (function)

Returns: an integer

This function computes a cyclic redundancy check number from a string *str*. See also CrcFile (9.7.7).

Example

```
gap> CrcString("GAP example string");
-50451670
```

27.9 Calendar Arithmetic

All calendar functions use the Gregorian calendar.

27.9.1 DaysInYear

▷ DaysInYear(*year*) (function)

returns the number of days in the year *year*.

27.9.2 DaysInMonth

▷ DaysInMonth(*month, year*) (function)

returns the number of days in month number *month* of *year*, and fail if month is not in the valid range.

Example

```
gap> DaysInYear(1998);
365
gap> DaysInMonth(3,1998);
31
```

27.9.3 DMYDay

▷ DMYDay(*day*) (function)

converts a number of days, starting 1-Jan-1970, to a list [day, month, year] in Gregorian calendar counting.

27.9.4 DayDMY

▷ DayDMY(*dmy*) (function)

returns the number of days from 01-Jan-1970 to the day given by *dmy*, which must be a list of the form [day, month, year] in Gregorian calendar counting. The result is fail on input outside valid ranges.

Note that this makes not much sense for early dates like: before 1582 (no Gregorian calendar at all), or before 1753 in many English speaking countries or before 1917 in Russia.

27.9.5 WeekDay

▷ WeekDay(*date*) (function)

returns the weekday of a day given by *date*, which can be a number of days since 1-Jan-1970 or a list [day, month, year].

27.9.6 StringDate

▷ StringDate(*date*) (function)

converts *date* to a readable string. *date* can be a number of days since 1-Jan-1970 or a list [day, month, year].

Example

```
gap> DayDMY([1,1,1970]);DayDMY([2,1,1970]);
0
1
gap> DMYDay(12345);
[ 20, 10, 2003 ]
gap> WeekDay([11,3,1998]);
"Wed"
gap> StringDate([11,3,1998]);
"11-Mar-1998"
```

27.9.7 HMSMSec

▷ HMSMSec(*msec*) (function)

converts a number *msec* of milliseconds into a list [hour, min, sec, milli].

27.9.8 SecHMSM

▷ SecHMSM(*hmsm*) (function)

is the reverse of HMSMSec (27.9.7).

27.9.9 StringTime

▷ StringTime(*time*) (function)

converts *time* (given as a number of milliseconds or a list [hour, min, sec, milli]) to a readable string.

Example

```
gap> HMSMSec(Factorial(10));
[ 1, 0, 28, 800 ]
gap> SecHMSM([1,10,5,13]);
4205013
gap> StringTime([1,10,5,13]);
" 1:10:05.013"
```

27.9.10 SecondsDMYhms

▷ SecondsDMYhms(*DMYhms*) (function)

returns the number of seconds from 01-Jan-1970, 00:00:00, to the time given by *DMYhms*, which must be a list of the form [day, month, year, hour, minute, second]. The remarks on the Gregorian calendar in the section on DayDMY (27.9.4) apply here as well. The last three arguments must lie in the appropriate ranges.

27.9.11 DMYhmsSeconds

▷ DMYhmsSeconds(*secs*) (function)

This is the inverse function to SecondsDMYhms (27.9.10).

Example

```
gap> SecondsDMYhms([ 9, 9, 2001, 1, 46, 40 ]);
1000000000
gap> DMYhmsSeconds(-1000000000);
[ 24, 4, 1938, 22, 13, 20 ]
```

27.10 Obtaining LaTeX Representations of Objects

For the purpose of generating LaTeX source code with GAP it is recommended to add new functions which will print the LaTeX source or return LaTeX strings for further processing.

An alternative approach could be based on methods for the default LaTeX representation for each appropriate type of objects. However, there is no clear notion of a default LaTeX code for any non-trivial mathematical object; moreover, different output may be required in different contexts.

While customisation of such an operation may require changes in a variety of methods that may be distributed all over the library, the user will have a clean overview of the whole process of LaTeX code generation if it is contained in a single function. Furthermore, there may be kinds of objects which are not detected by the method selection, or there may be a need in additional parameters specifying requirements for the output.

This is why having a special purpose function for each particular case is more suitable. GAP provides several functions that produce LaTeX strings for those situations where this is nontrivial and reasonable. A useful example is `LaTeXStringDecompositionMatrix` (71.11.5) from the GAP library, others can be found entering `?LaTeX` at the GAP prompt. Package authors are encouraged to add an index entry `LaTeX` to the documentation of all LaTeX string producing functions. This way, entering `?LaTeX` will give an overview of all documented functionality in this direction.

Chapter 28

Dictionaries and General Hash Tables

People and computers spend a large amount of time with searching. Dictionaries are an abstract data structure which facilitates searching for objects. Depending on the kind of objects the implementation will use a variety of possible internal storage methods that will aim to provide the fastest possible access to objects. These internal methods include

Hash Tables
for objects for which a hash function has been defined.

Direct Indexing
if the domain is small and cheaply enumerable

Sorted Lists
if a total order can be computed easily

Plain lists
for objects for which nothing but an equality test is available.

28.1 Using Dictionaries

The standard way to use dictionaries is to first create a dictionary (using `NewDictionary` (28.2.1), and then to store objects (and associated information) in it and look them up.

For the creation of objects the user has to make a few choices: Is the dictionary only to be used to check whether objects are known already, or whether associated information is to be stored with the objects. This second case is called a *lookup dictionary* and is selected by the second parameter of `NewDictionary` (28.2.1).

The second choice is to indicate which kind of objects are to be stored. This choice will decide the internal storage used. This kind of objects is specified by the first parameter to `NewDictionary` (28.2.1), which is a "sample" object.

In some cases however such a sample object is not specific enough. For example when storing vectors over a finite field, it would not be clear whether all vectors will be over a prime field or over a field extension. Such an issue can be resolved by indicating in an (optional) third parameter to `NewDictionary` (28.2.1) a *domain* which has to be a collection that will contain all objects to be used with this dictionary. (Such a domain may also be used internally to decide that direct indexing can be used).

The reason for this choice of giving two parameters is that in some cases no suitable collection of objects has been defined in GAP - for example for permutations there is no object representing the symmetric group on infinitely many points.

Once a dictionary has been created, it is possible to use `RepresentationsOfObject` (13.4.1) to check which representation is used by GAP.

In the following example, we create a dictionary to store permutations with associated information.

Example

```
gap> d:=NewDictionary((1,2,3),true);;
gap> AddDictionary(d,(1,2),1);
gap> AddDictionary(d,(5,6),9);
gap> AddDictionary(d,(4,7),2);
gap> LookupDictionary(d,(5,6));
9
gap> LookupDictionary(d,(5,7));
fail
```

A typical example of this use would be in an orbit algorithm. The dictionary would be used to store the elements known in the orbit together with their respective orbit positions.

We observe that this dictionary is stored internally by a sorted list. On the other hand, if we have an explicit, sorted element list, direct indexing is to be used.

Example

```
gap> RepresentationsOfObject(d);
[ "IsComponentObjectRep", "IsDictionaryDefaultRep",
  "IsListDictionary", "IsListLookupDictionary", "IsSortDictionary",
  "IsSortLookupDictionary" ]
gap> d:=NewDictionary((1,2,3),true,Elements(SymmetricGroup(5)));;
gap> RepresentationsOfObject(d);
[ "IsComponentObjectRep", "IsDictionaryDefaultRep",
  "IsPositionDictionary", "IsPositionDictionary" ]
```

(Just indicating `SymmetricGroup(5)` as a third parameter would still keep the first storage method, as indexing would be too expensive if no explicit element list is known.)

The same effect happens in the following example, in which we work with vectors: Indicating only a vector only enables sorted index, as it cannot be known whether all vectors will be defined over the prime field. On the other hand, providing the vector space (and thus limiting the domain) enables the use of hashing (which will be faster).

Example

```
gap> v:=GF(2)^7;;
gap> d:=NewDictionary(Zero(v),true);;
gap> RepresentationsOfObject(d);
[ "IsComponentObjectRep", "IsDictionaryDefaultRep",
  "IsListDictionary", "IsListLookupDictionary", "IsSortDictionary",
  "IsSortLookupDictionary" ]
gap> d:=NewDictionary(Zero(v),true,v);;
gap> RepresentationsOfObject(d);
[ "IsComponentObjectRep", "IsDictionaryDefaultRep",
  "IsPositionDictionary", "IsPositionDictionary" ]
```

28.2 Dictionaries

This section contains the formal declarations for dictionaries. For information on how to use them, please refer to the previous section 28.1. There are several ways how dictionaries are implemented: As lists, as sorted lists, as hash tables or via binary lists. A user however will just have to call `NewDictionary` (28.2.1) and obtain a "suitable" dictionary for the kind of objects she wants to create. It is possible however to create hash tables (see 28.4) and dictionaries using binary lists (see `DictionaryByPosition` (28.3.1)).

The use of two objects, *obj* and *objcoll* to parametrize the objects a dictionary is able to store might look confusing. However there are situations where either of them might be needed:

The first situation is that of objects, for which no formal "collection object" has been defined. A typical example here might be subspaces of a vector space. GAP does not formally define a "Grassmannian" or anything else to represent the multitude of all subspaces. So it is only possible to give the dictionary a "sample object".

The other situation is that of an object which might represent quite varied domains. The permutation $(1, 10^6)$ might be the nontrivial element of a cyclic group of order 2, it might be a representative of S_{10^6}. In the first situation the best approach might be just to have two entries for the two possible objects, in the second situation a much more elaborate approach might be needed.

An algorithm that creates a dictionary will usually know a priori, from what domain all the objects will be, giving this domain permits to use a more efficient dictionary.

This is particularly true for vectors. From a single vector one cannot decide whether a calculation will take place over the smallest field containing all its entries or over a larger field.

28.2.1 NewDictionary

▷ `NewDictionary(`*obj, look[, objcoll]*`)` (function)

creates a new dictionary for objects such as *obj*. If *objcoll* is given the dictionary will be for objects only from this collection, knowing this can improve the performance. If *objcoll* is given, *obj* may be replaced by `false`, i.e. no sample object is needed.

The function tries to find the right kind of dictionary for the basic dictionary functions to be quick. If *look* is `true`, the dictionary will be a lookup dictionary, otherwise it is an ordinary dictionary.

28.3 Dictionaries via Binary Lists

As there are situations where the approach via binary lists is explicitly desired, such dictionaries can be created deliberately.

28.3.1 DictionaryByPosition

▷ `DictionaryByPosition(`*list, lookup*`)` (function)

creates a new (lookup) dictionary which uses `PositionCanonical` (21.16.3) in *list* for indexing. The dictionary will have an entry *dict*`!.blist` which is a bit list corresponding to *list* indicating the known values. If *look* is `true`, the dictionary will be a lookup dictionary, otherwise it is an ordinary dictionary.

28.3.2 IsDictionary

▷ `IsDictionary(obj)` (Category)

A dictionary is a growable collection of objects that permits to add objects (with associated values) and to check whether an object is already known.

28.3.3 IsLookupDictionary

▷ `IsLookupDictionary(obj)` (Category)

A *lookup dictionary* is a dictionary, which permits not only to check whether an object is contained, but also to retrieve associated values, using the operation `LookupDictionary` (28.3.6).

28.3.4 AddDictionary

▷ `AddDictionary(dict, key[, val])` (operation)

adds *key* to the dictionary *dict*, storing the associated value *val* in case *dict* is a lookup dictionary. If *key* is not an object of the kind for which the dictionary was specified, or if *key* is known already to *dict*, the results are unpredictable.

28.3.5 KnowsDictionary

▷ `KnowsDictionary(dict, key)` (operation)

checks, whether *key* is known to the dictionary *dict*, and returns `true` or `false` accordingly. *key must* be an object of the kind for which the dictionary was specified, otherwise the results are unpredictable.

28.3.6 LookupDictionary

▷ `LookupDictionary(dict, key)` (operation)

looks up *key* in the lookup dictionary *dict* and returns the associated value. If *key* is not known to the dictionary, `fail` is returned.

28.4 General Hash Tables

These sections describe some particularities for hash tables. These are intended mainly for extending the implementation - programs requiring hash functionality ought to use the dictionary interface described above.

We hash by keys and also store a value. Keys cannot be removed from the table, but the corresponding value can be changed. Fast access to last hash index allows you to efficiently store more than one array of values –this facility should be used with care.

This code works for any kind of object, provided you have a `DenseIntKey` (28.5.1) method to convert the key into a positive integer. This method should ideally be implemented efficiently in the core.

Note that, for efficiency, it is currently impossible to create a hash table with non-positive integers.

28.5 Hash keys

The crucial step of hashing is to transform key objects into integers such that equal objects produce the same integer.

The actual function used will vary very much on the type of objects. However GAP provides already key functions for some commonly encountered objects.

28.5.1 DenseIntKey

▷ `DenseIntKey(`*`objcoll, obj`*`)` (operation)

returns a function that can be used as hash key function for objects such as *`obj`* in the collection *`objcoll`*. Typically, *`objcoll`* will be a large domain. If the domain is not available, it can be given as `false` in which case the hash key function will be determined only based on *`obj`*. (For a further discussion of these two arguments see `NewDictionary` (28.2.1)).

The function returned by `DenseIntKey` is guaranteed to give different values for different objects. If no suitable hash key function has been predefined, `fail` is returned.

28.5.2 SparseIntKey

▷ `SparseIntKey(`*`objcoll, obj`*`)` (operation)

returns a function that can be used as hash key function for objects such as *`obj`* in the collection *`objcoll`*. In contrast to `DenseIntKey` (28.5.1), the function returned may return the same key value for different objects. If no suitable hash key function has been predefined, `fail` is returned.

28.6 Dense hash tables

Dense hash tables are used for hashing dense sets without collisions, in particular integers. Keys are stored as an unordered list and values as an array with holes. The position of a value is given by the function returned by `DenseIntKey` (28.5.1), and so `KeyIntDense` must be one-to-one.

28.6.1 DenseHashTable

▷ `DenseHashTable()` (function)

Construct an empty dense hash table. This is the only correct way to construct such a table.

28.7 Sparse hash tables

Sparse hash tables are used for hashing sparse sets. Stores keys as an array with fail denoting an empty position, stores values as an array with holes. Uses the result of calling `SparseIntKey` (28.5.2)) of the key. `DefaultHashLength` is the default starting hash table length; the table is doubled when it becomes half full.

In sparse hash tables, the integer obtained from the hash key is then transformed to an index position by taking it modulo the length of the hash array.

28.7.1 SparseHashTable

▷ `SparseHashTable(`*`[intkeyfun]`*`)` (function)

Construct an empty sparse hash table. This is the only correct way to construct such a table. If the argument *`intkeyfun`* is given, this function will be used to obtain numbers for the keys passed to it.

28.7.2 DoubleHashArraySize

▷ `DoubleHashArraySize(`*`hash`*`)` (function)

Double the size of the hash array and rehash all the entries. This will also happen automatically when the hash array is half full.

Chapter 29

Records

Records are next to lists the most important way to collect objects together. A record is a collection of *components*. Each component has a unique *name*, which is an identifier that distinguishes this component, and a *value*, which is an object of arbitrary type. We often abbreviate *value of a component* to *element*. We also say that a record *contains* its elements. You can access and change the elements of a record using its name.

Record literals are written by writing down the components in order between "`rec(`" and "`)`", and separating them by commas "`,`". Each component consists of the name, the assignment operator "`:=`", and the value. The *empty record*, i.e., the record with no components, is written as `rec()`.

Example

```
gap> rec( a := 1, b := "2" ); # a record with two components
rec( a := 1, b := "2" )
gap> rec( a := 1, b := rec( c := 2 ) ); # record may contain records
rec( a := 1, b := rec( c := 2 ) )
```

We may use the `Display` (6.3.6) function to illustrate the hierarchy of the record components.

Example

```
gap> Display( last );
rec(
  a := 1,
  b := rec(
      c := 2 ) )
```

Records usually contain elements of various types, i.e., they are usually not homogeneous like lists.

29.1 IsRecord and RecNames

29.1.1 IsRecord

▷ `IsRecord(`*obj*`)` (Category)

▷ `IsRecordCollection(`*obj*`)` (Category)

▷ `IsRecordCollColl(`*obj*`)` (Category)

Example

```
gap> IsRecord( rec( a := 1, b := 2 ) );
true
gap> IsRecord( IsRecord );
false
```

29.1.2 RecNames

▷ RecNames(*record*) (attribute)

returns a list of strings corresponding to the names of the record components of the record *record*.

Example

```
gap> r := rec( a := 1, b := 2 );;
gap> Set(RecNames( r )); # 'Set' because ordering depends on GAP session
[ "a", "b" ]
```

Note that you cannot use the string result in the ordinary way to access or change a record component. You can use the *record*.(*name*) construct for that, see 29.2 and 29.3.

29.2 Accessing Record Elements

r.*name*

The above construct evaluates to the value of the record component with the name *name* in the record *r*. Note that the *name* is not evaluated, i.e. it is taken literal.

Example

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a;
1
gap> r.b;
2
```

r.(*name*)

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then evaluates to the element of the record *r* whose name is, as a string, equal to *name*.

Example

```
gap> old := rec( a := 1, b := 2 );;
gap> new := rec();
rec(  )
gap> for i in RecNames( old ) do
>  new.(i) := old.(i);
> od;
gap> Display( new );
rec(
  a := 1,
  b := 2 )
```

29.3 Record Assignment

r.name := obj

The record assignment assigns the object *obj*, which may be an object of arbitrary type, to the record component with the name *name*, which must be an identifier, of the record *r*. That means that accessing the element with name *name* of the record *r* will return *obj* after this assignment. If the record *r* has no component with the name *name*, the record is automatically extended to make room for the new component.

Example

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a := 10;;
gap> Display( r );
rec(
  a := 10,
  b := 2 )
gap> r.c := 3;;
gap> Display( r );
rec(
  a := 10,
  b := 2,
  c := 3 )
```

Note that assigning to a record changes the record.

The function IsBound (29.6.1) can be used to test if a record has a component with a certain name, the function Unbind (29.6.2) can be used to remove a component with a certain name again.

Example

```
gap> IsBound(r.a);
true
gap> IsBound(r.d);
false
gap> Unbind(r.b);
gap> Display( r );
rec(
  a := 10,
  c := 3 )
```

r.(name) := obj

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then assigns *obj* to the record component of the record *r* whose name is, as a string, equal to *name*.

29.4 Identical Records

With the record assignment (see 29.3) it is possible to change a record. This section describes the semantic consequences of this fact which are essentially the same as for lists (see 21.6).

Example

```
r := rec( a := 1 );
r := rec( a := 1, b := 2 );
```

The second assignment does not change the first record, instead it assigns a new record to the variable `r`. On the other hand, in the following example the record is changed by the second assignment.

Example

```
r := rec( a := 1 );
r.b := 2;
```

To understand the difference first think of a variable as a name for an object. The important point is that a record can have several names at the same time. An assignment *var* `:=` *r* means in this interpretation that *var* is a name for the object *r*. At the end of the following example `r2` still has the value `rec( a := 1 )` as this record has not been changed and nothing else has been assigned to `r2`.

Example

```
r1 := rec( a := 1 );
r2 := r1;
r1 := rec( a := 1, b := 2 );
```

But after the following example the record for which `r2` is a name has been changed and thus the value of `r2` is now `rec( a := 1, b := 2 )`.

Example

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

We shall say that two records are *identical* if changing one of them by a record assignment also changes the other one. This is slightly incorrect, because if *two* records are identical, there are actually only two names for *one* record. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical records must be equal, because there is only one records with two different names. Thus identity is an equivalence relation that is a refinement of equality.

Let us now consider under which circumstances two records are identical.

If you enter a record literal then the record denoted by this literal is a new record that is not identical to any other record. Thus in the following example `r1` and `r2` are not identical, though they are equal of course.

Example

```
r1 := rec( a := 1 );
r2 := rec( a := 1 );
```

Also in the following example, no records in the list `l` are identical.

Example

```
l := [];
for i in [1..10] do
  l[i] := rec( a := 1 );
od;
```

If you assign a record to a variable no new record is created. Thus the record value of the variable on the left hand side and the record on the right hand side of the assignment are identical. So in the following example `r1` and `r2` are identical records.

Example

```
r1 := rec( a := 1 );
r2 := r1;
```

If you pass a record as argument, the old record and the argument of the function are identical. Also if you return a record from a function, the old record and the value of the function call are identical. So in the following example r1 and r2 are identical records.

Example

```
r1 := rec( a := 1 );
f := function ( r ) return r; end;
r2 := f( r1 );
```

The functions StructuralCopy (12.7.2) and ShallowCopy (12.7.1) accept a record and return a new record that is equal to the old record but that is *not* identical to the old record. The difference between StructuralCopy (12.7.2) and ShallowCopy (12.7.1) is that in the case of ShallowCopy (12.7.1) the corresponding components of the new and the old records will be identical, whereas in the case of StructuralCopy (12.7.2) they will only be equal. So in the following example r1 and r2 are not identical records.

Example

```
r1 := rec( a := 1 );
r2 := ShallowCopy( r1 );
```

If you change a record it keeps its identity. Thus if two records are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two records that are not identical will never become identical if you change one of them. So in the following example both r1 and r2 are changed, and are still identical.

Example

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

29.5 Comparisons of Records

rec1 = *rec2*

rec1 <> *rec2*

Two records are considered equal, if for each component of one record the other record has a component of the same name with an equal value and vice versa.

Example

```
gap> rec( a := 1, b := 2 ) = rec( b := 2, a := 1 );
true
gap> rec( a := 1, b := 2 ) = rec( a := 2, b := 1 );
false
gap> rec( a := 1 ) = rec( a := 1, b := 2 );
false
gap> rec( a := 1 ) = 1;
false
```

rec1 < *rec2*

rec1 <= *rec2*

To compare records we imagine that the components of both records are sorted according to their names (the sorting depends on the GAP session, more precisely the order in which component names

were first used). Then the records are compared lexicographically with unbound elements considered smaller than anything else. Precisely one record *rec1* is considered less than another record *rec2* if *rec2* has a component with name *name2* and either *rec1* has no component with this name or *rec1*`.`*name2* `<` *rec2*`.`*name2* and for each component of *rec1* with name *name1* `<` *name2* *rec2* has a component with this name and *rec1*`.`*name1* `=` *rec2*`.`*name1*.

Example

```
gap> rec( axy := 1, bxy := 2 ) < rec( bxy := 2, axy := 1 ); # are equal
false
gap> rec( axy := 1 ) < rec( axy := 1, bxy := 2 ); # unbound is < 2
true
gap> # in new session the .axy components are compared first
gap> rec( axy := 1, bxy := 2 ) < rec( axy := 2, bxy := 0 ); # 1 < 2
true
gap> rec( axy := 1 ) < rec( axy := 0, bxy := 2 ); # 0 < 1
false
gap> rec( bxy := 1 ) < rec( bxy := 0, axy := 2 ); # unbound is < 2
true
```

29.6 IsBound and Unbind for Records

29.6.1 IsBound (for a record component)

▷ `IsBound(`*r*`.`*name*`)` (operation)

`IsBound` returns `true` if the record *r* has a component with the name *name* (which must be an identifier) and `false` otherwise. *r* must evaluate to a record, otherwise an error is signalled.

Example

```
gap> r := rec( a := 1, b := 2 );;
gap> IsBound( r.a );
true
gap> IsBound( r.c );
false
```

29.6.2 Unbind (unbind a record component)

▷ `Unbind(`*r*`.`*name*`)` (operation)

`Unbind` deletes the component with the name *name* in the record *r*. That is, after execution of `Unbind`, *r* no longer has a record component with this name. Note that it is not an error to unbind a nonexisting record component. *r* must evaluate to a record, otherwise an error is signalled.

Example

```
gap> r := rec( a := 1, b := 2 );;
gap> Unbind( r.a ); r;
rec( b := 2 )
gap> Unbind( r.c ); r;
rec( b := 2 )
```

Note that `IsBound` (29.6.1) and `Unbind` are special in that they do not evaluate their argument, otherwise `IsBound` (29.6.1) would always signal an error when it is supposed to return `false` and there would be no way to tell `Unbind` which component to remove.

29.7 Record Access Operations

Internally, record accesses are done using the operations listed in this section. For the records implemented in the kernel, kernel methods are provided for all these operations but otherwise it is possible to install methods for these operations for any object. This permits objects to simulate record behavior.

To save memory, records do not store a list of all component names, but only numbers identifying the components. There numbers are called *RNams*. GAP keeps a global list of all RNams that are used and provides functions to translate RNams to strings that give the component names and vice versa.

29.7.1 NameRNam

▷ `NameRNam(nr)` (function)

returns a string representing the component name corresponding to the RNam *nr*.

29.7.2 RNamObj (for a string)

▷ `RNamObj(str)` (function)
▷ `RNamObj(int)` (function)

returns a number (the RNam) corresponding to the string *str*. It is also possible to pass a positive integer *int* in which case the decimal expansion of *int* is used as a string.

Example

```
gap> NameRNam(798);
"BravaisSupergroups"
gap> RNamObj("blubberflutsch");
2075
gap> NameRNam(last);
"blubberflutsch"
```

The correspondence between strings and RNams is not predetermined ab initio, but RNams are assigned to component names dynamically on a "first come, first serve" basis. Therefore, depending on the version of the library you are using and on the assignments done so far, the *same* component name may be represented by *different* RNams in different GAP sessions.

29.7.3 \.

▷ `\.(obj, rnam)` (operation)
▷ `IsBound\.(obj, rnam)` (operation)
▷ `\.\:\=(obj, rnam, val)` (operation)
▷ `Unbind\.(obj, rnam)` (operation)

These operations are called for record accesses to arbitrary objects. If applicable methods are installed, they are called when the object is accessed as a record.

For records, the operations implement component access, test for element boundness, component assignment and removal of the component represented by the RNam *rnam*.

The component identifier *rnam* is always required to be in `IsPosInt` (14.2.2).

Chapter 30

Collections

A *collection* in GAP consists of elements in the same family (see 13.1). The most important kinds of collections are *homogeneous lists* (see 21) and *domains* (see 12.4). Note that a list is never a domain, and a domain is never a list. A list is a collection if and only if it is nonempty and homogeneous.

Basic operations for collections are `Size` (30.4.6) and `Enumerator` (30.3.2); for *finite* collections, `Enumerator` (30.3.2) admits to delegate the other operations for collections (see 30.4 and 30.5) to functions for lists (see 21). Obviously, special methods depending on the arguments are needed for the computation of e.g. the intersection of two *infinite* domains.

30.1 IsCollection (Filter)

30.1.1 IsCollection

▷ `IsCollection(`*obj*`)` (Category)

tests whether an object is a collection.

Some of the functions for lists and collections are described in the chapter about lists, mainly in Section 21.20. In the current chapter, we describe those functions for which the "collection aspect" seems to be more important than the "list aspect".

30.2 Collection Families

30.2.1 CollectionsFamily

▷ `CollectionsFamily(`*Fam*`)` (attribute)

For a family *Fam*, `CollectionsFamily` returns the family of all collections over *Fam*, that is, of all dense lists and domains that consist of objects in *Fam*.

The `NewFamily` (79.7.1) call in the standard method of `CollectionsFamily` is executed with second argument `IsCollection` (30.1.1), since every object in the collections family must be a collection, and with third argument the collections categories of the involved categories in the implied filter of *Fam*.

Note that families (see 13.1) are used to describe relations between objects. Important such relations are that between an element e and each collection of elements that lie in the same family as e,

and that between two collections whose elements lie in the same family. Therefore, all collections of elements in the family *Fam* form the new family CollectionsFamily(*Fam*).

30.2.2 IsCollectionFamily

▷ IsCollectionFamily(*obj*) (Category)

is true if *Fam* is a family of collections, and false otherwise.

30.2.3 ElementsFamily

▷ ElementsFamily(*Fam*) (attribute)

If *Fam* is a collections family (see IsCollectionFamily (30.2.2)) then ElementsFamily returns the family from which *Fam* was created by CollectionsFamily (30.2.1). The way a collections family is created, it always has its elements family stored. If *Fam* is not a collections family then an error is signalled.

Example

```
gap> fam:= FamilyObj( (1,2) );;
gap> collfam:= CollectionsFamily( fam );;
gap> fam = collfam;  fam = ElementsFamily( collfam );
false
true
gap> collfam = FamilyObj( [ (1,2,3) ] );
true
gap> collfam = FamilyObj( Group( () ) );
true
gap> collfam = CollectionsFamily( collfam );
false
```

30.2.4 CategoryCollections

▷ CategoryCollections(*filter*) (function)

Let *filter* be a filter that is true for all elements of a family *Fam*, by the construction of *Fam*. Then CategoryCollections returns the *collections category* of *filter*. This is a category that is true for all elements in CollectionsFamily(*Fam*).

For example, the construction of PermutationsFamily (42.1.3) guarantees that each of its elements lies in the filter IsPerm (42.1.1), and each collection of permutations (permutation group or dense list of permutations) lies in the category CategoryCollections(IsPerm). CategoryCollections(IsPerm). Note that this works only if the collections category is created *before* the collections family. So it is necessary to construct interesting collections categories immediately after the underlying category has been created.

30.3 Lists and Collections

The following functions take a *list or collection* as argument, and return a corresponding *list*. They differ in whether or not the result is mutable or immutable (see 12.6), guaranteed to be sorted, or

guaranteed to admit list access in constant time (see IsConstantTimeAccessList (21.1.6)).

30.3.1 IsListOrCollection

▷ IsListOrCollection(*obj*) (Category)

Several functions are defined for both lists and collections, for example Intersection (30.5.2), Iterator (30.8.1), and Random (30.7.1). IsListOrCollection is a supercategory of IsList (21.1.1) and IsCollection (30.1.1) (that is, all lists and collections lie in this category), which is used to describe the arguments of functions such as the ones listed above.

30.3.2 Enumerator

▷ Enumerator(*listorcoll*) (attribute)

Enumerator returns an immutable list *enum*. If the argument is a list (which may contain holes), then Length(*enum*) is the length of this list, and *enum* contains the elements (and holes) of this list in the same order. If the argument is a collection that is not a list, then Length(*enum*) is the number of different elements of *C*, and *enum* contains the different elements of the collection in an unspecified order, which may change for repeated calls of Enumerator. *enum*[*pos*] may not execute in constant time (see IsConstantTimeAccessList (21.1.6)), and the size of *enum* in memory is as small as is feasible.

For lists, the default method is Immutable (12.6.3). For collections that are not lists, there is no default method.

30.3.3 EnumeratorSorted

▷ EnumeratorSorted(*listorcoll*) (attribute)

EnumeratorSorted returns an immutable list *enum*. The argument must be a collection or a list *listorcoll* which may contain holes but whose elements lie in the same family (see 13.1). Length(*enum*) is the number of different elements of the argument, and *enum* contains the different elements in sorted order, w.r.t. <. *enum*[*pos*] may not execute in constant time (see IsConstantTimeAccessList (21.1.6)), and the size of *enum* in memory is as small as is feasible.

Example

```
gap> Enumerator( [ 1, 3,, 2 ] );
[ 1, 3,, 2 ]
gap> enum:= Enumerator( Rationals );;  elm:= enum[ 10^6 ];
-69/907
gap> Position( enum, elm );
1000000
gap> IsMutable( enum );  IsSortedList( enum );
false
false
gap> IsConstantTimeAccessList( enum );
false
gap> EnumeratorSorted( [ 1, 3,, 2 ] );
[ 1, 2, 3 ]
```

30.3.4 EnumeratorByFunctions (for a domain and a record)

▷ `EnumeratorByFunctions(`*D, record*`)` (function)
▷ `EnumeratorByFunctions(`*Fam, record*`)` (function)

`EnumeratorByFunctions` returns an immutable, dense, and duplicate-free list *enum* for which `IsBound` (21.5.1), element access via `\[\]` (21.2.1), `Length` (21.17.5), and `Position` (21.16.1) are computed via prescribed functions.

Let *record* be a record with at least the following components.

`ElementNumber`
: a function taking two arguments *enum* and *pos*, which returns *enum*`[ `*pos*` ]` (see 21.2); it can be assumed that the argument *pos* is a positive integer, but *pos* may be larger than the length of *enum* (in which case an error must be signalled); note that the result must be immutable since *enum* itself is immutable,

`NumberElement`
: a function taking two arguments *enum* and *elm*, which returns `Position(` *enum*`, `*elm*` )` (see `Position` (21.16.1)); it cannot be assumed that *elm* is really contained in *enum* (and `fail` must be returned if not); note that for the three argument version of `Position` (21.16.1), the method that is available for duplicate-free lists suffices.

Further (data) components may be contained in *record* which can be used by these function.

If the first argument is a domain *D* then *enum* lists the elements of *D* (in general *enum* is *not* sorted), and methods for `Length` (21.17.5), `IsBound` (21.5.1), and `PrintObj` (6.3.5) may use *D*.

If one wants to describe the result without creating a domain then the elements are given implicitly by the functions in *record*, and the first argument must be a family *Fam* which will become the family of *enum*; if *enum* is not homogeneous then *Fam* must be `ListsFamily`, otherwise it must be the collections family of any element in *enum*. In this case, additionally the following component in *record* is needed.

`Length`
: a function taking the argument *enum*, which returns the length of *enum* (see `Length` (21.17.5)).

The following components are optional; they are used if they are present but default methods are installed for the case that they are missing.

`IsBound\[\]`
: a function taking two arguments *enum* and *k*, which returns `IsBound(` *enum*`[ `*k*` ] )` (see 21.2); if this component is missing then `Length` (21.17.5) is used for computing the result,

`Membership`
: a function taking two arguments *elm* and *enum*, which returns `true` is *elm* is an element of *enum*, and `false` otherwise (see 21.2); if this component is missing then `NumberElement` is used for computing the result,

`AsList`
: a function taking one argument *enum*, which returns a list with the property that the access to each of its elements will take roughly the same time (see `IsConstantTimeAccessList` (21.1.6)); if this component is missing then `ConstantTimeAccessList` (21.17.6) is used for computing the result,

`ViewObj` **and** `PrintObj`

two functions that print what one wants to be printed when `View( enum )` or `Print( enum )` is called (see 6.3), if the `ViewObj` component is missing then the `PrintObj` method is used as a default.

If the result is known to have additional properties such as being strictly sorted (see `IsSSortedList` (21.17.4)) then it can be useful to set these properties after the construction of the enumerator, before it is used for the first time. And in the case that a new sorted enumerator of a domain is implemented via `EnumeratorByFunctions`, and this construction is installed as a method for the operation `Enumerator` (30.3.2), then it should be installed also as a method for `EnumeratorSorted` (30.3.3).

Note that it is *not* checked that `EnumeratorByFunctions` really returns a dense and duplicate-free list. `EnumeratorByFunctions` does *not* make a shallow copy of *record*, this record is changed in place, see 79.9.

It would be easy to implement a slightly generalized setup for enumerators that need not be duplicate-free (where the three argument version of `Position` (21.16.1) is supported), but the resulting overhead for the methods seems not to be justified.

30.3.5 List (for a collection)

▷ `List(`*C*`)` (function)

For a collection *C* (see 30) that is not a list, `List` returns a new mutable list *new* such that `Length(` *new* `)` is the number of different elements of *C*, and *new* contains the different elements of *C* in an unspecified order which may change for repeated calls. *new*`[`*pos*`]` executes in constant time (see `IsConstantTimeAccessList` (21.1.6)), and the size of *new* is proportional to its length. The generic method for this case is `ShallowCopy( Enumerator(` *C* `) )`.

Example

```
gap> l:= List( Group( (1,2,3) ) );
[ (), (1,3,2), (1,2,3) ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
true
false
true
```

(See also `List` (21.20.19).)

30.3.6 SortedList

▷ `SortedList(`*listorcoll*`)` (operation)

`SortedList` returns a new mutable and dense list *new*. The argument must be a collection or list *listorcoll* which may contain holes but whose elements lie in the same family (see 13.1). `Length(` *new* `)` is the number of elements of *listorcoll*, and *new* contains the elements in sorted order, w.r.t. `<=`. *new*`[`*pos*`]` executes in constant time (see `IsConstantTimeAccessList` (21.1.6)), and the size of *new* in memory is proportional to its length.

Example

```
gap> l:= SortedList( Group( (1,2,3) ) );
[ (), (1,2,3), (1,3,2) ]
```

```
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
true
true
true
gap> SortedList( [ 1, 2, 1,, 3, 2 ] );
[ 1, 1, 2, 2, 3 ]
```

30.3.7 SSortedList

▷ SSortedList(*listorcoll*) (operation)
▷ Set(*C*) (operation)

SSortedList ("strictly sorted list") returns a new dense, mutable, and duplicate free list *new*. The argument must be a collection or list *listorcoll* which may contain holes but whose elements lie in the same family (see 13.1). Length(*new*) is the number of different elements of *listorcoll*, and *new* contains the different elements in strictly sorted order, w.r.t. \< (31.11.1). *new*[*pos*] executes in constant time (see IsConstantTimeAccessList (21.1.6)), and the size of *new* in memory is proportional to its length.

Set is simply a synonym for SSortedList.

Example

```
gap> l:= SSortedList( Group( (1,2,3) ) );
[ (), (1,2,3), (1,3,2) ]
gap> IsMutable( l );  IsSSortedList( l );  IsConstantTimeAccessList( l );
true
true
true
gap> SSortedList( [ 1, 2, 1,, 3, 2 ] );
[ 1, 2, 3 ]
```

30.3.8 AsList

▷ AsList(*listorcoll*) (attribute)

AsList returns a immutable list *imm*. If the argument is a list (which may contain holes), then Length(*imm*) is the Length (21.17.5) value of this list, and *imm* contains the elements (and holes) of of the list in the same order. If the argument is a collection that is not a list, then Length(*imm*) is the number of different elements of this collection, and *imm* contains the different elements of the collection in an unspecified order, which may change for repeated calls of AsList. *imm*[*pos*] executes in constant time (see IsConstantTimeAccessList (21.1.6)), and the size of *imm* in memory is proportional to its length.

If you expect to do many element tests in the resulting list, it might be worth to use a sorted list instead, using AsSSortedList (30.3.10).

Example

```
gap> l:= AsList( [ 1, 3, 3,, 2 ] );
[ 1, 3, 3,, 2 ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
false
false
```

```
true
gap> AsList( Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

30.3.9 AsSortedList

▷ AsSortedList(*listorcoll*) (attribute)

AsSortedList returns a dense and immutable list *imm*. The argument must be a collection or list *listorcoll* which may contain holes but whose elements lie in the same family (see 13.1). Length(*imm*) is the number of elements of the argument, and *imm* contains the elements in sorted order, w.r.t. <=. *new*[*pos*] executes in constant time (see IsConstantTimeAccessList (21.1.6)), and the size of *imm* in memory is proportional to its length.

The only difference to the operation SortedList (30.3.6) is that AsSortedList returns an *immutable* list.

Example

```
gap> l:= AsSortedList( [ 1, 3, 3,, 2 ] );
[ 1, 2, 3, 3 ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
false
true
true
gap> IsSSortedList( l );
false
```

30.3.10 AsSSortedList

▷ AsSSortedList(*listorcoll*) (attribute)
▷ AsSet(*listorcoll*) (attribute)

AsSSortedList ("as strictly sorted list") returns a dense, immutable, and duplicate free list *imm*. The argument must be a collection or list *listorcoll* which may contain holes but whose elements lie in the same family (see 13.1). Length(*imm*) is the number of different elements of *listorcoll*, and *imm* contains the different elements in strictly sorted order, w.r.t. \< (31.11.1). *imm*[*pos*] executes in constant time (see IsConstantTimeAccessList (21.1.6)), and the size of *imm* in memory is proportional to its length.

Because the comparisons required for sorting can be very expensive for some kinds of objects, you should use AsList (30.3.8) instead if you do not require the result to be sorted.

The only difference to the operation SSortedList (30.3.7) is that AsSSortedList returns an *immutable* list.

AsSet is simply a synonym for AsSSortedList.

In general a function that returns a set of elements is free, in fact encouraged, to return a domain instead of the proper set of its elements. This allows one to keep a given structure, and moreover the representation by a domain object is usually more space efficient. AsSSortedList must of course *not* do this, its only purpose is to create the proper set of elements.

Example

```
gap> l:= AsSSortedList( l );
[ 1, 2, 3 ]
```

```
gap> IsMutable( l );  IsSSortedList( l );  IsConstantTimeAccessList( l );
false
true
true
gap> AsSSortedList( Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

30.3.11 Elements

▷ `Elements(`*C*`)` (function)

`Elements` does the same as `AsSSortedList` (30.3.10), that is, the return value is a strictly sorted list of the elements in the list or collection *C*.

`Elements` is only supported for backwards compatibility. In many situations, the sortedness of the "element list" for a collection is in fact not needed, and one can save a lot of time by asking for a list that is *not* necessarily sorted, using `AsList` (30.3.8). If one is really interested in the strictly sorted list of elements in *C* then one should use `AsSet` (30.3.10) or `AsSSortedList` (30.3.10) instead.

30.4 Attributes and Properties for Collections

30.4.1 IsEmpty

▷ `IsEmpty(`*listorcoll*`)` (property)

`IsEmpty` returns `true` if the collection or list *listorcoll* is *empty* (that is it contains no elements), and `false` otherwise.

30.4.2 IsFinite

▷ `IsFinite(`*C*`)` (property)

`IsFinite` returns `true` if the collection *C* is finite, and `false` otherwise.

The default method for `IsFinite` checks the size (see `Size` (30.4.6)) of *C*.

Methods for `IsFinite` may call `Size` (30.4.6), but methods for `Size` (30.4.6) must *not* call `IsFinite`.

30.4.3 IsTrivial

▷ `IsTrivial(`*C*`)` (property)

`IsTrivial` returns `true` if the collection *C* consists of exactly one element.

30.4.4 IsNonTrivial

▷ `IsNonTrivial(`*C*`)` (property)

`IsNonTrivial` returns `true` if the collection *C* is empty or consists of at least two elements (see `IsTrivial` (30.4.3)).

Example

```
gap> IsEmpty( [] ); IsEmpty( [ 1 .. 100 ] ); IsEmpty( Group( (1,2,3) ) );
true
false
false
gap> IsFinite( [ 1 .. 100 ] );  IsFinite( Integers );
true
false
gap> IsTrivial( Integers );  IsTrivial( Group( () ) );
false
true
gap> IsNonTrivial( Integers );  IsNonTrivial( Group( () ) );
true
false
```

30.4.5 IsWholeFamily

▷ IsWholeFamily(*C*) (property)

IsWholeFamily returns true if the collection *C* contains the whole family (see 13.1) of its elements.

Example

```
gap> IsWholeFamily( Integers )
>    ;  # all rationals and cyclotomics lie in the family
false
gap> IsWholeFamily( Integers mod 3 )
>    ;  # all finite field elements in char. 3 lie in this family
false
gap> IsWholeFamily( Integers mod 4 );
true
gap> IsWholeFamily( FreeGroup( 2 ) );
true
```

30.4.6 Size

▷ Size(*listorcoll*) (attribute)

Size returns the size of the list or collection *listorcoll*, which is either an integer or infinity (18.2.1). If the argument is a list then the result is its length (see Length (21.17.5)).

The default method for Size checks the length of an enumerator of *listorcoll*.

Methods for IsFinite (30.4.2) may call Size, but methods for Size must not call IsFinite (30.4.2).

Example

```
gap> Size( [1,2,3] );  Size( Group( () ) );  Size( Integers );
3
1
infinity
```

30.4.7 Representative

▷ `Representative(C)` (attribute)

`Representative` returns a *representative* of the collection `C`.

Note that `Representative` is free in choosing a representative if there are several elements in `C`. It is not even guaranteed that `Representative` returns the same representative if it is called several times for one collection. The main difference between `Representative` and `Random` (30.7.1) is that `Representative` is free to choose a value that is cheap to compute, while `Random` (30.7.1) must make an effort to randomly distribute its answers.

If `C` is a domain then there are methods for `Representative` that try to fetch an element from any known generator list of `C`, see 31. Note that `Representative` does not try to *compute* generators of `C`, thus `Representative` may give up and signal an error if `C` has no generators stored at all.

30.4.8 RepresentativeSmallest

▷ `RepresentativeSmallest(C)` (attribute)

returns the smallest element in the collection `C`, w.r.t. the ordering `\<` (31.11.1). While the operation defaults to comparing all elements, better methods are installed for some collections.

Example

```
gap> Representative( Rationals );
0
gap> Representative( [ -1, -2 .. -100 ] );
-1
gap> RepresentativeSmallest( [ -1, -2 .. -100 ] );
-100
```

30.5 Operations for Collections

30.5.1 IsSubset

▷ `IsSubset(C1, C2)` (operation)

`IsSubset` returns `true` if `C2`, which must be a collection, is a *subset* of `C1`, which also must be a collection, and `false` otherwise.

`C2` is considered a subset of `C1` if and only if each element of `C2` is also an element of `C1`. That is `IsSubset` behaves as if implemented as `IsSubsetSet( AsSSortedList( C1 ), AsSSortedList( C2 ) )`, except that it will also sometimes, but not always, work for infinite collections, and that it will usually work much faster than the above definition. Either argument may also be a proper set (see 21.19).

Example

```
gap> IsSubset( Rationals, Integers );
true
gap> IsSubset( Integers, [ 1, 2, 3 ] );
true
gap> IsSubset( Group( (1,2,3,4) ), [ (1,2,3) ] );
false
```

30.5.2 Intersection

▷ Intersection(*C1, C2, ...*) (function)
▷ Intersection(*list*) (function)
▷ Intersection2(*C1, C2*) (operation)

In the first form Intersection returns the intersection of the collections *C1*, *C2*, etc. In the second form *list* must be a *nonempty* list of collections and Intersection returns the intersection of those collections. Each argument or element of *list* respectively may also be a homogeneous list that is not a proper set, in which case Intersection silently applies Set (30.3.7) to it first.

The result of Intersection is the set of elements that lie in every of the collections *C1*, *C2*, etc. If the result is a list then it is mutable and new, i.e., not identical to any of *C1*, *C2*, etc.

Methods can be installed for the operation Intersection2 that takes only two arguments. Intersection calls Intersection2.

Methods for Intersection2 should try to maintain as much structure as possible, for example the intersection of two permutation groups is again a permutation group.

Example

```
gap> # this is one of the rare cases where the intersection of two
gap> # infinite domains works ('CF' is a shorthand for 'CyclotomicField'):
gap> Intersection( CyclotomicField(9), CyclotomicField(12) );
CF(3)
gap> D12 := Group( (2,6)(3,5), (1,2)(3,6)(4,5) );;
gap> Intersection( D12, Group( (1,2), (1,2,3,4,5) ) );
Group([ (1,5)(2,4) ])
gap> Intersection( D12, [ (1,3)(4,6), (1,2)(3,4) ] )
>    ;  # note that the second argument is not a proper set
[ (1,3)(4,6) ]
gap> # although the result is mathematically a group it is returned as a
gap> # proper set because the second argument is not regarded as a group:
gap> Intersection( D12, [ (), (1,2)(3,4), (1,3)(4,6), (1,4)(5,6) ] );
[ (), (1,3)(4,6) ]
gap> Intersection( Group( () ), [1,2,3] );
[  ]
gap> Intersection( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] )
>    ;  # two or more lists or collections as arguments are legal
[  ]
gap> Intersection( [ [1,2,4], [2,3,4], [1,3,4] ] )
>    ;  # or one list of lists or collections
[ 4 ]
```

30.5.3 Union

▷ Union(*C1, C2, ...*) (function)
▷ Union(*list*) (function)
▷ Union2(*C1, C2*) (operation)

In the first form Union returns the union of the collections *C1*, *C2*, etc. In the second form *list* must be a list of collections and Union returns the union of those collections. Each argument or element of *list* respectively may also be a homogeneous list that is not a proper set, in which case Union silently applies Set (30.3.7) to it first.

The result of Union is the set of elements that lie in any of the collections *C1*, *C2*, etc. If the result is a list then it is mutable and new, i.e., not identical to any of *C1*, *C2*, etc.

Methods can be installed for the operation Union2 that takes only two arguments. Union calls Union2.

Example

```
gap> Union( [ (1,2,3), (1,2,3,4) ], Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,2,3,4), (1,3,2), (1,3) ]
gap> Union( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] )
>    ;  # two or more lists or collections as arguments are legal
[ 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 20, 25 ]
gap> Union( [ [1,2,4], [2,3,4], [1,3,4] ] )
>    ;  # or one list of lists or collections
[ 1, 2, 3, 4 ]
gap> Union( [ ] );
[  ]
```

30.5.4 Difference

▷ Difference(*C1, C2*) (operation)

Difference returns the set difference of the collections *C1* and *C2*. Either argument may also be a homogeneous list that is not a proper set, in which case Difference silently applies Set (30.3.7) to it first.

The result of Difference is the set of elements that lie in *C1* but not in *C2*. Note that *C2* need not be a subset of *C1*. The elements of *C2*, however, that are not elements of *C1* play no role for the result. If the result is a list then it is mutable and new, i.e., not identical to *C1* or *C2*.

Example

```
gap> Difference( [ (1,2,3), (1,2,3,4) ], Group( (1,2,3), (1,2) ) );
[ (1,2,3,4) ]
```

30.6 Membership Test for Collections

30.6.1 \in (for a collection)

▷ \in(*obj, C*) (operation)

returns true if the object *obj* lies in the collection *C*, and false otherwise.

The infix version of the command

obj in *C*

calls the operation \in (30.6.1), for which methods can be installed.

Example

```
gap> 13 in Integers;  [ 1, 2 ] in Integers;
true
false
gap> g:= Group( (1,2) );;  (1,2) in g;  (1,2,3) in g;
true
false
```

30.7 Random Elements

The method used by GAP to obtain random elements may depend on the type object.

Most methods which produce random elements in GAP use a global random number generator (see GlobalMersenneTwister (14.7.4)). This random number generator is (deliberately) initialized to the same values when GAP is started, so different runs of GAP with the same input will always produce the same result, even if random calculations are involved.

See Reset (14.7.3) for a description of how to reset the random number generator to a previous state.

30.7.1 Random (for a list or collection)

▷ Random(*listorcoll*) (operation)

▷ Random(*from*, *to*) (operation)

Random returns a (pseudo-)random element of the list or collection *listorcoll*.

As lists or ranges are restricted in length ($2^{28}-1$ or $2^{60}-1$ depending on your system), the second form returns a random integer in the range *from* to *to* (inclusive) for arbitrary integers *from* and *to*.

The distribution of elements returned by Random depends on the argument. For a list, all elements are equally likely. The same holds usually for finite collections that are not lists. For infinite collections some reasonable distribution is used.

See the chapters of the various collections to find out which distribution is being used.

For some collections ensuring a reasonable distribution can be difficult and require substantial runtime. If speed at the cost of equal distribution is desired, the operation PseudoRandom (30.7.2) should be used instead.

Note that Random is of course *not* an attribute.

Example

```
gap> Random(Rationals);
4
gap> g:= Group( (1,2,3) );;  Random( g );  Random( g );
(1,3,2)
()
```

30.7.2 PseudoRandom

▷ PseudoRandom(*listorcoll*) (operation)

PseudoRandom returns a pseudo random element of the list or collection *listorcoll*, which can be roughly described as follows. For a list, PseudoRandom returns the same as Random (30.7.1). For collections that are not lists, the elements returned by PseudoRandom are *not* necessarily equally distributed, even for finite collections; the idea is that Random (30.7.1) returns elements according to a reasonable distribution, PseudoRandom returns elements that are cheap to compute but need not satisfy this strong condition, and Representative (30.4.7) returns arbitrary elements, probably the same element for each call.

30.7.3 RandomList

▷ RandomList(*list*) (function)

For a dense list *list*, RandomList returns a (pseudo-)random element with equal distribution.

This function uses the GlobalMersenneTwister (14.7.4) to produce the random elements (a source of high quality random numbers).

30.8 Iterators

30.8.1 Iterator

▷ Iterator(*listorcoll*) (operation)

Iterators provide a possibility to loop over the elements of a (countable) collection or list *listorcoll*, without repetition. For many collections *C*, an iterator of *C* need not store all elements of *C*, for example it is possible to construct an iterator of some infinite domains, such as the field of rational numbers.

Iterator returns a mutable *iterator iter* for its argument. If this argument is a list (which may contain holes), then *iter* iterates over the elements (but not the holes) of this list in the same order (see IteratorList (30.8.6) for details). If this argument is a collection but not a list then *iter* iterates over the elements of this collection in an unspecified order, which may change for repeated calls of Iterator. Because iterators returned by Iterator are mutable (see 12.6), each call of Iterator for the same argument returns a *new* iterator. Therefore Iterator is not an attribute (see 13.5).

The only operations for iterators are IsDoneIterator (30.8.4), NextIterator (30.8.5), and ShallowCopy (12.7.1). In particular, it is only possible to access the next element of the iterator with NextIterator (30.8.5) if there is one, and this can be checked with IsDoneIterator (30.8.4) For an iterator *iter*, ShallowCopy (12.7.1) returns a mutable iterator *new* that iterates over the remaining elements independent of *iter*; the results of IsDoneIterator (30.8.4) for *iter* and *new* are equal, and if *iter* is mutable then also the results of NextIterator (30.8.5) for *iter* and *new* are equal; note that = is not defined for iterators, so the equality of two iterators cannot be checked with =.

When Iterator is called for a *mutable* collection *C* then it is not defined whether *iter* respects changes to *C* occurring after the construction of *iter*, except if the documentation explicitly promises a certain behaviour. The latter is the case if the argument is a mutable list (see IteratorList (30.8.6) for subtleties in this case).

It is possible to have for-loops run over mutable iterators instead of lists.

In some situations, one can construct iterators with a special succession of elements, see IteratorByBasis (61.6.6) for the possibility to loop over the elements of a vector space w.r.t. a given basis.

For lists, Iterator is implemented by IteratorList (30.8.6). For collections *C* that are not lists, the default method is IteratorList(Enumerator(*C*)). Better methods depending on *C* should be provided if possible.

For random access to the elements of a (possibly infinite) collection, *enumerators* are used. See 21.23 for the facility to compute a list from *C*, which provides a (partial) mapping from *C* to the positive integers.

Example

```
gap> iter:= Iterator( GF(5) );
<iterator>
```

```
gap> l:= [];;
gap> for i in iter do Add( l, i ); od; l;
[ 0*Z(5), Z(5)^0, Z(5), Z(5)^2, Z(5)^3 ]
gap> iter:= Iterator( [ 1, 2, 3, 4 ] );;  l:= [];;
gap> for i in iter do
>      new:= ShallowCopy( iter );
>      for j in new do Add( l, j ); od;
>    od; l;
[ 2, 3, 4, 3, 4, 4 ]
```

30.8.2 IteratorSorted

▷ IteratorSorted(*listorcoll*) (operation)

IteratorSorted returns a mutable iterator. The argument must be a collection or a list that is not necessarily dense but whose elements lie in the same family (see 13.1). It loops over the different elements in sorted order.

For a collection *C* that is not a list, the generic method is IteratorList(EnumeratorSorted(*C*)).

30.8.3 IsIterator

▷ IsIterator(*obj*) (Category)

Every iterator lies in the category IsIterator.

30.8.4 IsDoneIterator

▷ IsDoneIterator(*iter*) (operation)

If *iter* is an iterator for the list or collection *C* then IsDoneIterator(*iter*) is true if all elements of *C* have been returned already by NextIterator(*iter*), and false otherwise.

30.8.5 NextIterator

▷ NextIterator(*iter*) (operation)

Let *iter* be a mutable iterator for the list or collection *C*. If IsDoneIterator(*iter*) is false then NextIterator is applicable to *iter*, and the result is the next element of *C*, according to the succession defined by *iter*.

If IsDoneIterator(*iter*) is true then it is not defined what happens when NextIterator is called for *iter*; that is, it may happen that an error is signalled or that something meaningless is returned, or even that GAP crashes.

Example

```
gap> iter:= Iterator( [ 1, 2, 3, 4 ] );
<iterator>
gap> sum:= 0;;
gap> while not IsDoneIterator( iter ) do
>      sum:= sum + NextIterator( iter );
```

```
>    od;
gap> IsDoneIterator( iter ); sum;
true
10
gap> ir:= Iterator( Rationals );;
gap> l:= [];; for i in [1..20] do Add( l, NextIterator( ir ) ); od; l;
[ 0, 1, -1, 1/2, 2, -1/2, -2, 1/3, 2/3, 3/2, 3, -1/3, -2/3, -3/2, -3,
  1/4, 3/4, 4/3, 4, -1/4 ]
gap> for i in ir do
>      if DenominatorRat( i ) > 10 then break; fi;
>    od;
gap> i;
1/11
```

30.8.6 IteratorList

▷ `IteratorList(`*list*`)` (function)

`IteratorList` returns a new iterator that allows iteration over the elements of the list *list* (which may have holes) in the same order.

If *list* is mutable then it is in principle possible to change *list* after the call of `IteratorList`. In this case all changes concerning positions that have not yet been reached in the iteration will also affect the iterator. For example, if *list* is enlarged then the iterator will iterate also over the new elements at the end of the changed list.

Note that changes of *list* will also affect all shallow copies of *list*.

30.8.7 TrivialIterator

▷ `TrivialIterator(`*elm*`)` (function)

is a mutable iterator for the collection `[` *elm* `]` that consists of exactly one element *elm* (see `IsTrivial` (30.4.3)).

30.8.8 IteratorByFunctions

▷ `IteratorByFunctions(`*record*`)` (function)

`IteratorByFunctions` returns a (mutable) iterator *iter* for which `NextIterator` (30.8.5), `IsDoneIterator` (30.8.4), and `ShallowCopy` (12.7.1) are computed via prescribed functions.

Let *record* be a record with at least the following components.

`NextIterator`
: a function taking one argument *iter*, which returns the next element of *iter* (see `NextIterator` (30.8.5)); for that, the components of *iter* are changed,

`IsDoneIterator`
: a function taking one argument *iter*, which returns the `IsDoneIterator` (30.8.4) value of *iter*,

`ShallowCopy`
: a function taking one argument *`iter`*, which returns a record for which `IteratorByFunctions` can be called in order to create a new iterator that is independent of *`iter`* but behaves like *`iter`* w.r.t. the operations `NextIterator` (30.8.5) and `IsDoneIterator` (30.8.4).

Further (data) components may be contained in *`record`* which can be used by these function.

`IteratorByFunctions` does *not* make a shallow copy of *`record`*, this record is changed in place (see Section 79.9).

Chapter 31

Domains and their Elements

Domain is GAP's name for structured sets. The ring of Gaussian integers $\mathbb{Z}[\sqrt{-1}]$ is an example of a domain, the group D_{12} of symmetries of a regular hexahedron is another.

The GAP library predefines some domains. For example the ring of Gaussian integers is predefined as `GaussianIntegers` (60.5.1) (see 60.5) and the field of rationals is predefined as `Rationals` (17.1.1) (see 17). Most domains are constructed by functions, which are called *domain constructors* (see 31.3). For example the group D_{12} is constructed by the construction `Group( (1,2,3,4,5,6), (2,6)(3,5) )` (see `Group` (39.2.1)) and the finite field with 16 elements is constructed by `GaloisField( 16 )` (see `GaloisField` (59.3.2)).

The first place where you need domains in GAP is the obvious one. Sometimes you simply want to deal with a domain. For example if you want to compute the size of the group D_{12}, you had better be able to represent this group in a way that the `Size` (30.4.6) function can understand.

The second place where you need domains in GAP is when you want to be able to specify that an operation or computation takes place in a certain domain. For example suppose you want to factor 10 in the ring of Gaussian integers. Saying `Factors( 10 )` will not do, because this will return the factorization `[ 2, 5 ]` in the ring of integers. To allow operations and computations to happen in a specific domain, `Factors` (56.5.9), and many other functions as well, accept this domain as optional first argument. Thus `Factors( GaussianIntegers, 10 )` yields the desired result `[ 1+E(4), 1-E(4), 2+E(4), 2-E(4) ]`. (The imaginary unit $\sqrt{-1}$ is written as `E(4)` in GAP, see `E` (18.1.1).)

An introduction to the most important facts about domains is given in Chapter (**Tutorial: Domains**).

There are only few *operations* especially for domains (see 31.9), operations such as `Intersection` (30.5.2) and `Random` (30.7.1) are defined for the more general situation of collections (see Chapter 30).

31.1 Operational Structure of Domains

Domains have an *operational structure*, that is, a collection of operations under which the domain is closed. For example, a group is closed under multiplication, taking the zeroth power of elements, and taking inverses of elements. The operational structure may be empty, examples of domains without additional structure are the underlying relations of general mappings (see 32.3).

The operations under which a domain is closed are a subset of the operations that the elements of a domain admit. It is possible that the elements admit more operations. For example, matrices can be multiplied and added. But addition plays no role in a group of matrices, and multiplication plays no

role in a vector space of matrices. In particular, a matrix group is not closed under addition.

Note that the elements of a domain exist independently of this domain, usually they existed already before the domain was created. So it makes sense to say that a domain is *generated* by some elements with respect to certain operations.

Of course, different sets of operations yield different notions of generation. For example, the group generated by some matrices is different from the ring generated by these matrices, and these two will in general be different from the vector space generated by the same matrices, over a suitable field.

The other way round, the same set of elements may be obtained by generation w.r.t. different notions of generation. For example, one can get the group generated by two elements g and h also as the monoid generated by the elements g, g^{-1}, h, h^{-1}; if both g and h have finite order then of course the group generated by g and h coincides with the monoid generated by g and h.

Additionally to the operational structure, a domain can have properties. For example, the multiplication of a group is associative, and the multiplication in a field is commutative.

Note that associativity and commutativity depend on the set of elements for which one considers the multiplication, i.e., it depends on the domain. For example, the multiplication in a full matrix ring over a field is not commutative, whereas its restriction to the set of diagonal matrices is commutative.

One important difference between the operational structure and the properties of a domain is that the operational structure is fixed when the domain is constructed, whereas properties can be discovered later. For example, take a domain whose operational structure is given by closure under multiplication. If it is discovered that the inverses of all its elements also do (by chance) lie in this domain, being closed under taking inverses is *not* added to the operational structure. But a domain with operational structure of multiplication, taking the identity, and taking inverses will be treated as a group as soon as the multiplication is found out to be associative for this domain.

The operational structures available in GAP form a hierarchy, which is explicitly formulated in terms of domain categories, see 31.6.

31.2 Equality and Comparison of Domains

Equality and *comparison* of domains are defined as follows.

Two domains are considered *equal* if and only if the sets of their elements as computed by `AsSSortedList` (30.3.10)) are equal. Thus, in general = behaves as if each domain operand were replaced by its set of elements. Except that = will also sometimes, but not always, work for infinite domains, for which of course GAP cannot compute the set of elements. Note that this implies that domains with different algebraic structure may well be equal. As a special case of this, either operand of = may also be a proper set (see 21.19), i.e., a sorted list without holes or duplicates (see `AsSSortedList` (30.3.10)), and = will return `true` if and only if this proper set is equal to the set of elements of the argument that is a domain.

No general *ordering* of arbitrary domains via < is defined in GAP 4. This is because a well-defined < for domains or, more general, for collections, would have to be compatible with = and would need to be transitive and antisymmetric in order to be used to form ordered sets. In particular, < would have to be independent of the algebraic structure of its arguments because this holds for =, and thus there would be hardly a situation where one could implement an efficient comparison method. (Note that in the case that two domains are comparable with <, the result is in general *not* compatible with the set theoretical subset relation, which can be decided with `IsSubset` (30.5.1).)

31.3 Constructing Domains

For several operational structures (see 31.1), GAP provides functions to construct domains with this structure (note that such functions do not exist for all operational structures). For example, `Group` (39.2.1) returns groups, `VectorSpace` (61.2.1) returns vector spaces etc.:

Struct`( `*arg1*`, `*arg2*`, ... )`

The syntax of these functions may vary, dependent on the structure in question. Usually a domain is constructed as the closure of some elements under the given operations, that is, the domain is given by its *generators*. For example, a group can be constructed from a list of generating permutations or matrices or whatever is admissible as group elements, and a vector space over a given field F can be constructed from F and a list of appropriate vectors.

The idea of generation and generators in GAP is that the domain returned by a function such as `Group`, `Algebra`, or `FreeLeftModule` *contains* the given generators. This implies that the generators of a group must know how they are multiplied and inverted, the generators of a module must know how they are added and how scalar multiplication works, and so on. Thus one cannot use for example permutations as generators of a vector space.

The function *Struct* first checks whether the arguments admit the construction of a domain with the desired structure. This is done by calling the operation

`IsGeneratorsOf`*Struct*`( [`*info*`, ]`*gens*` )`

where *arglist* is the list of given generators and *info* an argument of *Struct*, for example the field of scalars in the case that a vector space shall be constructed. If the check failed then *Struct* returns `fail`, otherwise it returns the result of *Struct*`ByGenerators` (see below). (So if one wants to omit the check then one should call *Struct*`ByGenerators` directly.)

`GeneratorsOf`*Struct*`( `*D*`)`

For a domain *D* with operational structure corresponding to *Struct*, the attribute `GeneratorsOf`*Struct* returns a list of corresponding generators of *D*. If these generators were not yet stored in *D* then *D* must know *some* generators if `GeneratorsOf`*Struct* shall have a chance to compute the desired result; for example, monoid generators of a group can be computed from known group generators (and vice versa). Note that several notions of generation may be meaningful for a given domain, so it makes no sense to ask for "the generators of a domain". Further note that the generators may depend on other information about *D*. For example the generators of a vector space depend on the underlying field of scalars; the vector space generators of a vector space over the field with four elements need not generate the same vector space when this is viewed as a space over the field with two elements.

Struct`ByGenerators( [`*info*`, ]`*gens*` )`

Domain construction from generators *gens* is implemented by operations *Struct*`ByGenerators`, which are called by the simple functions *Struct*; methods can be installed only for the operations. Note that additional information *info* may be necessary to construct the domain; for example, a vector space needs the underlying field of scalars in addition to the list of vector space generators. The `GeneratorsOf`*Struct* value of the returned domain need *not* be equal to *gens*. But if a domain *D* is printed as *Struct*`([a, b, ...])` and if there is an attribute `GeneratorsOf`*Struct* then the list `GeneratorsOf`*Struct*`( `*D*` )` is guaranteed to be equal to `[ a, b, ... ]`.

Struct`WithGenerators( [`*info*`, ]`*gens*` )`

The only difference between *Struct*`ByGenerators` and *Struct*`WithGenerators` is that the latter guarantees that the `GeneratorsOf`*Struct* value of the result is equal to the given generators *gens*.

`Closure`*`Struct`*`( D, obj )`

For constructing a domain as the closure of a given domain with an element or another domain, one can use the operation `Closure`*`Struct`*. It returns the smallest domain with operational structure corresponding to *`Struct`* that contains *`D`* as a subset and *`obj`* as an element.

31.4 Changing the Structure

The same set of elements can have different operational structures. For example, it may happen that a monoid M does in fact contain the inverses of all of its elements; if M has not been constructed as a group (see 31.6) then it is reasonable to ask for the group that is equal to M.

`As`*`Struct`*`( [`*`info`*`, ]`*`D`*` )`

If *`D`* is a domain that is closed under the operational structure given by *`Struct`* then `As`*`Struct`* returns a domain *`E`* that consists of the same elements (that is, *`D`*` = `*`E`*) and that has this operational structure (that is, `Is`*`Struct`*`( `*`E`*` )` is `true`); if *`D`* is not closed under the structure given by *`Struct`* then `As`*`Struct`* returns `fail`.

If additional information besides generators are necessary to define *`D`* then the argument *`info`* describes the value of this information for the desired domain. For example, if we want to view *`D`* as a vector space over the field with two elements then we may call `AsVectorSpace( GF(2), `*`D`*` )`; this allows us to change the underlying field of scalars, for example if *`D`* is a vector space over the field with four elements. Again, if *`D`* is not equal to a domain with the desired structure and additional information then `fail` is returned.

In the case that no additional information *`info`* is related to the structure given by *`Struct`*, the operation `As`*`Struct`* is in fact an attribute (see 13.5).

See the index of the GAP Reference Manual for an overview of the available `As`*`Struct`* functions.

31.5 Changing the Representation

Often it is useful to answer questions about a domain via computations in a different but isomorphic domain. In the sense that this approach keeps the structure and changes the underlying set of elements, it can be viewed as a counterpart of keeping the set of elements and changing its structure (see 31.4).

One reason for doing so can be that computations with the elements in the given domain are not very efficient. For example, if one is given a solvable matrix group (see Chapter 44) then one can compute an isomorphism to a polycyclicly presented group G, say (see Chapter 45); the multiplication of two matrices –which is essentially determined by the dimension of the matrices– is much more expensive than the multiplication of two elements in G –which is essentially determined by the composition length of G.

`Isomorphism`*`Rep`*`Struct`*`( D )`

If *`D`* is a domain that is closed under the operational structure given by *`Struct`* then `Isomorphism`*`Rep`*`Struct` returns a mapping *`hom`* from *`D`* to a domain E having structure given by *`Struct`*, such that *`hom`* respects the structure *`Struct`* and *`Rep`* describes the representation of the elements in E. If no domain E with the required properties exists then `fail` is returned.

For example, `IsomorphismPermGroup` (43.3.1) takes a group as its argument and returns a group homomorphism (see 40) onto an isomorphic permutation group (see Chapter 43) provided the original group is finite; for infinite groups, `IsomorphismPermGroup` (43.3.1) returns `fail`. Similarly,

IsomorphismPcGroup (46.5.2) returns a group homomorphism from its argument to a polycyclicly presented group (see 46) if the argument is polycyclic, and fail otherwise.

See the index of the GAP Reference Manual for an overview of the available IsomorphismRepStruct functions.

31.6 Domain Categories

As mentioned in 31.1, the operational structure of a domain is fixed when the domain is constructed. For example, if D was constructed by Monoid (51.2.2) then D is in general not regarded as a group in GAP, even if D is in fact closed under taking inverses. In this case, IsGroup (39.2.7) returns false for D. The operational structure determines which operations are applicable for a domain, so for example SylowSubgroup (39.13.1) is not defined for D and therefore will signal an error.

IsStruct(D)

The functions IsStruct implement the tests whether a domain D has the respective operational structure (upon construction). IsStruct is a filter (see 13) that involves certain categories (see 13.3) and usually also certain properties (see 13.7). For example, IsGroup (39.2.7) is equivalent to IsMagmaWithInverses and IsAssociative, the first being a category and the second being a property.

Implications between domain categories describe the hierarchy of operational structures available in GAP. Here are some typical examples.

- IsDomain (31.9.1) is implied by each domain category,
- IsMagma (35.1.1) is implied by each category that describes the closure under multiplication *,
- IsAdditiveMagma (55.1.4) is implied by each category that describes the closure under addition +,
- IsMagmaWithOne (35.1.2) implies IsMagma (35.1.1); a *magma-with-one* is a magma such that each element (and thus also the magma itself) can be asked for its zeroth power,
- IsMagmaWithInverses (35.1.4) implies IsMagmaWithOne (35.1.2); a *magma-with-inverses* is a magma such that each element can be asked for its inverse; important special cases are *groups*, which in addition are associative,
- a *ring* is a magma that is also an additive group,
- a *ring-with-one* is a ring that is also a magma-with-one,
- a *division ring* is a ring-with-one that is also closed under taking inverses of nonzero elements,
- a *field* is a commutative division ring.

Each operational structure Struct has associated with it a domain category IsStruct, and operations StructByGenerators for constructing a domain from generators, GeneratorsOfStruct for storing and accessing generators w.r.t. this structure, ClosureStruct for forming the closure, and AsStruct for getting a domain with the desired structure from one with weaker operational structure and for testing whether a given domain can be regarded as a domain with Struct.

The functions applicable to domains with the various structures are described in the corresponding chapters of the Reference Manual. For example, functions for rings, fields, groups, and vector

spaces are described in Chapters 56, 58, 39, and 61, respectively. More general functions for arbitrary collections can be found in Chapter 30.

31.7 Parents

31.7.1 Parent

▷ `Parent(D)` (function)
▷ `SetParent(D, P)` (operation)
▷ `HasParent(D)` (function)

It is possible to assign to a domain `D` one other domain `P` containing `D` as a subset, in order to exploit this subset relation between `D` and `P`. Note that `P` need not have the same operational structure as `D`, for example `P` may be a magma and `D` a field.

The assignment is done by calling `SetParent`, and `P` is called the *parent* of `D`. If `D` has already a parent, calls to `SetParent` will be ignored.

If `D` has a parent `P` –this can be checked with `HasParent`– then `P` can be used to gain information about `D`. First, the call of `SetParent` causes `UseSubsetRelation` (31.13.1) to be called. Second, for a domain `D` with parent, information relative to the parent can be stored in `D`; for example, there is an attribute `NormalizerInParent` for storing `Normalizer( P, D )` in the case that `D` is a group. (More about such parent dependent attributes can be found in 85.2.) Note that because of this relative information, one cannot change the parent; that is, one can set the parent only once, subsequent calls to `SetParent` for the same domain `D` are ignored. Further note that contrary to `UseSubsetRelation` (31.13.1), also knowledge about the parent `P` might be used that is discovered after the `SetParent` call.

A stored parent can be accessed using `Parent`. If `D` has no parent then `Parent` returns `D` itself, and `HasParent` will return `false` also after a call to `Parent`. So `Parent` is *not* an attribute, the underlying attribute to store the parent is `ParentAttr`.

Certain functions that return domains with parent already set, for example `Subgroup` (39.3.1), are described in Section 31.8. Whenever a function has this property, the GAP Reference Manual states this explicitly. Note that these functions *do not guarantee* a certain parent, for example `DerivedSubgroup` (39.12.3) for a perfect group G may return G itself, and if G had already a parent then this is not replaced by G. As a rule of thumb, GAP avoids to set a domain as its own parent, which is consistent with the behaviour of `Parent`, at least until a parent is set explicitly with `SetParent`.

Example

```
gap> g:= Group( (1,2,3), (1,2) );; h:= Group( (1,2) );;
gap> HasParent( g );  HasParent( h );
false
false
gap> SetParent( h, g );
gap> Parent( g );  Parent( h );
Group([ (1,2,3), (1,2) ])
Group([ (1,2,3), (1,2) ])
gap> HasParent( g );  HasParent( h );
false
true
```

31.8 Constructing Subdomains

For many domains *D*, there are functions that construct certain subsets *S* of *D* as domains with parent (see 31.7) already set to *D*. For example, if *G* is a group that contains the elements in the list *gens* then Subgroup(*G*, *gens*) returns a group *S* that is generated by the elements in *gens* and with Parent(*S*) = *G*.

Sub*struct*(*D*, *gens*)

More general, if *D* is a domain whose algebraic structure is given by the function *Struct* (for example Group, Algebra, Field) then the function Sub*struct* (for example Subgroup, Subalgebra, Subfield) returns domains with structure *Struct* and parent set to the first argument.

Sub*struct*NC(*D*, *gens*)

Each function Sub*struct* checks that the *Struct* generated by *gens* is in fact a subset of *D*. If one wants to omit this check then one can call Sub*struct*NC instead; the suffix NC stands for "no check".

AsSub*struct*(*D*, *S*)

first constructs As*Struct*([*info*,]*S*), where *info* depends on *D* and *S*, and then sets the parent (see 31.7) of this new domain to *D*.

IsSub*struct*(*D*, *S*)

There is no real need for functions that check whether a domain *S* is a Sub*struct* of a domain *D*, since this is equivalent to the checks whether *S* is a *Struct* and *S* is a subset of *D*. Note that in many cases, only the subset relation is what one really wants to check, and that appropriate methods for the operation IsSubset (30.5.1) are available for many special situations, such as the test whether a group is contained in another group, where only generators need to be checked.

If a function IsSub*struct* is available in GAP then it is implemented as first a call to Is*Struct* for the second argument and then a call to IsSubset (30.5.1) for the two arguments.

31.9 Operations for Domains

For the meaning of the attributes Characteristic (31.10.1), One (31.10.2), Zero (31.10.3) in the case of a domain argument, see 31.10.

31.9.1 IsGeneralizedDomain

▷ IsGeneralizedDomain(*obj*) (Category)
▷ IsDomain(*obj*) (Category)

For some purposes, it is useful to deal with objects that are similar to domains but that are not collections in the sense of GAP because their elements may lie in different families; such objects are called *generalized domains*. An instance of generalized domains are "operation domains", for example any G-set for a permutation group G consisting of some union of points, sets of points, sets of sets of points etc., under a suitable action.

IsDomain is a synonym for IsGeneralizedDomain and IsCollection.

31.9.2 GeneratorsOfDomain

▷ GeneratorsOfDomain(*D*) (attribute)

For a domain *D*, GeneratorsOfDomain returns a list containing all elements of *D*, perhaps with repetitions. Note that if the domain *D* shall be generated by a list of some elements w.r.t. the empty operational structure (see 31.1), the only possible choice of elements is to take all elements of *D*. See 31.3 and 31.4 for concepts of other notions of generation.

For many domains that have *natural generators by construction* (for example, the natural generators of a free group of rank two are the two generators stored as value of the attribute GeneratorsOfGroup (39.2.4), and the natural generators of a free associative algebra are those generators stored as value of the attribute GeneratorsOfAlgebra (62.9.1)), each *natural* generator can be accessed using the . operator. For a domain *D*, *D*.i returns the *i*-th generator if *i* is a positive integer, and if name is the name of a generator of *D* then *D*.name returns this generator.

31.9.3 Domain

▷ Domain(*[Fam,]generators*) (function)
▷ DomainByGenerators(*Fam, generators*) (operation)

Domain returns the domain consisting of the elements in the homogeneous list *generators*. If *generators* is empty then a family *Fam* must be entered as the first argument, and the returned (empty) domain lies in the collections family of *Fam*.

DomainByGenerators is the operation called by Domain.

31.10 Attributes and Properties of Elements

The following attributes and properties for elements and domains correspond to the operational structure.

31.10.1 Characteristic

▷ Characteristic(*obj*) (attribute)

Characteristic returns the *characteristic* of *obj*.

If *obj* is a family, all of whose elements lie in IsAdditiveElementWithZero (31.14.5) then its characteristic is the least positive integer *n*, if any, such that IsZero(n*x) is true for all x in the family *obj*, otherwise it is 0.

If *obj* is a collections family of a family *g* which has a characteristic, then the characteristic of *obj* is the same as the characteristic of *g*.

For other families *obj* the characteristic is not defined and fail will be returned.

For any object *obj* which is in the filter IsAdditiveElementWithZero (31.14.5) or in the filter IsAdditiveMagmaWithZero (55.1.5) the characteristic of *obj* is the same as the characteristic of its family if that is defined and undefined otherwise.

For all other objects *obj* the characteristic is undefined and may return fail or a "no method found" error.

31.10.2 OneImmutable

▷ OneImmutable(*obj*) (attribute)
▷ OneAttr(*obj*) (attribute)

▷ One(*obj*) (attribute)
▷ Identity(*obj*) (attribute)
▷ OneMutable(*obj*) (operation)
▷ OneOp(*obj*) (operation)
▷ OneSameMutability(*obj*) (operation)
▷ OneSM(*obj*) (operation)

OneImmutable, OneMutable, and OneSameMutability return the multiplicative neutral element of the multiplicative element *obj*.

They differ only w.r.t. the mutability of the result. OneImmutable is an attribute and hence returns an immutable result. OneMutable is guaranteed to return a new *mutable* object whenever a mutable version of the required element exists in GAP (see IsCopyable (12.6.1)). OneSameMutability returns a result that is mutable if *obj* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

If *obj* is a multiplicative element then OneSameMutability(*obj*) is equivalent to *obj*^0.

OneAttr, One and Identity are synonyms of OneImmutable. OneSM is a synonym of OneSameMutability. OneOp is a synonym of OneMutable.

If *obj* is a domain or a family then One is defined as the identity element of all elements in *obj*, provided that all these elements have the same identity. For example, the family of all cyclotomics has the identity element 1, but a collections family (see CollectionsFamily (30.2.1)) may contain matrices of all dimensions and then it cannot have a unique identity element. Note that One is applicable to a domain only if it is a magma-with-one (see IsMagmaWithOne (35.1.2)); use MultiplicativeNeutralElement (35.4.10) otherwise.

The identity of an object need not be distinct from its zero, so for example a ring consisting of a single element can be regarded as a ring-with-one (see 56). This is particularly useful in the case of finitely presented algebras, where any factor of a free algebra-with-one is again an algebra-with-one, no matter whether or not it is a zero algebra.

The default method of One for multiplicative elements calls OneMutable (note that methods for OneMutable must *not* delegate to One); so other methods to compute identity elements need to be installed only for OneOp and (in the case of copyable objects) OneSameMutability.

For domains, One may call Representative (30.4.7), but Representative (30.4.7) is allowed to fetch the identity of a domain *D* only if HasOne(*D*) is true.

31.10.3 ZeroImmutable

▷ ZeroImmutable(*obj*) (attribute)
▷ ZeroAttr(*obj*) (attribute)
▷ Zero(*obj*) (attribute)
▷ ZeroMutable(*obj*) (operation)
▷ ZeroOp(*obj*) (operation)
▷ ZeroSameMutability(*obj*) (operation)
▷ ZeroSM(*obj*) (operation)

ZeroImmutable, ZeroMutable, and ZeroSameMutability all return the additive neutral element of the additive element *obj*.

They differ only w.r.t. the mutability of the result. ZeroImmutable is an attribute and hence returns an immutable result. ZeroMutable is guaranteed to return a new *mutable* object whenever a mutable version of the required element exists in GAP (see IsCopyable (12.6.1)). ZeroSameMutability returns a result that is mutable if *obj* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

ZeroSameMutability(*obj*) is equivalent to 0 * *obj*.

ZeroAttr and Zero are synonyms of ZeroImmutable. ZeroSM is a synonym of ZeroSameMutability. ZeroOp is a synonym of ZeroMutable.

If *obj* is a domain or a family then Zero is defined as the zero element of all elements in *obj*, provided that all these elements have the same zero. For example, the family of all cyclotomics has the zero element 0, but a collections family (see CollectionsFamily (30.2.1)) may contain matrices of all dimensions and then it cannot have a unique zero element. Note that Zero is applicable to a domain only if it is an additive magma-with-zero (see IsAdditiveMagmaWithZero (55.1.5)); use AdditiveNeutralElement (55.3.5) otherwise.

The default method of Zero for additive elements calls ZeroMutable (note that methods for ZeroMutable must *not* delegate to Zero); so other methods to compute zero elements need to be installed only for ZeroMutable and (in the case of copyable objects) ZeroSameMutability.

For domains, Zero may call Representative (30.4.7), but Representative (30.4.7) is allowed to fetch the zero of a domain *D* only if HasZero(*D*) is true.

31.10.4 MultiplicativeZeroOp

▷ MultiplicativeZeroOp(*elt*) (operation)

Returns: A multiplicative zero element.

for an element *elt* in the category IsMultiplicativeElementWithZero (31.14.12), MultiplicativeZeroOp returns the element z in the family F of *elt* with the property that $z*m = z = m*z$ holds for all $m \in F$, if such an element can be determined.

Families of elements in the category IsMultiplicativeElementWithZero (31.14.12) often arise from adjoining a new zero to an existing magma. See InjectionZeroMagma (35.2.13) or MagmaWithZeroAdjoined (35.2.13) for details.

Example

```
gap> G:=AlternatingGroup(5);;
gap> x:=Representative(MagmaWithZeroAdjoined(G));
<group with 0 adjoined elt: ()>
gap> MultiplicativeZeroOp(x);
<group with 0 adjoined elt: 0>
```

31.10.5 IsOne

▷ IsOne(*elm*) (property)

is true if *elm* = One(*elm*), and false otherwise.

31.10.6 IsZero

▷ IsZero(*elm*) (property)

is true if *elm* = Zero(*elm*), and false otherwise.

31.10.7 IsIdempotent

▷ IsIdempotent(*elt*) (property)

returns true iff *elt* is its own square. (Even if IsZero (31.10.6) returns true for *elt*.)

31.10.8 InverseImmutable

▷ InverseImmutable(*elm*) (attribute)
▷ InverseAttr(*elm*) (attribute)
▷ Inverse(*elm*) (attribute)
▷ InverseMutable(*elm*) (operation)
▷ InverseOp(*elm*) (operation)
▷ InverseSameMutability(*elm*) (operation)
▷ InverseSM(*elm*) (operation)

InverseImmutable, InverseMutable, and InverseSameMutability all return the multiplicative inverse of an element *elm*, that is, an element *inv* such that *elm* * *inv* = *inv* * *elm* = One(*elm*) holds; if *elm* is not invertible then fail (see 20.2) is returned.

Note that the above definition implies that a (general) mapping is invertible in the sense of Inverse only if its source equals its range (see 32.14). For a bijective mapping f whose source and range differ, InverseGeneralMapping (32.2.3) can be used to construct a mapping g with the property that $f * g$ is the identity mapping on the source of f and $g * f$ is the identity mapping on the range of f.

The operations differ only w.r.t. the mutability of the result. InverseImmutable is an attribute and hence returns an immutable result. InverseMutable is guaranteed to return a new *mutable* object whenever a mutable version of the required element exists in GAP. InverseSameMutability returns a result that is mutable if *elm* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

InverseSameMutability(*elm*) is equivalent to *elm*^-1.

InverseAttr and Inverse are synonyms of InverseImmutable. InverseSM is a synonym of InverseSameMutability. InverseOp is a synonym of InverseMutable.

The default method of InverseImmutable calls InverseMutable (note that methods for InverseMutable must *not* delegate to InverseImmutable); other methods to compute inverses need to be installed only for InverseMutable and (in the case of copyable objects) InverseSameMutability.

31.10.9 AdditiveInverseImmutable

▷ AdditiveInverseImmutable(*elm*) (attribute)
▷ AdditiveInverseAttr(*elm*) (attribute)

▷ AdditiveInverse(*elm*) (attribute)
▷ AdditiveInverseMutable(*elm*) (operation)
▷ AdditiveInverseOp(*elm*) (operation)
▷ AdditiveInverseSameMutability(*elm*) (operation)
▷ AdditiveInverseSM(*elm*) (operation)

AdditiveInverseImmutable, AdditiveInverseMutable, and AdditiveInverseSameMutability all return the additive inverse of *elm*.

They differ only w.r.t. the mutability of the result. AdditiveInverseImmutable is an attribute and hence returns an immutable result. AdditiveInverseMutable is guaranteed to return a new *mutable* object whenever a mutable version of the required element exists in GAP (see IsCopyable (12.6.1)). AdditiveInverseSameMutability returns a result that is mutable if *elm* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

AdditiveInverseSameMutability(*elm*) is equivalent to -*elm*.

AdditiveInverseAttr and AdditiveInverse are synonyms of AdditiveInverseImmutable. AdditiveInverseSM is a synonym of AdditiveInverseSameMutability. AdditiveInverseOp is a synonym of AdditiveInverseMutable.

The default method of AdditiveInverse calls AdditiveInverseMutable (note that methods for AdditiveInverseMutable must *not* delegate to AdditiveInverse); so other methods to compute additive inverses need to be installed only for AdditiveInverseMutable and (in the case of copyable objects) AdditiveInverseSameMutability.

31.10.10 Order

▷ Order(*elm*) (attribute)

is the multiplicative order of *elm*. This is the smallest positive integer *n* such that *elm* ^ *n* = One(*elm*) if such an integer exists. If the order is infinite, Order may return the value infinity (18.2.1), but it also might run into an infinite loop trying to test the order.

31.11 Comparison Operations for Elements

Binary comparison operations have been introduced already in 4.12. The underlying operations for which methods can be installed are the following.

31.11.1 \= and \<

▷ \=(*left-expr, right-expr*) (operation)
▷ \<(*left-expr, right-expr*) (operation)

Note that the comparisons via <>, <=, >, and >= are delegated to the operations \= (31.11.1) and \< (31.11.1).

In general, objects in *different* families cannot be compared with \< (31.11.1). For the reason and for exceptions from this rule, see 4.12.

31.11.2 CanEasilyCompareElements

▷ `CanEasilyCompareElements(`*obj*`)` (property)
▷ `CanEasilyCompareElementsFamily(`*fam*`)` (function)
▷ `CanEasilySortElements(`*obj*`)` (property)
▷ `CanEasilySortElementsFamily(`*fam*`)` (function)

For some objects a "normal form" is hard to compute and thus equality of elements of a domain might be expensive to test. Therefore GAP provides a (slightly technical) property with which an algorithm can test whether an efficient equality test is available for elements of a certain kind.

`CanEasilyCompareElements` indicates whether the elements in the family *fam* of *obj* can be easily compared with \= (31.11.1).

The default method for this property is to ask the family of *obj*, the default method for the family is to return `false`.

The ability to compare elements may depend on the successful computation of certain information. (For example for finitely presented groups it might depend on the knowledge of a faithful permutation representation.) This information might change over time and thus it might not be a good idea to store a value `false` too early in a family. Instead the function `CanEasilyCompareElementsFamily` should be called for the family of *obj* which returns `false` if the value of `CanEasilyCompareElements` is not known for the family without computing it. (This is in fact what the above mentioned family dispatch does.)

If a family knows ab initio that it can compare elements this property should be set as implied filter *and* filter for the family (the 3rd and 4th argument of `NewFamily` (79.7.1) respectively). This guarantees that code which directly asks the family gets a right answer.

The property `CanEasilySortElements` and the function `CanEasilySortElementsFamily` behave exactly in the same way, except that they indicate that objects can be compared via \< (31.11.1). This property implies `CanEasilyCompareElements`, as the ordering must be total.

31.12 Arithmetic Operations for Elements

Binary arithmetic operations have been introduced already in 4.13. The underlying operations for which methods can be installed are the following.

31.12.1 \+, *, \/, \^, \mod

▷ `\+(`*left-expr, right-expr*`)` (operation)
▷ `\*(`*left-expr, right-expr*`)` (operation)
▷ `\/(`*left-expr, right-expr*`)` (operation)
▷ `\^(`*left-expr, right-expr*`)` (operation)
▷ `\mod(`*left-expr, right-expr*`)` (operation)

For details about special methods for * (31.12.1), \/ (31.12.1), \^ (31.12.1) and `\mod` (31.12.1), consult the appropriate index entries for them.

31.12.2 LeftQuotient

▷ LeftQuotient(*elm1, elm2*) (operation)

returns the product *elm1*^(-1) * *elm2*. For some types of objects (for example permutations) this product can be evaluated more efficiently than by first inverting *elm1* and then forming the product with *elm2*.

31.12.3 Comm

▷ Comm(*elm1, elm2*) (operation)

returns the *commutator* of *elm1* and *elm2*. The commutator is defined as the product $elm1^{-1} * elm2^{-1} * elm1 * elm2$.

Example

```
gap> a:= (1,3)(4,6);; b:= (1,6,5,4,3,2);;
gap> Comm( a, b );
(1,5,3)(2,6,4)
gap> LeftQuotient( a, b );
(1,2)(3,6)(4,5)
```

31.12.4 LieBracket

▷ LieBracket(*elm1, elm2*) (operation)

returns the element *elm1 * elm2 - elm2 * elm1*.

The addition \+ (31.12.1) is assumed to be associative but *not* assumed to be commutative (see IsAdditivelyCommutative (55.3.1)). The multiplication * (31.12.1) is *not* assumed to be commutative or associative (see IsCommutative (35.4.9), IsAssociative (35.4.7)).

31.12.5 Sqrt

▷ Sqrt(*obj*) (operation)

Sqrt returns a square root of *obj*, that is, an object x with the property that $x \cdot x = obj$ holds. If such an x is not unique then the choice of x depends on the type of *obj*. For example, ER (18.4.2) is the Sqrt method for rationals (see IsRat (17.2.1)).

31.13 Relations Between Domains

Domains are often constructed relative to other domains. The probably most usual case is to form a *subset* of a domain, for example the intersection (see Intersection (30.5.2)) of two domains, or a Sylow subgroup of a given group (see SylowSubgroup (39.13.1)).

In such a situation, the new domain can gain knowledge by exploiting that several attributes are maintained under taking subsets. For example, the intersection of an arbitrary domain with a finite domain is clearly finite, a Sylow subgroup of an abelian group is abelian, too, and so on.

Since usually the new domain has access to the knowledge of the old domain(s) only when it is created (see 31.8 for the exception), this is the right moment to take advantage of the subset relation, using UseSubsetRelation (31.13.1).

Analogous relations occur when a *factor structure* is created from a domain and a subset (see UseFactorRelation (31.13.2)), and when a domain *isomorphic* to a given one is created (see UseIsomorphismRelation (31.13.3)).

The functions InstallSubsetMaintenance (31.13.4), InstallIsomorphismMaintenance (31.13.6), and InstallFactorMaintenance (31.13.5) are used to tell GAP under what conditions an attribute is maintained under taking subsets, or forming factor structures or isomorphic domains. This is used only when a new attribute is created, see 79.3. For the attributes already available, such as IsFinite (30.4.2) and IsCommutative (35.4.9), the maintenances are already notified.

31.13.1 UseSubsetRelation

▷ UseSubsetRelation(*super, sub*) (operation)

Methods for this operation transfer possibly useful information from the domain *super* to its subset *sub*, and vice versa.

UseSubsetRelation is designed to be called automatically whenever substructures of domains are constructed. So the methods must be *cheap*, and the requirements should be as sharp as possible!

To achieve that *all* applicable methods are executed, all methods for this operation except the default method must end with TryNextMethod(). This default method deals with the information that is available by the calls of InstallSubsetMaintenance (31.13.4) in the GAP library.

Example

```
gap> g:= Group( (1,2), (3,4), (5,6) );; h:= Group( (1,2), (3,4) );;
gap> IsAbelian( g );  HasIsAbelian( h );
true
false
gap> UseSubsetRelation( g, h );;  HasIsAbelian( h );  IsAbelian( h );
true
true
```

31.13.2 UseFactorRelation

▷ UseFactorRelation(*numer, denom, factor*) (operation)

Methods for this operation transfer possibly useful information from the domain *numer* or its subset *denom* to the domain *factor* that is isomorphic to the factor of *numer* by *denom*, and vice versa. *denom* may be fail, for example if *factor* is just known to be a factor of *numer* but *denom* is not available as a GAP object; in this case those factor relations are used that are installed without special requirements for *denom*.

UseFactorRelation is designed to be called automatically whenever factor structures of domains are constructed. So the methods must be *cheap*, and the requirements should be as sharp as possible!

To achieve that *all* applicable methods are executed, all methods for this operation except the default method must end with a call to TryNextMethod (78.4.1). This default method deals with the information that is available by the calls of InstallFactorMaintenance (31.13.5) in the GAP library.

Example

```
gap> g:= Group( (1,2,3,4), (1,2) );; h:= Group( (1,2,3), (1,2) );;
gap> IsSolvableGroup( g );  HasIsSolvableGroup( h );
true
false
gap> UseFactorRelation(g, Subgroup( g, [ (1,2)(3,4), (1,3)(2,4) ] ), h);;
gap> HasIsSolvableGroup( h );  IsSolvableGroup( h );
true
true
```

31.13.3 UseIsomorphismRelation

▷ `UseIsomorphismRelation(`*old, new*`)` (operation)

Methods for this operation transfer possibly useful information from the domain *old* to the isomorphic domain *new*.

`UseIsomorphismRelation` is designed to be called automatically whenever isomorphic structures of domains are constructed. So the methods must be *cheap*, and the requirements should be as sharp as possible!

To achieve that *all* applicable methods are executed, all methods for this operation except the default method must end with a call to `TryNextMethod` (78.4.1). This default method deals with the information that is available by the calls of `InstallIsomorphismMaintenance` (31.13.6) in the GAP library.

Example

```
gap> g:= Group( (1,2) );;  h:= Group( [ [ -1 ] ] );;
gap> Size( g );  HasSize( h );
2
false
gap> UseIsomorphismRelation( g, h );;  HasSize( h );  Size( h );
true
2
```

31.13.4 InstallSubsetMaintenance

▷ `InstallSubsetMaintenance(`*opr, super_req, sub_req*`)` (function)

opr must be a property or an attribute. The call of `InstallSubsetMaintenance` has the effect that for a domain D in the filter *super_req*, and a domain S in the filter *sub_req*, the call `UseSubsetRelation`(D,S) (see `UseSubsetRelation` (31.13.1)) sets a known value of *opr* for D as value of *opr* also for S. A typical example for which `InstallSubsetMaintenance` is applied is given by *opr* = `IsFinite`, *super_req* = `IsCollection and IsFinite`, and *sub_req* = `IsCollection`.

If *opr* is a property and the filter *super_req* lies in the filter *opr* then we can use also the following inverse implication. If D is in the filter whose intersection with *opr* is *super_req* and if S is in the filter *sub_req*, S is a subset of D, and the value of *opr* for S is `false` then the value of *opr* for D is also `false`.

31.13.5 InstallFactorMaintenance

▷ `InstallFactorMaintenance(`*opr, numer_req, denom_req, factor_req*`)` (function)

opr must be a property or an attribute. The call of `InstallFactorMaintenance` has the effect that for collections N, D, F in the filters *numer_req*, *denom_req*, and *factor_req*, respectively, the call `UseFactorRelation`(N,D,F) (see `UseFactorRelation` (31.13.2)) sets a known value of *opr* for N as value of *opr* also for F. A typical example for which `InstallFactorMaintenance` is applied is given by *opr* = `IsFinite`, *numer_req* = `IsCollection and IsFinite`, *denom_req* = `IsCollection`, and *factor_req* = `IsCollection`.

For the other direction, if *numer_req* involves the filter *opr* then a known `false` value of *opr* for F implies a `false` value for D provided that D lies in the filter obtained from *numer_req* by removing *opr*.

Note that an implication of a factor relation holds in particular for the case of isomorphisms. So one need *not* install an isomorphism maintained method when a factor maintained method is already installed. For example, `UseIsomorphismRelation` (31.13.3) will transfer a known `IsFinite` (30.4.2) value because of the installed factor maintained method.

31.13.6 InstallIsomorphismMaintenance

▷ `InstallIsomorphismMaintenance(`*opr, old_req, new_req*`)` (function)

opr must be a property or an attribute. The call of `InstallIsomorphismMaintenance` has the effect that for a domain D in the filter *old_req*, and a domain E in the filter *new_req*, the call `UseIsomorphismRelation`(D,E) (see `UseIsomorphismRelation` (31.13.3)) sets a known value of *opr* for D as value of *opr* also for E. A typical example for which `InstallIsomorphismMaintenance` is applied is given by *opr* = `Size`, *old_req* = `IsCollection`, and *new_req* = `IsCollection`.

31.14 Useful Categories of Elements

This section and the following one are rather technical, and may be interesting only for those GAP users who want to implement new kinds of elements.

It deals with certain categories of elements that are useful mainly for the design of elements, from the viewpoint that one wants to form certain domains of these elements. For example, a domain closed under multiplication `*` (a so-called magma, see Chapter 35) makes sense only if its elements can be multiplied, and the latter is indicated by the category `IsMultiplicativeElement` (31.14.10) for each element. Again note that the underlying idea is that a domain is regarded as *generated* by given elements, and that these elements carry information about the desired domain. For general information on categories and their hierarchies, see 13.3.

More special categories of this kind are described in the contexts where they arise, they are `IsRowVector` (23.1.1), `IsMatrix` (24.2.1), `IsOrdinaryMatrix` (24.2.2), and `IsLieMatrix` (24.2.3).

31.14.1 IsExtAElement

▷ `IsExtAElement(`*obj*`)` (Category)

An *external additive element* is an object that can be added via + with other elements (not necessarily in the same family, see 13.1).

31.14.2 IsNearAdditiveElement

▷ `IsNearAdditiveElement(`*obj*`)` (Category)

A *near-additive element* is an object that can be added via + with elements in its family (see 13.1); this addition is not necessarily commutative.

31.14.3 IsAdditiveElement

▷ `IsAdditiveElement(`*obj*`)` (Category)

An *additive element* is an object that can be added via + with elements in its family (see 13.1); this addition is commutative.

31.14.4 IsNearAdditiveElementWithZero

▷ `IsNearAdditiveElementWithZero(`*obj*`)` (Category)

A *near-additive element-with-zero* is an object that can be added via + with elements in its family (see 13.1), and that is an admissible argument for the operation `Zero` (31.10.3); this addition is not necessarily commutative.

31.14.5 IsAdditiveElementWithZero

▷ `IsAdditiveElementWithZero(`*obj*`)` (Category)

An *additive element-with-zero* is an object that can be added via + with elements in its family (see 13.1), and that is an admissible argument for the operation `Zero` (31.10.3); this addition is commutative.

31.14.6 IsNearAdditiveElementWithInverse

▷ `IsNearAdditiveElementWithInverse(`*obj*`)` (Category)

A *near-additive element-with-inverse* is an object that can be added via + with elements in its family (see 13.1), and that is an admissible argument for the operations `Zero` (31.10.3) and `AdditiveInverse` (31.10.9); this addition is not necessarily commutative.

31.14.7 IsAdditiveElementWithInverse

▷ IsAdditiveElementWithInverse(*obj*) (Category)

An *additive element-with-inverse* is an object that can be added via + with elements in its family (see 13.1), and that is an admissible argument for the operations Zero (31.10.3) and AdditiveInverse (31.10.9); this addition is commutative.

31.14.8 IsExtLElement

▷ IsExtLElement(*obj*) (Category)

An *external left element* is an object that can be multiplied from the left, via *, with other elements (not necessarily in the same family, see 13.1).

31.14.9 IsExtRElement

▷ IsExtRElement(*obj*) (Category)

An *external right element* is an object that can be multiplied from the right, via *, with other elements (not necessarily in the same family, see 13.1).

31.14.10 IsMultiplicativeElement

▷ IsMultiplicativeElement(*obj*) (Category)

A *multiplicative element* is an object that can be multiplied via * with elements in its family (see 13.1).

31.14.11 IsMultiplicativeElementWithOne

▷ IsMultiplicativeElementWithOne(*obj*) (Category)

A *multiplicative element-with-one* is an object that can be multiplied via * with elements in its family (see 13.1), and that is an admissible argument for the operation One (31.10.2).

31.14.12 IsMultiplicativeElementWithZero

▷ IsMultiplicativeElementWithZero(*elt*) (Category)

Returns: true or false.

This is the category of elements in a family which can be the operands of * (multiplication) and the operation MultiplicativeZero (35.4.11).

Example

```
gap> S:=Semigroup(Transformation( [ 1, 1, 1 ] ));;
gap> M:=MagmaWithZeroAdjoined(S);
<<commutative transformation semigroup on 3 pts with 1 generator>
with 0 adjoined>
gap> x:=Representative(M);
<semigroup with 0 adjoined elt: Transformation( [ 1, 1, 1 ] )>
```

```
gap> IsMultiplicativeElementWithZero(x);
true
gap> MultiplicativeZeroOp(x);
<semigroup with 0 adjoined elt: 0>
```

31.14.13 IsMultiplicativeElementWithInverse

▷ `IsMultiplicativeElementWithInverse(`*obj*`)` (Category)

A *multiplicative element-with-inverse* is an object that can be multiplied via * with elements in its family (see 13.1), and that is an admissible argument for the operations `One` (31.10.2) and `Inverse` (31.10.8). (Note the word "admissible": an object in this category does not necessarily have an inverse, `Inverse` (31.10.8) may return `fail`.)

31.14.14 IsVector

▷ `IsVector(`*obj*`)` (Category)

A *vector* is an additive-element-with-inverse that can be multiplied from the left and right with other objects (not necessarily of the same type). Examples are cyclotomics, finite field elements, and of course row vectors (see below).

Note that not all lists of ring elements are regarded as vectors, for example lists of matrices are not vectors. This is because although the category `IsAdditiveElementWithInverse` (31.14.7) is implied by the meet of its collections category and `IsList` (21.1.1), the family of a list entry may not imply `IsAdditiveElementWithInverse` (31.14.7) for all its elements.

31.14.15 IsNearRingElement

▷ `IsNearRingElement(`*obj*`)` (Category)

`IsNearRingElement` is just a synonym for the meet of `IsNearAdditiveElementWithInverse` (31.14.6) and `IsMultiplicativeElement` (31.14.10).

31.14.16 IsRingElement

▷ `IsRingElement(`*obj*`)` (Category)

`IsRingElement` is just a synonym for the meet of `IsAdditiveElementWithInverse` (31.14.7) and `IsMultiplicativeElement` (31.14.10).

31.14.17 IsNearRingElementWithOne

▷ `IsNearRingElementWithOne(`*obj*`)` (Category)

`IsNearRingElementWithOne` is just a synonym for the meet of `IsNearAdditiveElementWithInverse` (31.14.6) and `IsMultiplicativeElementWithOne` (31.14.11).

31.14.18 IsRingElementWithOne

▷ IsRingElementWithOne(*obj*) (Category)

IsRingElementWithOne is just a synonym for the meet of IsAdditiveElementWithInverse (31.14.7) and IsMultiplicativeElementWithOne (31.14.11).

31.14.19 IsNearRingElementWithInverse

▷ IsNearRingElementWithInverse(*obj*) (Category)

IsNearRingElementWithInverse is just a synonym for the meet of IsNearAdditiveElementWithInverse (31.14.6) and IsMultiplicativeElementWithInverse (31.14.13).

31.14.20 IsRingElementWithInverse

▷ IsRingElementWithInverse(*obj*) (Category)
▷ IsScalar(*obj*) (Category)

IsRingElementWithInverse and IsScalar are just synonyms for the meet of IsAdditiveElementWithInverse (31.14.7) and IsMultiplicativeElementWithInverse (31.14.13).

31.15 Useful Categories for all Elements of a Family

The following categories of elements are to be understood mainly as categories for all objects in a family, they are usually used as third argument of NewFamily (see 79.7). The purpose of each of the following categories is then to guarantee that each collection of its elements automatically lies in its collections category (see CategoryCollections (30.2.4)).

For example, the multiplication of permutations is associative, and it is stored in the family of permutations that each permutation lies in IsAssociativeElement (31.15.1). As a consequence, each magma consisting of permutations (more precisely: each collection that lies in the family CollectionsFamily(PermutationsFamily), see CollectionsFamily (30.2.1)) automatically lies in CategoryCollections(IsAssociativeElement). A magma in this category is always known to be associative, via a logical implication (see 78.7).

Similarly, if a family knows that all its elements are in the categories IsJacobianElement (31.15.5) and IsZeroSquaredElement (31.15.6), then each algebra of these elements is automatically known to be a Lie algebra (see Chapter 62).

31.15.1 IsAssociativeElement

▷ IsAssociativeElement(*obj*) (Category)
▷ IsAssociativeElementCollection(*obj*) (Category)
▷ IsAssociativeElementCollColl(*obj*) (Category)

An element *obj* in the category IsAssociativeElement knows that the multiplication of any elements in the family of *obj* is associative. For example, all permutations lie in this category, as well as those ordinary matrices (see IsOrdinaryMatrix (24.2.2)) whose entries are also in IsAssociativeElement.

31.15.2 IsAdditivelyCommutativeElement

▷ IsAdditivelyCommutativeElement(*obj*) (Category)
▷ IsAdditivelyCommutativeElementCollection(*obj*) (Category)
▷ IsAdditivelyCommutativeElementCollColl(*obj*) (Category)
▷ IsAdditivelyCommutativeElementFamily(*obj*) (Category)

An element *obj* in the category IsAdditivelyCommutativeElement knows that the addition of any elements in the family of *obj* is commutative. For example, each finite field element and each rational number lies in this category.

31.15.3 IsCommutativeElement

▷ IsCommutativeElement(*obj*) (Category)
▷ IsCommutativeElementCollection(*obj*) (Category)
▷ IsCommutativeElementCollColl(*obj*) (Category)

An element *obj* in the category IsCommutativeElement knows that the multiplication of any elements in the family of *obj* is commutative. For example, each finite field element and each rational number lies in this category.

31.15.4 IsFiniteOrderElement

▷ IsFiniteOrderElement(*obj*) (Category)
▷ IsFiniteOrderElementCollection(*obj*) (Category)
▷ IsFiniteOrderElementCollColl(*obj*) (Category)

An element *obj* in the category IsFiniteOrderElement knows that it has finite multiplicative order. For example, each finite field element and each permutation lies in this category. However the value may be false even if *obj* has finite order, but if this was not known when *obj* was constructed.

Although it is legal to set this filter for any object with finite order, this is really useful only in the case that all elements of a family are known to have finite order.

31.15.5 IsJacobianElement

▷ IsJacobianElement(*obj*) (Category)
▷ IsJacobianElementCollection(*obj*) (Category)
▷ IsJacobianElementCollColl(*obj*) (Category)
▷ IsRestrictedJacobianElement(*obj*) (Category)
▷ IsRestrictedJacobianElementCollection(*obj*) (Category)
▷ IsRestrictedJacobianElementCollColl(*obj*) (Category)

An element *obj* in the category IsJacobianElement knows that the multiplication of any elements in the family F of *obj* satisfies the Jacobi identity, that is, $x*y*z+z*x*y+y*z*x$ is zero for all x, y, z in F.

For example, each Lie matrix (see IsLieMatrix (24.2.3)) lies in this category.

31.15.6 IsZeroSquaredElement

▷ IsZeroSquaredElement(*obj*) (Category)
▷ IsZeroSquaredElementCollection(*obj*) (Category)
▷ IsZeroSquaredElementCollColl(*obj*) (Category)

An element *obj* in the category IsZeroSquaredElement knows that *obj*^2 = Zero(*obj*). For example, each Lie matrix (see IsLieMatrix (24.2.3)) lies in this category.

Although it is legal to set this filter for any zero squared object, this is really useful only in the case that all elements of a family are known to have square zero.

Chapter 32

Mappings

A *mapping* in GAP is what is called a "function" in mathematics. GAP also implements *generalized mappings* in which one element might have several images, these can be imagined as subsets of the cartesian product and are often called "relations".

Most operations are declared for general mappings and therefore this manual often refers to "(general) mappings", unless you deliberately need the generalization you can ignore the "general" bit and just read it as "mappings".

A *general mapping* F in GAP is described by its source S, its range R, and a subset Rel of the direct product $S \times R$, which is called the underlying relation of F. S, R, and Rel are generalized domains (see 12.4). The corresponding attributes for general mappings are `Source` (32.3.8), `Range` (32.3.7), and `UnderlyingRelation` (32.3.9).

Note that general mappings themselves are *not* domains. One reason for this is that two general mappings with same underlying relation are regarded as equal only if also the sources are equal and the ranges are equal. Other, more technical, reasons are that general mappings and domains have different basic operations, and that general mappings are arithmetic objects (see 32.6); both should not apply to domains.

Each element of an underlying relation of a general mapping lies in the category of direct product elements (see `IsDirectProductElement` (32.1.1)).

For each $s \in S$, the set $\{r \in R | (s,r) \in Rel\}$ is called the set of *images* of s. Analogously, for $r \in R$, the set $\{s \in S | (s,r) \in Rel\}$ is called the set of *preimages* of r.

The *ordering* of general mappings via < is defined by the ordering of source, range, and underlying relation. Specifically, if the source and range domains of *map1* and *map2* are the same, then one considers the union of the preimages of *map1* and *map2* as a strictly ordered set. The smaller of *map1* and *map2* is the one whose image is smaller on the first point of this sequence where they differ.

For mappings which preserve an algebraic structure a *kernel* is defined. Depending on the structure preserved the operation to compute this kernel is called differently, see Section 32.7.

Some technical details of general mappings are described in section 32.13.

32.1 IsDirectProductElement (Filter)

32.1.1 IsDirectProductElement

▷ `IsDirectProductElement(`*obj*`)` (Category)

IsDirectProductElement is a subcategory of the meet of IsDenseList (21.1.2), IsMultiplicativeElementWithInverse (31.14.13), IsAdditiveElementWithInverse (31.14.7), and IsCopyable (12.6.1), where the arithmetic operations (addition, zero, additive inverse, multiplication, powering, one, inverse) are defined componentwise.

Note that each of these operations will cause an error message if its result for at least one component cannot be formed.

For an object in the filter IsDirectProductElement, ShallowCopy (12.7.1) returns a mutable plain list with the same entries. The sum and the product of a direct product element and a list in IsListDefault (21.12.3) is the list of sums and products, respectively. The sum and the product of a direct product element and a non-list is the direct product element of componentwise sums and products, respectively.

32.2 Creating Mappings

32.2.1 GeneralMappingByElements

▷ GeneralMappingByElements(*S, R, elms*) (function)

is the general mapping with source *S* and range *R*, and with underlying relation consisting of the collection *elms* of direct product elements.

32.2.2 MappingByFunction

▷ MappingByFunction(*S, R, fun[, invfun]*) (function)
▷ MappingByFunction(*S, R, fun, false, prefun*) (function)

MappingByFunction returns a mapping map with source *S* and range *R*, such that each element s of *S* is mapped to the element $fun(s)$, where *fun* is a GAP function.

If the argument *invfun* is bound then map is a bijection between *S* and *R*, and the preimage of each element r of *R* is given by $invfun(r)$, where *invfun* is a GAP function.

If five arguments are given and the fourth argument is false then the GAP function *prefun* can be used to compute a single preimage also if map is not bijective.

The mapping returned by MappingByFunction lies in the filter IsNonSPGeneralMapping (32.14.1), see 32.14.

32.2.3 InverseGeneralMapping

▷ InverseGeneralMapping(*map*) (attribute)

The *inverse general mapping* of a general mapping *map* is the general mapping whose underlying relation (see UnderlyingRelation (32.3.9)) contains a pair (r,s) if and only if the underlying relation of *map* contains the pair (s,r).

See the introduction to Chapter 32 for the subtleties concerning the difference between InverseGeneralMapping and Inverse (31.10.8).

Note that the inverse general mapping of a mapping *map* is in general only a general mapping. If *map* knows to be bijective its inverse general mapping will know to be a mapping. In this case also Inverse(*map*) works.

32.2.4 CompositionMapping

▷ CompositionMapping(*map1, map2, ...*) (function)

CompositionMapping allows one to compose arbitrarily many general mappings, and delegates each step to CompositionMapping2 (32.2.5).

Additionally, the properties IsInjective (32.3.4) and IsSingleValued (32.3.2) are maintained; if the source of the $i+1$-th general mapping is identical to the range of the i-th general mapping, also IsTotal (32.3.1) and IsSurjective (32.3.5) are maintained. (So one should not call CompositionMapping2 (32.2.5) directly if one wants to maintain these properties.)

Depending on the types of *map1* and *map2*, the returned mapping might be constructed completely new (for example by giving domain generators and their images, this is for example the case if both mappings preserve the same algebraic structures and GAP can decompose elements of the source of *map2* into generators) or as an (iterated) composition (see IsCompositionMappingRep (32.2.6)).

32.2.5 CompositionMapping2

▷ CompositionMapping2(*map2, map1*) (operation)
▷ CompositionMapping2General(*map2, map1*) (function)

CompositionMapping2 returns the composition of *map2* and *map1*, this is the general mapping that maps an element first under *map1*, and then maps the images under *map2*.

(Note the reverse ordering of arguments in the composition via the multiplication * (31.12.1).

CompositionMapping2General is the method that forms a composite mapping with two constituent mappings. (This is used in some algorithms.)

32.2.6 IsCompositionMappingRep

▷ IsCompositionMappingRep(*map*) (Representation)

Mappings in this representation are stored as composition of two mappings, (pre)images of elements are computed in a two-step process. The constituent mappings of the composition can be obtained via ConstituentsCompositionMapping (32.2.7).

32.2.7 ConstituentsCompositionMapping

▷ ConstituentsCompositionMapping(*map*) (function)

If *map* is stored in the representation IsCompositionMappingRep (32.2.6) as composition of two mappings *map1* and *map2*, this function returns the two constituent mappings in a list [*map1*, *map2*].

32.2.8 ZeroMapping

▷ ZeroMapping(*S, R*) (operation)

A zero mapping is a total general mapping that maps each element of its source to the zero element of its range.

(Each mapping with empty source is a zero mapping.)

32.2.9 IdentityMapping

▷ `IdentityMapping(D)` (attribute)

is the bijective mapping with source and range equal to the collection `D`, which maps each element of `D` to itself.

32.2.10 Embedding

▷ `Embedding(S, T)` (operation)
▷ `Embedding(S, i)` (operation)

returns the embedding of the domain `S` in the domain `T`, or in the second form, some domain indexed by the positive integer `i`. The precise natures of the various methods are described elsewhere: for Lie algebras, see `LieFamily` (64.1.3); for group products, see 49.6 for a general description, or for examples see 49.1 for direct products, 49.2 for semidirect products, or 49.4 for wreath products; or for magma rings see 65.3.

32.2.11 Projection

▷ `Projection(S, T)` (operation)
▷ `Projection(S, i)` (operation)
▷ `Projection(S)` (operation)

returns the projection of the domain `S` onto the domain `T`, or in the second form, some domain indexed by the positive integer `i`, or in the third form some natural quotient domain of `S`. Various methods are defined for group products; see 49.6 for a general description, or for examples see 49.1 for direct products, 49.2 for semidirect products, 49.3 for subdirect products, or 49.4 for wreath products.

32.2.12 RestrictedMapping

▷ `RestrictedMapping(map, subdom)` (operation)

If *subdom* is a subdomain of the source of the general mapping *map*, this operation returns the restriction of *map* to *subdom*.

32.3 Properties and Attributes of (General) Mappings

32.3.1 IsTotal

▷ `IsTotal(map)` (property)

is `true` if each element in the source S of the general mapping *map* has images, i.e., $s^{map} \neq \emptyset$ for all $s \in S$, and `false` otherwise.

32.3.2 IsSingleValued

▷ `IsSingleValued(`*map*`)` (property)

is `true` if each element in the source S of the general mapping *map* has at most one image, i.e., $|s^{map}| \leq 1$ for all $s \in S$, and `false` otherwise.

Equivalently, `IsSingleValued( `*map*` )` is `true` if and only if the preimages of different elements in R are disjoint.

32.3.3 IsMapping

▷ `IsMapping(`*map*`)` (property)

A *mapping* *map* is a general mapping that assigns to each element `elm` of its source a unique element `Image( `*map*`, elm )` of its range.

Equivalently, the general mapping *map* is a mapping if and only if it is total and single-valued (see `IsTotal` (32.3.1), `IsSingleValued` (32.3.2)).

32.3.4 IsInjective

▷ `IsInjective(`*map*`)` (property)

is `true` if the images of different elements in the source S of the general mapping *map* are disjoint, i.e., $x^{map} \cap y^{map} = \emptyset$ for $x \neq y \in S$, and `false` otherwise.

Equivalently, `IsInjective( `*map*` )` is `true` if and only if each element in the range of *map* has at most one preimage in S.

32.3.5 IsSurjective

▷ `IsSurjective(`*map*`)` (property)

is `true` if each element in the range R of the general mapping *map* has preimages in the source S of *map*, i.e., $\{s \in S \mid x \in s^{map}\} \neq \emptyset$ for all $x \in R$, and `false` otherwise.

32.3.6 IsBijective

▷ `IsBijective(`*map*`)` (property)

A general mapping *map* is *bijective* if and only if it is an injective and surjective mapping (see `IsMapping` (32.3.3), `IsInjective` (32.3.4), `IsSurjective` (32.3.5)).

32.3.7 Range (of a general mapping)

▷ `Range(`*map*`)` (attribute)

The range of a general mapping.

32.3.8 Source

▷ `Source(map)` (attribute)

The source of a general mapping.

32.3.9 UnderlyingRelation

▷ `UnderlyingRelation(map)` (attribute)

The *underlying relation* of a general mapping `map` is the domain of pairs (s, r), with s in the source and r in the range of `map` (see `Source` (32.3.8), `Range` (32.3.7)), and $r \in$ `ImagesElm( map, s )`.

Each element of the underlying relation is represented by a direct product element (see `IsDirectProductElement` (32.1.1)).

32.3.10 UnderlyingGeneralMapping

▷ `UnderlyingGeneralMapping(map)` (attribute)

attribute for underlying relations of general mappings

32.4 Images under Mappings

32.4.1 ImagesSource

▷ `ImagesSource(map)` (attribute)

is the set of images of the source of the general mapping `map`.

`ImagesSource` delegates to `ImagesSet` (32.4.4), it is introduced only to store the image of `map` as attribute value.

32.4.2 ImagesRepresentative

▷ `ImagesRepresentative(map, elm)` (operation)

If `elm` is an element of the source of the general mapping `map` then `ImagesRepresentative` returns either a representative of the set of images of `elm` under `map` or `fail`, the latter if and only if `elm` has no images under `map`.

Anything may happen if `elm` is not an element of the source of `map`.

32.4.3 ImagesElm

▷ `ImagesElm(map, elm)` (operation)

If `elm` is an element of the source of the general mapping `map` then `ImagesElm` returns the set of all images of `elm` under `map`.

Anything may happen if `elm` is not an element of the source of `map`.

32.4.4 ImagesSet

▷ ImagesSet(*map, elms*) (operation)

If *elms* is a subset of the source of the general mapping *map* then ImagesSet returns the set of all images of *elms* under *map*.

The result will be either a proper set or a domain. Anything may happen if *elms* is not a subset of the source of *map*.

32.4.5 ImageElm

▷ ImageElm(*map, elm*) (operation)

If *elm* is an element of the source of the total and single-valued mapping *map* then ImageElm returns the unique image of *elm* under *map*.

Anything may happen if *elm* is not an element of the source of *map*.

32.4.6 Image

▷ Image(*map*) (function)
▷ Image(*map, elm*) (function)
▷ Image(*map, coll*) (function)

Image(*map*) is the *image* of the general mapping *map*, i.e., the subset of elements of the range of *map* that are actually values of *map*. *Note* that in this case the argument may also be multi-valued.

Image(*map*, *elm*) is the image of the element *elm* of the source of the mapping *map* under *map*, i.e., the unique element of the range to which *map* maps *elm*. This can also be expressed as *elm*^*map*. Note that *map* must be total and single valued, a multi valued general mapping is not allowed (see Images (32.4.7)).

Image(*map*, *coll*) is the image of the subset *coll* of the source of the mapping *map* under *map*, i.e., the subset of the range to which *map* maps elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. Note that in this case *map* may also be multi-valued. (If *coll* and the result are lists then the positions of entries do in general *not* correspond.)

Image delegates to ImagesSource (32.4.1) when called with one argument, and to ImageElm (32.4.5) resp. ImagesSet (32.4.4) when called with two arguments.

If the second argument is not an element or a subset of the source of the first argument, an error is signalled.

32.4.7 Images

▷ Images(*map*) (function)
▷ Images(*map, elm*) (function)
▷ Images(*map, coll*) (function)

Images(*map*) is the *image* of the general mapping *map*, i.e., the subset of elements of the range of *map* that are actually values of *map*.

`Images( map, elm )` is the set of images of the element *elm* of the source of the general mapping *map* under *map*, i.e., the set of elements of the range to which *map* maps *elm*.

`Images( map, coll )` is the set of images of the subset *coll* of the source of the general mapping *map* under *map*, i.e., the subset of the range to which *map* maps elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. (If *coll* and the result are lists then the positions of entries do in general *not* correspond.)

`Images` delegates to `ImagesSource` (32.4.1) when called with one argument, and to `ImagesElm` (32.4.3) resp. `ImagesSet` (32.4.4) when called with two arguments.

If the second argument is not an element or a subset of the source of the first argument, an error is signalled.

32.5 Preimages under Mappings

32.5.1 PreImagesRange

▷ `PreImagesRange(`*map*`)` (attribute)

is the set of preimages of the range of the general mapping *map*.

`PreImagesRange` delegates to `PreImagesSet` (32.5.5), it is introduced only to store the preimage of *map* as attribute value.

32.5.2 PreImagesElm

▷ `PreImagesElm(`*map, elm*`)` (operation)

If *elm* is an element of the range of the general mapping *map* then `PreImagesElm` returns the set of all preimages of *elm* under *map*.

Anything may happen if *elm* is not an element of the range of *map*.

32.5.3 PreImageElm

▷ `PreImageElm(`*map, elm*`)` (operation)

If *elm* is an element of the range of the injective and surjective general mapping *map* then `PreImageElm` returns the unique preimage of *elm* under *map*.

Anything may happen if *elm* is not an element of the range of *map*.

32.5.4 PreImagesRepresentative

▷ `PreImagesRepresentative(`*map, elm*`)` (operation)

If *elm* is an element of the range of the general mapping *map* then `PreImagesRepresentative` returns either a representative of the set of preimages of *elm* under *map* or `fail`, the latter if and only if *elm* has no preimages under *map*.

Anything may happen if *elm* is not an element of the range of *map*.

32.5.5 PreImagesSet

▷ PreImagesSet(*map, elms*) (operation)

If *elms* is a subset of the range of the general mapping *map* then PreImagesSet returns the set of all preimages of *elms* under *map*.

Anything may happen if *elms* is not a subset of the range of *map*.

32.5.6 PreImage

▷ PreImage(*map*) (function)
▷ PreImage(*map, elm*) (function)
▷ PreImage(*map, coll*) (function)

PreImage(*map*) is the preimage of the general mapping *map*, i.e., the subset of elements of the source of *map* that actually have values under *map*. Note that in this case the argument may also be non-injective or non-surjective.

PreImage(*map*, *elm*) is the preimage of the element *elm* of the range of the injective and surjective mapping *map* under *map*, i.e., the unique element of the source which is mapped under *map* to *elm*. Note that *map* must be injective and surjective (see PreImages (32.5.7)).

PreImage(*map*, *coll*) is the preimage of the subset *coll* of the range of the general mapping *map* under *map*, i.e., the subset of the source which is mapped under *map* to elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. Note that in this case *map* may also be non-injective or non-surjective. (If *coll* and the result are lists then the positions of entries do in general *not* correspond.)

PreImage delegates to PreImagesRange (32.5.1) when called with one argument, and to PreImageElm (32.5.3) resp. PreImagesSet (32.5.5) when called with two arguments.

If the second argument is not an element or a subset of the range of the first argument, an error is signalled.

32.5.7 PreImages

▷ PreImages(*map*) (function)
▷ PreImages(*map, elm*) (function)
▷ PreImages(*map, coll*) (function)

PreImages(*map*) is the preimage of the general mapping *map*, i.e., the subset of elements of the source of *map* that have actually values under *map*.

PreImages(*map*, *elm*) is the set of preimages of the element *elm* of the range of the general mapping *map* under *map*, i.e., the set of elements of the source which *map* maps to *elm*.

PreImages(*map*, *coll*) is the set of images of the subset *coll* of the range of the general mapping *map* under *map*, i.e., the subset of the source which *map* maps to elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. (If *coll* and the result are lists then the positions of entries do in general *not* correspond.)

PreImages delegates to PreImagesRange (32.5.1) when called with one argument, and to PreImagesElm (32.5.2) resp. PreImagesSet (32.5.5) when called with two arguments.

If the second argument is not an element or a subset of the range of the first argument, an error is signalled.

32.6 Arithmetic Operations for General Mappings

General mappings are arithmetic objects. One can form groups and vector spaces of general mappings provided that they are invertible or can be added and admit scalar multiplication, respectively.

For two general mappings with same source, range, preimage, and image, the *sum* is defined pointwise, i.e., the images of a point under the sum is the set of all sums with first summand in the images of the first general mapping and second summand in the images of the second general mapping.

Scalar multiplication of general mappings is defined likewise.

The *product* of two general mappings is defined as the composition. This multiplication is always associative. In addition to the composition via `*`, general mappings can be composed –in reversed order– via `CompositionMapping` (32.2.4).

General mappings are in the category of multiplicative elements with inverses. Similar to matrices, not every general mapping has an inverse or an identity, and we define the behaviour of `One` (31.10.2) and `Inverse` (31.10.8) for general mappings as follows. `One` (31.10.2) returns `fail` when called for a general mapping whose source and range differ, otherwise `One` (31.10.2) returns the identity mapping of the source. (Note that the source may differ from the preimage). `Inverse` (31.10.8) returns `fail` when called for a non-bijective general mapping or for a general mapping whose source and range differ; otherwise `Inverse` (31.10.8) returns the inverse mapping.

Besides the usual inverse of multiplicative elements, which means that `Inverse( g ) * g = g * Inverse( g ) = One( g )`, for general mappings we have the attribute `InverseGeneralMapping` (32.2.3). If F is a general mapping with source S, range R, and underlying relation Rel then `InverseGeneralMapping( F )` has source R, range S, and underlying relation $\{(r,s) \mid (s,r) \in Rel\}$. For a general mapping that has an inverse in the usual sense, i.e., for a bijection of the source, of course both concepts coincide.

`Inverse` (31.10.8) may delegate to `InverseGeneralMapping` (32.2.3). `InverseGeneralMapping` (32.2.3) must not delegate to `Inverse` (31.10.8), but a known value of `Inverse` (31.10.8) may be fetched. So methods to compute the inverse of a general mapping should be installed for `InverseGeneralMapping` (32.2.3).

(Note that in many respects, general mappings behave similar to matrices, for example one can define left and right identities and inverses, which do not fit into the current concepts of GAP.)

32.7 Mappings which are Compatible with Algebraic Structures

From an algebraical point of view, the most important mappings are those which are compatible with a structure. For Magmas, Groups and Rings, GAP supports the following four types of such mappings:

1. General mappings that respect multiplication
2. General mappings that respect addition
3. General mappings that respect scalar mult.
4. General mappings that respect multiplicative and additive structure

(Very technical note: GAP defines categories `IsSPGeneralMapping` and `IsNonSPGeneralMapping`. The distinction between these is orthogonal to the structure compatibility described here and should not be confused.)

32.8 Magma Homomorphisms

32.8.1 IsMagmaHomomorphism

▷ IsMagmaHomomorphism(*mapp*) (property)

A *magma homomorphism* is a total single valued mapping which respects multiplication.

32.8.2 MagmaHomomorphismByFunctionNC

▷ MagmaHomomorphismByFunctionNC(*G, H, fn*) (function)

Creates the homomorphism from *G* to *H* without checking that *fn* is a homomorphism.

32.8.3 NaturalHomomorphismByGenerators

▷ NaturalHomomorphismByGenerators(*f, s*) (operation)

returns a mapping from the magma *f* with n generators to the magma *s* with n generators, which maps the i-th generator of *f* to the i-th generator of *s*.

32.9 Mappings that Respect Multiplication

32.9.1 RespectsMultiplication

▷ RespectsMultiplication(*mapp*) (property)

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then RespectsMultiplication returns true if S and R are magmas such that $(s_1, r_1), (s_2, r_2) \in F$ implies $(s_1 * s_2, r_1 * r_2) \in F$, and false otherwise.

If *mapp* is single-valued then RespectsMultiplication returns true if and only if the equation *s1^mapp * s2^mapp = (s1 * s2)^mapp* holds for all *s1*, *s2* in S.

32.9.2 RespectsOne

▷ RespectsOne(*mapp*) (property)

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where *S* and *R* are the source and the range of *mapp*, respectively. Then RespectsOne returns true if *S* and *R* are magmas-with-one such that (One(*S*),One(*R*)) $\in F$, and false otherwise.

If *mapp* is single-valued then RespectsOne returns true if and only if the equation One(*S*)^*mapp* = One(*R*) holds.

32.9.3 RespectsInverses

▷ RespectsInverses(*mapp*) (property)

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where *S* and *R* are the source and the range of *mapp*, respectively. Then `RespectsInverses` returns `true` if *S* and *R* are magmas-with-inverses such that, for $s \in S$ and $r \in R$, $(s,r) \in F$ implies $(s^{-1}, r^{-1}) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsInverses` returns `true` if and only if the equation `Inverse( s )^mapp = Inverse( s^mapp )` holds for all *s* in *S*.

32.9.4 IsGroupGeneralMapping

▷ `IsGroupGeneralMapping(`*mapp*`)` (property)
▷ `IsGroupHomomorphism(`*mapp*`)` (property)

A *group general mapping* is a mapping which respects multiplication and inverses. If it is total and single valued it is called a *group homomorphism*.

Chapter 40 explains group homomorphisms in more detail.

32.9.5 KernelOfMultiplicativeGeneralMapping

▷ `KernelOfMultiplicativeGeneralMapping(`*mapp*`)` (attribute)

Let *mapp* be a general mapping. Then `KernelOfMultiplicativeGeneralMapping` returns the set of all elements in the source of *mapp* that have the identity of the range in their set of images.

(This is a monoid if *mapp* respects multiplication and one, and if the source of *mapp* is associative.)

32.9.6 CoKernelOfMultiplicativeGeneralMapping

▷ `CoKernelOfMultiplicativeGeneralMapping(`*mapp*`)` (attribute)

Let *mapp* be a general mapping. Then `CoKernelOfMultiplicativeGeneralMapping` returns the set of all elements in the range of *mapp* that have the identity of the source in their set of preimages.

(This is a monoid if *mapp* respects multiplication and one, and if the range of *mapp* is associative.)

32.10 Mappings that Respect Addition

32.10.1 RespectsAddition

▷ `RespectsAddition(`*mapp*`)` (property)

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsAddition` returns `true` if S and R are additive magmas such that $(s_1, r_1), (s_2, r_2) \in F$ implies $(s_1 + s_2, r_1 + r_2) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsAddition` returns `true` if and only if the equation `s1^mapp + s2^mapp = (s1+s2)^mapp` holds for all *s1*, *s2* in *S*.

32.10.2 RespectsAdditiveInverses

▷ `RespectsAdditiveInverses(`*mapp*`)` (property)

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then RespectsAdditiveInverses returns true if S and R are additive-magmas-with-inverses such that $(s,r) \in F$ implies $(-s,-r) \in F$, and false otherwise.

If *mapp* is single-valued then RespectsAdditiveInverses returns true if and only if the equation AdditiveInverse(*s*)^*mapp* = AdditiveInverse(*s*^*mapp*) holds for all *s* in S.

32.10.3 RespectsZero

▷ RespectsZero(*mapp*) (property)

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where *S* and *R* are the source and the range of *mapp*, respectively. Then RespectsZero returns true if *S* and *R* are additive-magmas-with-zero such that (Zero(*S*),Zero(*R*)) $\in F$, and false otherwise.

If *mapp* is single-valued then RespectsZero returns true if and only if the equation Zero(*S*)^*mapp* = Zero(*R*) holds.

32.10.4 IsAdditiveGroupGeneralMapping

▷ IsAdditiveGroupGeneralMapping(*mapp*) (property)
▷ IsAdditiveGroupHomomorphism(*mapp*) (property)

IsAdditiveGroupGeneralMapping specifies whether a general mapping *mapp* respects addition (see RespectsAddition (32.10.1)) and respects additive inverses (see RespectsAdditiveInverses (32.10.2)).

IsAdditiveGroupHomomorphism is a synonym for the meet of IsAdditiveGroupGeneralMapping and IsMapping (32.3.3).

32.10.5 KernelOfAdditiveGeneralMapping

▷ KernelOfAdditiveGeneralMapping(*mapp*) (attribute)

Let *mapp* be a general mapping. Then KernelOfAdditiveGeneralMapping returns the set of all elements in the source of *mapp* that have the zero of the range in their set of images.

32.10.6 CoKernelOfAdditiveGeneralMapping

▷ CoKernelOfAdditiveGeneralMapping(*mapp*) (attribute)

Let *mapp* be a general mapping. Then CoKernelOfAdditiveGeneralMapping returns the set of all elements in the range of *mapp* that have the zero of the source in their set of preimages.

32.11 Linear Mappings

Also see Sections 32.9, 32.10, and KernelOfMultiplicativeGeneralMapping (32.9.5), CoKernelOfMultiplicativeGeneralMapping (32.9.6).

32.11.1 RespectsScalarMultiplication

▷ `RespectsScalarMultiplication(`*mapp*`)` (property)

Let *mapp* be a general mapping, with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsScalarMultiplication` returns `true` if S and R are left modules with the left acting domain D of S contained in the left acting domain of R and such that $(s,r) \in F$ implies $(c*s, c*r) \in F$ for all $c \in D$, and `false` otherwise.

If *mapp* is single-valued then `RespectsScalarMultiplication` returns `true` if and only if the equation `c * s^mapp = (c * s)^mapp` holds for all `c` in D and `s` in S.

32.11.2 IsLeftModuleGeneralMapping

▷ `IsLeftModuleGeneralMapping(`*mapp*`)` (property)
▷ `IsLeftModuleHomomorphism(`*mapp*`)` (property)

`IsLeftModuleGeneralMapping` specifies whether a general mapping *mapp* satisfies the property `IsAdditiveGroupGeneralMapping` (32.10.4) and respects scalar multiplication (see `RespectsScalarMultiplication` (32.11.1)).

`IsLeftModuleHomomorphism` is a synonym for the meet of `IsLeftModuleGeneralMapping` and `IsMapping` (32.3.3).

32.11.3 IsLinearMapping

▷ `IsLinearMapping(`*F*`, `*mapp*`)` (operation)

For a field *F* and a general mapping *mapp*, `IsLinearMapping` returns `true` if *mapp* is an *F*-linear mapping, and `false` otherwise.

A mapping f is a linear mapping (or vector space homomorphism) if the source and range are vector spaces over the same division ring D, and if $f(a+b) = f(a) + f(b)$ and $f(s*a) = s*f(a)$ hold for all elements a, b in the source of f and $s \in D$.

32.12 Ring Homomorphisms

32.12.1 IsRingGeneralMapping

▷ `IsRingGeneralMapping(`*mapp*`)` (property)
▷ `IsRingHomomorphism(`*mapp*`)` (property)

`IsRingGeneralMapping` specifies whether a general mapping *mapp* satisfies the property `IsAdditiveGroupGeneralMapping` (32.10.4) and respects multiplication (see `RespectsMultiplication` (32.9.1)).

`IsRingHomomorphism` is a synonym for the meet of `IsRingGeneralMapping` and `IsMapping` (32.3.3).

32.12.2 IsRingWithOneGeneralMapping

▷ IsRingWithOneGeneralMapping(*mapp*) (property)
▷ IsRingWithOneHomomorphism(*mapp*) (property)

32.12.3 IsAlgebraGeneralMapping

▷ IsAlgebraGeneralMapping(*mapp*) (property)
▷ IsAlgebraHomomorphism(*mapp*) (property)

IsAlgebraGeneralMapping specifies whether a general mapping *mapp* satisfies both properties IsRingGeneralMapping (32.12.1) and (see IsLeftModuleGeneralMapping (32.11.2)).

IsAlgebraHomomorphism is a synonym for the meet of IsAlgebraGeneralMapping and IsMapping (32.3.3).

32.12.4 IsAlgebraWithOneGeneralMapping

▷ IsAlgebraWithOneGeneralMapping(*mapp*) (property)
▷ IsAlgebraWithOneHomomorphism(*mapp*) (property)

IsAlgebraWithOneGeneralMapping specifies whether a general mapping *mapp* satisfies both properties IsAlgebraGeneralMapping (32.12.3) and RespectsOne (32.9.2).

IsAlgebraWithOneHomomorphism is a synonym for the meet of IsAlgebraWithOneGeneralMapping and IsMapping (32.3.3).

32.12.5 IsFieldHomomorphism

▷ IsFieldHomomorphism(*mapp*) (property)

A general mapping is a field homomorphism if and only if it is a ring homomorphism with source a field.

32.13 General Mappings

32.13.1 IsGeneralMapping

▷ IsGeneralMapping(*map*) (Category)

Each general mapping lies in the category IsGeneralMapping. It implies the categories IsMultiplicativeElementWithInverse (31.14.13) and IsAssociativeElement (31.15.1); for a discussion of these implications, see 32.6.

32.13.2 IsConstantTimeAccessGeneralMapping

▷ IsConstantTimeAccessGeneralMapping(*map*) (property)

is `true` if the underlying relation of the general mapping *map* knows its `AsList` (30.3.8) value, and `false` otherwise.

In the former case, *map* is allowed to use this list for calls to `ImagesElm` (32.4.3) etc.

32.13.3 IsEndoGeneralMapping

▷ `IsEndoGeneralMapping(`*obj*`)` (property)

If a general mapping has this property then its source and range are equal.

32.14 Technical Matters Concerning General Mappings

`Source` (32.3.8) and `Range` (32.3.7) are basic operations for general mappings. `UnderlyingRelation` (32.3.9) is secondary, its default method sets up a domain that delegates tasks to the general mapping. (Note that this allows one to handle also infinite relations by generic methods if source or range of the general mapping is finite.)

The distinction between basic operations and secondary operations for general mappings may be a little bit complicated. Namely, each general mapping must be in one of the two categories `IsNonSPGeneralMapping` (32.14.1), `IsSPGeneralMapping` (32.14.1). (The category `IsGeneralMapping` (32.13.1) is defined as the disjoint union of these two.)

For general mappings of the first category, `ImagesElm` (32.4.3) and `PreImagesElm` (32.5.2) are basic operations. (Note that in principle it is possible to delegate from `PreImagesElm` (32.5.2) to `ImagesElm` (32.4.3).) Methods for the secondary operations `ImageElm` (32.4.5), `PreImageElm` (32.5.3), `ImagesSet` (32.4.4), `PreImagesSet` (32.5.5), `ImagesRepresentative` (32.4.2), and `PreImagesRepresentative` (32.5.4) may use `ImagesElm` (32.4.3) and `PreImagesElm` (32.5.2), respectively, and methods for `ImagesElm` (32.4.3), `PreImagesElm` (32.5.2) must *not* call the secondary operations. In particular, there are no generic methods for `ImagesElm` (32.4.3) and `PreImagesElm` (32.5.2).

Methods for `ImagesSet` (32.4.4) and `PreImagesSet` (32.5.5) must *not* use `PreImagesRange` (32.5.1) and `ImagesSource` (32.4.1), e.g., compute the intersection of the set in question with the preimage of the range resp. the image of the source.

For general mappings of the second category (which means structure preserving general mappings), the situation is different. The set of preimages under a group homomorphism, for example, is either empty or can be described as a coset of the (multiplicative) kernel. So it is reasonable to have `ImagesRepresentative` (32.4.2), `PreImagesRepresentative` (32.5.4), `KernelOfMultiplicativeGeneralMapping` (32.9.5), and `CoKernelOfMultiplicativeGeneralMapping` (32.9.6) as basic operations here, and to make `ImagesElm` (32.4.3) and `PreImagesElm` (32.5.2) secondary operations that may delegate to these.

In order to avoid infinite recursions, we must distinguish between the two different types of mappings.

(Note that the basic domain operations such as `AsList` (30.3.8) for the underlying relation of a general mapping may use either `ImagesElm` (32.4.3) or `ImagesRepresentative` (32.4.2) and the appropriate cokernel. Conversely, if `AsList` (30.3.8) for the underlying relation is known then `ImagesElm` (32.4.3) resp. `ImagesRepresentative` (32.4.2) may delegate to it, the general mapping gets the property `IsConstantTimeAccessGeneralMapping` (32.13.2) for this; note that this is not allowed if only an enumerator of the underlying relation is known.)

Secondary operations are IsInjective (32.3.4), IsSingleValued (32.3.2), IsSurjective (32.3.5), IsTotal (32.3.1); they may use the basic operations, and must not be used by them.

Methods for the operations ImagesElm (32.4.3), ImagesRepresentative (32.4.2), ImagesSet (32.4.4), ImageElm (32.4.5), PreImagesElm (32.5.2), PreImagesRepresentative (32.5.4), PreImagesSet (32.5.5), and PreImageElm (32.5.3) take two arguments, a general mapping *map* and an element or collection of elements *elm*. These methods must *not* check whether *elm* lies in the source or the range of *map*. In the case that *elm* does not, fail may be returned as well as any other GAP object, and even an error message is allowed. Checks of the arguments are done only by the functions Image (32.4.6), Images (32.4.7), PreImage (32.5.6), and PreImages (32.5.7), which then delegate to the operations listed above.

32.14.1 IsSPGeneralMapping

▷ IsSPGeneralMapping(*map*) (Category)
▷ IsNonSPGeneralMapping(*map*) (Category)

32.14.2 IsGeneralMappingFamily

▷ IsGeneralMappingFamily(*obj*) (Category)

The family category of the category of general mappings.

32.14.3 FamilyRange

▷ FamilyRange(*Fam*) (attribute)

is the elements family of the family of the range of each general mapping in the family *Fam*.

32.14.4 FamilySource

▷ FamilySource(*Fam*) (attribute)

is the elements family of the family of the source of each general mapping in the family *Fam*.

32.14.5 FamiliesOfGeneralMappingsAndRanges

▷ FamiliesOfGeneralMappingsAndRanges(*Fam*) (attribute)

is a list that stores at the odd positions the families of general mappings with source in the family *Fam*, at the even positions the families of ranges of the general mappings.

32.14.6 GeneralMappingsFamily

▷ GeneralMappingsFamily(*sourcefam, rangefam*) (function)

All general mappings with same source family *FS* and same range family *FR* lie in the family GeneralMappingsFamily(*FS*, *FR*).

32.14.7 TypeOfDefaultGeneralMapping

▷ `TypeOfDefaultGeneralMapping(`*`source, range, filter`*`)` (function)

is the type of mappings with `IsDefaultGeneralMappingRep` with source *`source`* and range *`range`* and additional categories *`filter`*.

Chapter 33

Relations

A *binary relation* R on a set X is a subset of $X \times X$. A binary relation can also be thought of as a (general) mapping from X to itself or as a directed graph where each edge represents an element of R.

In GAP, a relation is conceptually represented as a general mapping from X to itself. The category `IsBinaryRelation` (33.1.1) is a synonym for `IsEndoGeneralMapping` (32.13.3). Attributes and properties of relations in GAP are supported for relations, via considering relations as a subset of $X \times X$, or as a directed graph; examples include finding the strongly connected components of a relation, via `StronglyConnectedComponents` (33.4.5), or enumerating the tuples of the relation.

33.1 General Binary Relations

This section lists general constructors of relations.

33.1.1 IsBinaryRelation

▷ `IsBinaryRelation(`*R*`)` (Category)

is exactly the same category as (i.e. a synonym for) `IsEndoGeneralMapping` (32.13.3).

33.1.2 BinaryRelationByElements

▷ `BinaryRelationByElements(`*domain, elms*`)` (function)

is the binary relation on *domain* and with underlying relation consisting of the tuples collection *elms*. This construction is similar to `GeneralMappingByElements` (32.2.1) where the source and range are the same set.

33.1.3 IdentityBinaryRelation

▷ `IdentityBinaryRelation(`*degree*`)` (function)
▷ `IdentityBinaryRelation(`*domain*`)` (function)

is the binary relation which consists of diagonal pairs, i.e., pairs of the form (x,x). In the first form if a positive integer *degree* is given then the domain is the set of the integers $\{1,\ldots,$ *degree*$\}$. In the second form, the objects x are from the domain *domain*.

33.1.4 EmptyBinaryRelation (for a degree)

▷ `EmptyBinaryRelation(`*degree*`)` (function)
▷ `EmptyBinaryRelation(`*domain*`)` (function)

is the relation with `R` empty. In the first form of the command with *degree* an integer, the domain is the set of points $\{1,\ldots,$*degree*$\}$. In the second form, the domain is that given by the argument *domain*.

33.2 Properties and Attributes of Binary Relations

33.2.1 IsReflexiveBinaryRelation

▷ `IsReflexiveBinaryRelation(R)` (property)

returns `true` if the binary relation `R` is reflexive, and `false` otherwise.

A binary relation R (as a set of pairs) on a set X is *reflexive* if for all $x \in X$, $(x,x) \in R$. Alternatively, R as a mapping is reflexive if for all $x \in X$, x is an element of the image set $R(x)$.

A reflexive binary relation is necessarily a total endomorphic mapping (tested via `IsTotal` (32.3.1)).

33.2.2 IsSymmetricBinaryRelation

▷ `IsSymmetricBinaryRelation(R)` (property)

returns `true` if the binary relation `R` is symmetric, and `false` otherwise.

A binary relation R (as a set of pairs) on a set X is *symmetric* if $(x,y) \in R$ then $(y,x) \in R$. Alternatively, R as a mapping is symmetric if for all $x \in X$, the preimage set of x under R equals the image set $R(x)$.

33.2.3 IsTransitiveBinaryRelation

▷ `IsTransitiveBinaryRelation(R)` (property)

returns `true` if the binary relation `R` is transitive, and `false` otherwise.

A binary relation `R` (as a set of pairs) on a set X is *transitive* if $(x,y),(y,z) \in R$ implies $(x,z) \in R$. Alternatively, R as a mapping is transitive if for all $x \in X$, the image set $R(R(x))$ of the image set $R(x)$ of x is a subset of $R(x)$.

33.2.4 IsAntisymmetricBinaryRelation

▷ `IsAntisymmetricBinaryRelation(rel)` (property)

returns `true` if the binary relation *rel* is antisymmetric, and `false` otherwise.

A binary relation `R` (as a set of pairs) on a set X is *antisymmetric* if $(x,y),(y,x) \in R$ implies $x = y$. Alternatively, R as a mapping is antisymmetric if for all $x \in X$, the intersection of the preimage set of x under R and the image set $R(x)$ is $\{x\}$.

33.2.5 IsPreOrderBinaryRelation

▷ IsPreOrderBinaryRelation(*rel*) (property)

returns true if the binary relation *rel* is a preorder, and false otherwise.
A *preorder* is a binary relation that is both reflexive and transitive.

33.2.6 IsPartialOrderBinaryRelation

▷ IsPartialOrderBinaryRelation(*rel*) (property)

returns true if the binary relation *rel* is a partial order, and false otherwise.
A *partial order* is a preorder which is also antisymmetric.

33.2.7 IsHasseDiagram

▷ IsHasseDiagram(*rel*) (property)

returns true if the binary relation *rel* is a Hasse Diagram of a partial order, i.e., was computed via HasseDiagramBinaryRelation (33.4.4).

33.2.8 IsEquivalenceRelation

▷ IsEquivalenceRelation(*R*) (property)

returns true if the binary relation *R* is an equivalence relation, and false otherwise.
Recall, that a relation *R* is an *equivalence relation* if it is symmetric, transitive, and reflexive.

33.2.9 Successors

▷ Successors(*R*) (attribute)

returns the list of images of a binary relation *R*. If the underlying domain of the relation is not $\{1,\ldots,n\}$, for some positive integer n, then an error is signalled.

The returned value of Successors is a list of lists where the lists are ordered as the elements according to the sorted order of the underlying set of *R*. Each list consists of the images of the element whose index is the same as the list with the underlying set in sorted order.

The Successors of a relation is the adjacency list representation of the relation.

33.2.10 DegreeOfBinaryRelation

▷ DegreeOfBinaryRelation(*R*) (attribute)

returns the size of the underlying domain of the binary relation *R*. This is most natural when working with a binary relation on points.

33.2.11 PartialOrderOfHasseDiagram

▷ PartialOrderOfHasseDiagram(*HD*) (attribute)

is the partial order associated with the Hasse Diagram *HD* i.e. the partial order generated by the reflexive and transitive closure of *HD*.

33.3 Binary Relations on Points

We have special construction methods when the underlying *X* of our relation is the set of integers $\{1,\ldots,n\}$.

33.3.1 BinaryRelationOnPoints

▷ BinaryRelationOnPoints(*list*) (function)
▷ BinaryRelationOnPointsNC(*list*) (function)

Given a list of n lists, each containing elements from the set $\{1,\ldots,n\}$, this function constructs a binary relation such that 1 is related to *list*[1], 2 to *list*[2] and so on. The first version checks whether the list supplied is valid. The the NC version skips this check.

33.3.2 RandomBinaryRelationOnPoints

▷ RandomBinaryRelationOnPoints(*degree*) (function)

creates a relation on points with degree *degree*.

33.3.3 AsBinaryRelationOnPoints

▷ AsBinaryRelationOnPoints(*trans*) (function)
▷ AsBinaryRelationOnPoints(*perm*) (function)
▷ AsBinaryRelationOnPoints(*rel*) (function)

return the relation on points represented by general relation *rel*, transformation *trans* or permutation *perm*. If *rel* is already a binary relation on points then *rel* is returned.

Transformations and permutations are special general endomorphic mappings and have a natural representation as a binary relation on points.

In the last form, an isomorphic relation on points is constructed where the points are indices of the elements of the underlying domain in sorted order.

33.4 Closure Operations and Other Constructors

33.4.1 ReflexiveClosureBinaryRelation

▷ ReflexiveClosureBinaryRelation(*R*) (operation)

is the smallest binary relation containing the binary relation `R` which is reflexive. This closure inherits the properties symmetric and transitive from `R`. E.g., if `R` is symmetric then its reflexive closure is also.

33.4.2 SymmetricClosureBinaryRelation

▷ `SymmetricClosureBinaryRelation(R)` (operation)

is the smallest binary relation containing the binary relation `R` which is symmetric. This closure inherits the properties reflexive and transitive from `R`. E.g., if `R` is reflexive then its symmetric closure is also.

33.4.3 TransitiveClosureBinaryRelation

▷ `TransitiveClosureBinaryRelation(rel)` (operation)

is the smallest binary relation containing the binary relation `R` which is transitive. This closure inherits the properties reflexive and symmetric from `R`. E.g., if `R` is symmetric then its transitive closure is also.

`TransitiveClosureBinaryRelation` is a modified version of the Floyd-Warshall method of solving the all-pairs shortest-paths problem on a directed graph. Its asymptotic runtime is $O(n^3)$ where n is the size of the vertex set. It only assumes there is an arbitrary (but fixed) ordering of the vertex set.

33.4.4 HasseDiagramBinaryRelation

▷ `HasseDiagramBinaryRelation(partial-order)` (operation)

is the smallest relation contained in the partial order *partial-order* whose reflexive and transitive closure is equal to *partial-order*.

33.4.5 StronglyConnectedComponents

▷ `StronglyConnectedComponents(R)` (operation)

returns an equivalence relation on the vertices of the binary relation `R`.

33.4.6 PartialOrderByOrderingFunction

▷ `PartialOrderByOrderingFunction(dom, orderfunc)` (function)

constructs a partial order whose elements are from the domain *dom* and are ordered using the ordering function *orderfunc*. The ordering function must be a binary function returning a boolean value. If the ordering function does not describe a partial order then `fail` is returned.

33.5 Equivalence Relations

An *equivalence relation* `E` over the set `X` is a relation on `X` which is reflexive, symmetric, and transitive. A *partition* `P` is a set of subsets of `X` such that for all $R, S \in P$, $R \cap S$ is the empty set and $\cup P = X$. An equivalence relation induces a partition such that if $(x, y) \in E$ then x, y are in the same element of `P`.

Like all binary relations in GAP equivalence relations are regarded as general endomorphic mappings (and the operations, properties and attributes of general mappings are available). However, partitions provide an efficient way of representing equivalence relations. Moreover, only the non-singleton classes or blocks are listed allowing for small equivalence relations to be represented on infinite sets. Hence the main attribute of equivalence relations is `EquivalenceRelationPartition` (33.6.1) which provides the partition induced by the given equivalence.

33.5.1 EquivalenceRelationByPartition

▷ `EquivalenceRelationByPartition(`*`domain, list`*`)` (function)
▷ `EquivalenceRelationByPartitionNC(`*`domain, list`*`)` (function)

constructs the equivalence relation over the set *`domain`* which induces the partition represented by *`list`*. This representation includes only the non-trivial blocks (or equivalent classes). *`list`* is a list of lists, each of these lists contain elements of *`domain`* and are pairwise mutually exclusive.

The list of lists do not need to be in any order nor do the elements in the blocks (see `EquivalenceRelationPartition` (33.6.1)). a list of elements of *`domain`* The partition *`list`* is a list of lists, each of these is a list of elements of *`domain`* that makes up a block (or equivalent class). The *`domain`* is the domain over which the relation is defined, and *`list`* is a list of lists, each of these is a list of elements of *`domain`* which are related to each other. *`list`* need only contain the nontrivial blocks and singletons will be ignored. The `NC` version will not check to see if the lists are pairwise mutually exclusive or that they contain only elements of the domain.

33.5.2 EquivalenceRelationByRelation

▷ `EquivalenceRelationByRelation(`*`rel`*`)` (function)

returns the smallest equivalence relation containing the binary relation *`rel`*.

33.5.3 EquivalenceRelationByPairs

▷ `EquivalenceRelationByPairs(`*`D, elms`*`)` (function)
▷ `EquivalenceRelationByPairsNC(`*`D, elms`*`)` (function)

return the smallest equivalence relation on the domain *`D`* such that every pair in *`elms`* is in the relation.

In the `NC` form, it is not checked that *`elms`* are in the domain *`D`*.

33.5.4 EquivalenceRelationByProperty

▷ `EquivalenceRelationByProperty(`*`domain, property`*`)` (function)

creates an equivalence relation on *domain* whose only defining datum is that of having the property *property*.

33.6 Attributes of and Operations on Equivalence Relations

33.6.1 EquivalenceRelationPartition

▷ EquivalenceRelationPartition(*equiv*) (attribute)

returns a list of lists of elements of the underlying set of the equivalence relation *equiv*. The lists are precisely the nonsingleton equivalence classes of the equivalence. This allows us to describe "small" equivalences on infinite sets.

33.6.2 GeneratorsOfEquivalenceRelationPartition

▷ GeneratorsOfEquivalenceRelationPartition(*equiv*) (attribute)

is a set of generating pairs for the equivalence relation *equiv*. This set is not unique. The equivalence *equiv* is the smallest equivalence relation over the underlying set which contains the generating pairs.

33.6.3 JoinEquivalenceRelations

▷ JoinEquivalenceRelations(*equiv1, equiv2*) (operation)
▷ MeetEquivalenceRelations(*equiv1, equiv2*) (operation)

JoinEquivalenceRelations returns the smallest equivalence relation containing both the equivalence relations *equiv1* and *equiv2*.

MeetEquivalenceRelations returns the intersection of the two equivalence relations *equiv1* and *equiv2*.

33.7 Equivalence Classes

33.7.1 IsEquivalenceClass

▷ IsEquivalenceClass(*obj*) (Category)

returns true if the object *obj* is an equivalence class, and false otherwise.

An *equivalence class* is a collection of elements which are mutually related to each other in the associated equivalence relation. Note, this is a special category of objects and not just a list of elements.

33.7.2 EquivalenceClassRelation

▷ EquivalenceClassRelation(*C*) (attribute)

returns the equivalence relation of which *C* is a class.

33.7.3 EquivalenceClasses (attribute)

▷ `EquivalenceClasses(`*`rel`*`)` (attribute)

returns a list of all equivalence classes of the equivalence relation *rel*. Note that it is possible for different methods to yield the list in different orders, so that for two equivalence relations $c1$ and $c2$ we may have $c1 = c2$ without having `EquivalenceClasses`$(c1)$ =`EquivalenceClasses`$(c2)$.

33.7.4 EquivalenceClassOfElement

▷ `EquivalenceClassOfElement(`*`rel, elt`*`)` (operation)
▷ `EquivalenceClassOfElementNC(`*`rel, elt`*`)` (operation)

return the equivalence class of *elt* in the binary relation *rel*, where *elt* is an element (i.e. a pair) of the domain of *rel*. In the `NC` form, it is not checked that *elt* is in the domain over which *rel* is defined.

Chapter 34

Orderings

In GAP an ordering is a relation defined on a family, which is reflexive, anti-symmetric and transitive.

34.1 IsOrdering (Filter)

34.1.1 IsOrdering

▷ `IsOrdering(`*`obj`*`)` (Category)

returns `true` if and only if the object *`ord`* is an ordering.

34.1.2 OrderingsFamily

▷ `OrderingsFamily(`*`fam`*`)` (attribute)

for a family *`fam`*, returns the family of all orderings on elements of *`fam`*.

34.2 Building new orderings

34.2.1 OrderingByLessThanFunctionNC

▷ `OrderingByLessThanFunctionNC(`*`fam, lt[, l]`*`)` (operation)

Called with two arguments, `OrderingByLessThanFunctionNC` returns the ordering on the elements of the elements of the family *`fam`*, according to the `LessThanFunction` (34.3.5) value given by *`lt`*, where *`lt`* is a function that takes two arguments in *`fam`* and returns `true` or `false`.

Called with three arguments, for a family *`fam`*, a function *`lt`* that takes two arguments in *`fam`* and returns `true` or `false`, and a list *`l`* of properties of orderings, `OrderingByLessThanFunctionNC` returns the ordering on the elements of *`fam`* with `LessThanFunction` (34.3.5) value given by *`lt`* and with the properties from *`l`* set to `true`.

34.2.2 OrderingByLessThanOrEqualFunctionNC

▷ `OrderingByLessThanOrEqualFunctionNC(`*`fam, lteq[, l]`*`)` (operation)

Called with two arguments, OrderingByLessThanOrEqualFunctionNC returns the ordering on the elements of the elements of the family *fam* according to the LessThanOrEqualFunction (34.3.6) value given by *lteq*, where *lteq* is a function that takes two arguments in *fam* and returns true or false.

Called with three arguments, for a family *fam*, a function *lteq* that takes two arguments in *fam* and returns true or false, and a list *l* of properties of orderings, OrderingByLessThanOrEqualFunctionNC returns the ordering on the elements of *fam* with LessThanOrEqualFunction (34.3.6) value given by *lteq* and with the properties from *l* set to true.

Notice that these functions do not check whether *fam* and *lt* or *lteq* are compatible, and whether the properties listed in *l* are indeed satisfied.

Example

```
gap> f := FreeSemigroup("a","b");;
gap> a := GeneratorsOfSemigroup(f)[1];;
gap> b := GeneratorsOfSemigroup(f)[2];;
gap> lt := function(x,y) return Length(x)<Length(y); end;
function( x, y ) ... end
gap> fam := FamilyObj(a);;
gap> ord := OrderingByLessThanFunctionNC(fam,lt);
Ordering
```

34.3 Properties and basic functionality

34.3.1 IsWellFoundedOrdering

▷ IsWellFoundedOrdering(*ord*) (property)

for an ordering *ord*, returns true if and only if the ordering is well founded. An ordering *ord* is well founded if it admits no infinite descending chains. Normally this property is set at the time of creation of the ordering and there is no general method to check whether a certain ordering is well founded.

34.3.2 IsTotalOrdering

▷ IsTotalOrdering(*ord*) (property)

for an ordering *ord*, returns true if and only if the ordering is total. An ordering *ord* is total if any two elements of the family are comparable under *ord*. Normally this property is set at the time of creation of the ordering and there is no general method to check whether a certain ordering is total.

34.3.3 IsIncomparableUnder

▷ IsIncomparableUnder(*ord, el1, el2*) (operation)

for an ordering *ord* on the elements of the family of *el1* and *el2*, returns true if $el1 \neq el2$ and IsLessThanUnder(*ord*,*el1*,*el2*), IsLessThanUnder(*ord*,*el2*,*el1*) are both false; and returns false otherwise.

34.3.4 FamilyForOrdering

▷ `FamilyForOrdering(`*`ord`*`)` (attribute)

for an ordering *ord*, returns the family of elements that the ordering *ord* compares.

34.3.5 LessThanFunction

▷ `LessThanFunction(`*`ord`*`)` (attribute)

for an ordering *ord*, returns a function f which takes two elements *el*1, *el*2 in `FamilyForOrdering(`*ord*`)` and returns `true` if *el*1 is strictly less than *el*2 (with respect to *ord*), and returns `false` otherwise.

34.3.6 LessThanOrEqualFunction

▷ `LessThanOrEqualFunction(`*`ord`*`)` (attribute)

for an ordering *ord*, returns a function that takes two elements *el*1, *el*2 in `FamilyForOrdering(`*ord*`)` and returns `true` if *el*1 is less than *or equal to* *el*2 (with respect to *ord*), and returns `false` otherwise.

34.3.7 IsLessThanUnder

▷ `IsLessThanUnder(`*`ord, el1, el2`*`)` (operation)

for an ordering *ord* on the elements of the family of *el1* and *el2*, returns `true` if *el1* is (strictly) less than *el2* with respect to *ord*, and `false` otherwise.

34.3.8 IsLessThanOrEqualUnder

▷ `IsLessThanOrEqualUnder(`*`ord, el1, el2`*`)` (operation)

for an ordering *ord* on the elements of the family of *el1* and *el2*, returns `true` if *el1* is less than or equal to *el2* with respect to *ord*, and `false` otherwise.

Example

```
gap> IsLessThanUnder(ord,a,a*b);
true
gap> IsLessThanOrEqualUnder(ord,a*b,a*b);
true
gap> IsIncomparableUnder(ord,a,b);
true
gap> FamilyForOrdering(ord) = FamilyObj(a);
true
```

34.4 Orderings on families of associative words

We now consider orderings on families of associative words.

Examples of families of associative words are the families of elements of a free semigroup or a free monoid; these are the two cases that we consider mostly. Associated with those families is an alphabet, which is the semigroup (resp. monoid) generating set of the correspondent free semigroup (resp. free monoid). For definitions of the orderings considered, see Sims [Sim94].

34.4.1 IsOrderingOnFamilyOfAssocWords

▷ `IsOrderingOnFamilyOfAssocWords(`*`ord`*`)` (property)

for an ordering *ord*, returns true if *ord* is an ordering over a family of associative words.

34.4.2 IsTranslationInvariantOrdering

▷ `IsTranslationInvariantOrdering(`*`ord`*`)` (property)

for an ordering *ord* on a family of associative words, returns `true` if and only if the ordering is translation invariant.

This is a property of orderings on families of associative words. An ordering *ord* over a family F, with alphabet X is translation invariant if `IsLessThanUnder(` *ord*, u, v `)` implies that for any $a,b \in X^*$, `IsLessThanUnder(` *ord*, $a*u*b$, $a*v*b$ `)`.

34.4.3 IsReductionOrdering

▷ `IsReductionOrdering(`*`ord`*`)` (property)

for an ordering *ord* on a family of associative words, returns `true` if and only if the ordering is a reduction ordering. An ordering *ord* is a reduction ordering if it is well founded and translation invariant.

34.4.4 OrderingOnGenerators

▷ `OrderingOnGenerators(`*`ord`*`)` (attribute)

for an ordering *ord* on a family of associative words, returns a list in which the generators are considered. This could be indeed the ordering of the generators in the ordering, but, for example, if a weight is associated to each generator then this is not true anymore. See the example for `WeightLexOrdering` (34.4.8).

34.4.5 LexicographicOrdering

▷ `LexicographicOrdering(`*`D[, gens]`*`)` (operation)

Let *D* be a free semigroup, a free monoid, or the elements family of such a domain. Called with only argument *D*, `LexicographicOrdering` returns the lexicographic ordering on the elements of *D*.

The optional argument *gens* can be either the list of free generators of *D*, in the desired order, or a list of the positions of these generators, in the desired order, and `LexicographicOrdering` returns the lexicographic ordering on the elements of *D* with the ordering on the generators as given.

Example

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> lex := LexicographicOrdering(f,[2,3,1]);
Ordering
gap> IsLessThanUnder(lex,f.2*f.3,f.3);
true
gap> IsLessThanUnder(lex,f.3,f.2);
false
```

34.4.6 ShortLexOrdering

▷ ShortLexOrdering(*D[, gens]*) (operation)

Let *D* be a free semigroup, a free monoid, or the elements family of such a domain. Called with only argument *D*, ShortLexOrdering returns the shortlex ordering on the elements of *D*.

The optional argument *gens* can be either the list of free generators of *D*, in the desired order, or a list of the positions of these generators, in the desired order, and ShortLexOrdering returns the shortlex ordering on the elements of *D* with the ordering on the generators as given.

34.4.7 IsShortLexOrdering

▷ IsShortLexOrdering(*ord*) (property)

for an ordering *ord* of a family of associative words, returns true if and only if *ord* is a shortlex ordering.

Example

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> sl := ShortLexOrdering(f,[2,3,1]);
Ordering
gap> IsLessThanUnder(sl,f.1,f.2);
false
gap> IsLessThanUnder(sl,f.3,f.2);
false
gap> IsLessThanUnder(sl,f.3,f.1);
true
```

34.4.8 WeightLexOrdering

▷ WeightLexOrdering(*D, gens, wt*) (operation)

Let *D* be a free semigroup, a free monoid, or the elements family of such a domain. *gens* can be either the list of free generators of *D*, in the desired order, or a list of the positions of these generators, in the desired order. Let *wt* be a list of weights. WeightLexOrdering returns the weightlex ordering on the elements of *D* with the ordering on the generators and weights of the generators as given.

34.4.9 IsWeightLexOrdering

▷ IsWeightLexOrdering(*ord*) (property)

for an ordering *ord* on a family of associative words, returns true if and only if *ord* is a weightlex ordering.

34.4.10 WeightOfGenerators

▷ WeightOfGenerators(*ord*) (attribute)

for a weightlex ordering *ord*, returns a list with length the size of the alphabet of the family. This list gives the weight of each of the letters of the alphabet which are used for weightlex orderings with respect to the ordering given by OrderingOnGenerators (34.4.4).

Example

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> wtlex := WeightLexOrdering(f,[f.2,f.3,f.1],[3,2,1]);
Ordering
gap> IsLessThanUnder(wtlex,f.1,f.2);
true
gap> IsLessThanUnder(wtlex,f.3,f.2);
true
gap> IsLessThanUnder(wtlex,f.3,f.1);
false
gap> OrderingOnGenerators(wtlex);
[ s2, s3, s1 ]
gap> WeightOfGenerators(wtlex);
[ 3, 2, 1 ]
```

34.4.11 BasicWreathProductOrdering

▷ BasicWreathProductOrdering(*D[, gens]*) (operation)

Let *D* be a free semigroup, a free monoid, or the elements family of such a domain. Called with only argument *D*, BasicWreathProductOrdering returns the basic wreath product ordering on the elements of *D*.

The optional argument *gens* can be either the list of free generators of *D*, in the desired order, or a list of the positions of these generators, in the desired order, and BasicWreathProductOrdering returns the lexicographic ordering on the elements of *D* with the ordering on the generators as given.

34.4.12 IsBasicWreathProductOrdering

▷ IsBasicWreathProductOrdering(*ord*) (property)

Example

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> basic := BasicWreathProductOrdering(f,[2,3,1]);
Ordering
```

```
gap> IsLessThanUnder(basic,f.3,f.1);
true
gap> IsLessThanUnder(basic,f.3*f.2,f.1);
true
gap> IsLessThanUnder(basic,f.3*f.2*f.1,f.1*f.3);
false
```

34.4.13 WreathProductOrdering

▷ `WreathProductOrdering(`*D[, gens], levels*`)` (operation)

Let *D* be a free semigroup, a free monoid, or the elements family of such a domain, let *gens* be either the list of free generators of *D*, in the desired order, or a list of the positions of these generators, in the desired order, and let *levels* be a list of levels for the generators. If *gens* is omitted then the default ordering is taken. `WreathProductOrdering` returns the wreath product ordering on the elements of *D* with the ordering on the generators as given.

34.4.14 IsWreathProductOrdering

▷ `IsWreathProductOrdering(`*ord*`)` (property)

specifies whether an ordering is a wreath product ordering (see `WreathProductOrdering` (34.4.13)).

34.4.15 LevelsOfGenerators

▷ `LevelsOfGenerators(`*ord*`)` (attribute)

for a wreath product ordering *ord*, returns the levels of the generators as given at creation (with respect to `OrderingOnGenerators` (34.4.4)).

Example

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> wrp := WreathProductOrdering(f,[1,2,3],[1,1,2,]);
Ordering
gap> IsLessThanUnder(wrp,f.3,f.1);
false
gap> IsLessThanUnder(wrp,f.3,f.2);
false
gap> IsLessThanUnder(wrp,f.1,f.2);
true
gap> LevelsOfGenerators(wrp);
[ 1, 1, 2 ]
```

Chapter 35

Magmas

This chapter deals with domains (see 31) that are closed under multiplication *. Following [Bou70], we call them *magmas* in GAP. Together with the domains closed under addition + (see 55), they are the basic algebraic structures; every semigroup, monoid (see 51), group (see 39), ring (see 56), or field (see 58) is a magma. In the cases of a *magma-with-one* or *magma-with-inverses*, additional multiplicative structure is present, see 35.1. For functions to create free magmas, see 36.4.

35.1 Magma Categories

35.1.1 IsMagma

▷ `IsMagma(`*obj*`)` (Category)

A *magma* in GAP is a domain M with (not necessarily associative) multiplication *: $M \times M \to M$.

35.1.2 IsMagmaWithOne

▷ `IsMagmaWithOne(`*obj*`)` (Category)

A *magma-with-one* in GAP is a magma M with an operation `^0` (or `One` (31.10.2)) that yields the identity of M.

So a magma-with-one M does always contain a unique multiplicatively neutral element e, i.e., e `*` $m = m = m$ `*` e holds for all $m \in M$ (see `MultiplicativeNeutralElement` (35.4.10)). This element e can be computed with the operation `One` (31.10.2) as `One(` M `)`, and e is also equal to `One(` m `)` and to m`^0` for each element $m \in M$.

Note that a magma may contain a multiplicatively neutral element but *not* an identity (see `One` (31.10.2)), and a magma containing an identity may *not* lie in the category `IsMagmaWithOne` (see Section 31.6).

35.1.3 IsMagmaWithInversesIfNonzero

▷ `IsMagmaWithInversesIfNonzero(`*obj*`)` (Category)

An object in this GAP category is a magma-with-one M with an operation `^-1`: $M \setminus Z \to M \setminus Z$ that maps each element m of $M \setminus Z$ to its inverse m`^-1` (or `Inverse( m )`, see `Inverse` (31.10.8)), where Z is either empty or consists exactly of one element of M.

This category was introduced mainly to describe division rings, since the nonzero elements in a division ring form a group; So an object M in `IsMagmaWithInversesIfNonzero` will usually have both a multiplicative and an additive structure (see 55), and the set Z, if it is nonempty, contains exactly the zero element (see `Zero` (31.10.3)) of M.

35.1.4 IsMagmaWithInverses

▷ `IsMagmaWithInverses(`*obj*`)` (Category)

A *magma-with-inverses* in GAP is a magma-with-one M with an operation `^-1`: $M \to M$ that maps each element m of M to its inverse m`^-1` (or `Inverse( m )`, see `Inverse` (31.10.8)).

Note that not every trivial magma is a magma-with-one, but every trivial magma-with-one is a magma-with-inverses. This holds also if the identity of the magma-with-one is a zero element. So a magma-with-inverses-if-nonzero can be a magma-with-inverses if either it contains no zero element or consists of a zero element that has itself as zero-th power.

35.2 Magma Generation

This section describes functions that create magmas from generators (see `Magma` (35.2.1), `MagmaWithOne` (35.2.2), `MagmaWithInverses` (35.2.3)), the underlying operations for which methods can be installed (see `MagmaByGenerators` (35.2.4), `MagmaWithOneByGenerators` (35.2.5), `MagmaWithInversesByGenerators` (35.2.6)), functions for forming submagmas (see `Submagma` (35.2.7), `SubmagmaWithOne` (35.2.8), `SubmagmaWithInverses` (35.2.9)), and functions that form a magma equal to a given collection (see `AsMagma` (35.2.10), `AsSubmagma` (35.2.11)).

`InjectionZeroMagma` (35.2.13) creates a new magma which is the original magma with a zero adjoined.

35.2.1 Magma

▷ `Magma(`*[Fam,]gens*`)` (function)

returns the magma M that is generated by the elements in the list *gens*, that is, the closure of *gens* under multiplication `\*` (31.12.1). The family *Fam* of M can be entered as the first argument; this is obligatory if *gens* is empty (and hence also M is empty).

35.2.2 MagmaWithOne

▷ `MagmaWithOne(`*[Fam,]gens*`)` (function)

returns the magma-with-one M that is generated by the elements in the list *gens*, that is, the closure of *gens* under multiplication `\*` (31.12.1) and `One` (31.10.2). The family *Fam* of M can be entered as first argument; this is obligatory if *gens* is empty (and hence M is trivial).

35.2.3 MagmaWithInverses

▷ MagmaWithInverses([Fam,]gens) (function)

returns the magma-with-inverses M that is generated by the elements in the list *gens*, that is, the closure of *gens* under multiplication * (31.12.1), One (31.10.2), and Inverse (31.10.8). The family *Fam* of M can be entered as first argument; this is obligatory if *gens* is empty (and hence M is trivial).

35.2.4 MagmaByGenerators

▷ MagmaByGenerators([Fam,]gens) (operation)

An underlying operation for Magma (35.2.1).

35.2.5 MagmaWithOneByGenerators

▷ MagmaWithOneByGenerators([Fam,]gens) (operation)

An underlying operation for MagmaWithOne (35.2.2).

35.2.6 MagmaWithInversesByGenerators

▷ MagmaWithInversesByGenerators([Fam,]gens) (operation)

An underlying operation for MagmaWithInverses (35.2.3).

35.2.7 Submagma

▷ Submagma(D, gens) (function)
▷ SubmagmaNC(D, gens) (function)

Submagma returns the magma generated by the elements in the list *gens*, with parent the domain *D*. SubmagmaNC does the same, except that it is not checked whether the elements of *gens* lie in *D*.

35.2.8 SubmagmaWithOne

▷ SubmagmaWithOne(D, gens) (function)
▷ SubmagmaWithOneNC(D, gens) (function)

SubmagmaWithOne returns the magma-with-one generated by the elements in the list *gens*, with parent the domain *D*. SubmagmaWithOneNC does the same, except that it is not checked whether the elements of *gens* lie in *D*.

35.2.9 SubmagmaWithInverses

▷ SubmagmaWithInverses(D, gens) (function)
▷ SubmagmaWithInversesNC(D, gens) (function)

SubmagmaWithInverses returns the magma-with-inverses generated by the elements in the list *gens*, with parent the domain *D*. SubmagmaWithInversesNC does the same, except that it is not checked whether the elements of *gens* lie in *D*.

35.2.10 AsMagma

▷ AsMagma(*C*) (attribute)

For a collection *C* whose elements form a magma, AsMagma returns this magma. Otherwise fail is returned.

35.2.11 AsSubmagma

▷ AsSubmagma(*D, C*) (operation)

Let *D* be a domain and *C* a collection. If *C* is a subset of *D* that forms a magma then AsSubmagma returns this magma, with parent *D*. Otherwise fail is returned.

35.2.12 IsMagmaWithZeroAdjoined

▷ IsMagmaWithZeroAdjoined(*M*) (Category)

Returns: true or false.

IsMagmaWithZeroAdjoined returns true if the magma *M* was created using InjectionZeroMagma (35.2.13) or MagmaWithZeroAdjoined (35.2.13) and returns false if it was not.

Example

```
gap> S:=Semigroup(Transformation([1,1,1]), Transformation([1,3,2]));;
gap> IsMagmaWithZeroAdjoined(S);
false
gap> M:=MagmaWithZeroAdjoined(S);
<<transformation semigroup on 3 pts with 2 generators>
  with 0 adjoined>
gap> IsMagmaWithZeroAdjoined(M);
true
```

35.2.13 InjectionZeroMagma

▷ InjectionZeroMagma(*M*) (attribute)
▷ MagmaWithZeroAdjoined(*M*) (operation)

InjectionZeroMagma returns an embedding from the magma *M* into a new magma formed from *M* by adjoining a single new element which is the multiplicative zero of the resulting magma. The elements of the new magma form a family of elements in the category IsMultiplicativeElementWithZero (31.14.12) and the magma itself satisfies IsMagmaWithZeroAdjoined (35.2.12).

MagmaWithZeroAdjoined is just shorthand for Range(InjectionZeroMagma(*M*))).

If N is a magma with zero adjoined, then the embedding used to create N can be recovered using UnderlyingInjectionZeroMagma (35.2.14).

Example

```
gap> S:=Monoid(Transformation( [ 7, 7, 5, 3, 1, 3, 7 ] ),
> Transformation( [ 5, 1, 4, 1, 4, 4, 7 ] ));;
gap> MultiplicativeZero(S);
Transformation( [ 7, 7, 7, 7, 7, 7, 7 ] )
gap> T:=MagmaWithZeroAdjoined(S);
<<transformation monoid on 7 pts with 2 generators> with 0 adjoined>
gap> map:=UnderlyingInjectionZeroMagma(T);;
gap> x:=Transformation( [ 7, 7, 7, 3, 7, 3, 7 ] );;
gap> x^map;
<monoid with 0 adjoined elt: Transformation( [ 7, 7, 7, 3, 7, 3, 7 ]
 )>
gap> PreImage(map, x^map)=x;
true
```

35.2.14 UnderlyingInjectionZeroMagma

▷ `UnderlyingInjectionZeroMagma(`*M*`)` (attribute)

`UnderlyingInjectionZeroMagma` returns the embedding used to create the magma with zero adjoined *M*.

Example

```
gap> S:=Monoid(Transformation( [ 8, 7, 5, 3, 1, 3, 8, 8 ] ),
> Transformation( [ 5, 1, 4, 1, 4, 4, 7, 8 ] ));;
gap> MultiplicativeZero(S);
Transformation( [ 8, 8, 8, 8, 8, 8, 8, 8 ] )
gap> T:=MagmaWithZeroAdjoined(S);
<<transformation monoid on 8 pts with 2 generators> with 0 adjoined>
gap> UnderlyingInjectionZeroMagma(T);
MappingByFunction( <transformation monoid on 8 pts with 2 generators>
 , <<transformation monoid on 8 pts with 2 generators>
  with 0 adjoined>, function( elt ) ... end, function( x ) ... end )
```

35.3 Magmas Defined by Multiplication Tables

The most elementary (but of course usually not recommended) way to implement a magma with only few elements is via a multiplication table.

35.3.1 MagmaByMultiplicationTable

▷ `MagmaByMultiplicationTable(`*A*`)` (function)

For a square matrix *A* with n rows such that all entries of *A* are in the range $[1..n]$, `MagmaByMultiplicationTable` returns a magma M with multiplication $*$ defined by *A*. That is, M consists of the elements $m_1, m_2, \ldots, m_n$, and $m_i * m_j = m_k$, with $k = A[i][j]$.

The ordering of elements is defined by $m_1 < m_2 < \cdots < m_n$, so m_i can be accessed as `MagmaElement( M, i )`, see `MagmaElement` (35.3.4).

Example
```
gap> MagmaByMultiplicationTable([[1,2,3],[2,3,1],[1,1,1]]);
<magma with 3 generators>
```

35.3.2 MagmaWithOneByMultiplicationTable

▷ MagmaWithOneByMultiplicationTable(*A*) (function)

The only differences between MagmaByMultiplicationTable (35.3.1) and MagmaWithOneByMultiplicationTable are that the latter returns a magma-with-one (see MagmaWithOne (35.2.2)) if the magma described by the matrix *A* has an identity, and returns fail if not.

Example
```
gap> MagmaWithOneByMultiplicationTable([[1,2,3],[2,3,1],[3,1,1]]);
<magma-with-one with 3 generators>
gap> MagmaWithOneByMultiplicationTable([[1,2,3],[2,3,1],[1,1,1]]);
fail
```

35.3.3 MagmaWithInversesByMultiplicationTable

▷ MagmaWithInversesByMultiplicationTable(*A*) (function)

MagmaByMultiplicationTable (35.3.1) and MagmaWithInversesByMultiplicationTable differ only in that the latter returns magma-with-inverses (see MagmaWithInverses (35.2.3)) if each element in the magma described by the matrix *A* has an inverse, and returns fail if not.

Example
```
gap> MagmaWithInversesByMultiplicationTable([[1,2,3],[2,3,1],[3,1,2]]);
<magma-with-inverses with 3 generators>
gap> MagmaWithInversesByMultiplicationTable([[1,2,3],[2,3,1],[3,2,1]]);
fail
```

35.3.4 MagmaElement

▷ MagmaElement(*M, i*) (function)

For a magma *M* and a positive integer *i*, MagmaElement returns the *i*-th element of *M*, w.r.t. the ordering <. If *M* has less than *i* elements then fail is returned.

35.3.5 MultiplicationTable

▷ MultiplicationTable(*elms*) (attribute)
▷ MultiplicationTable(*M*) (attribute)

For a list *elms* of elements that form a magma M, MultiplicationTable returns a square matrix A of positive integers such that $A[i][j] = k$ holds if and only if $elms[i] * elms[j] = elms[k]$. This matrix can be used to construct a magma isomorphic to M, using MagmaByMultiplicationTable (35.3.1).

For a magma *M*, MultiplicationTable returns the multiplication table w.r.t. the sorted list of elements of *M*.

Example

```
gap> l:= [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ];;
gap> a:= MultiplicationTable( l );
[ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ]
gap> m:= MagmaByMultiplicationTable( a );
<magma with 4 generators>
gap> One( m );
m1
gap> elm:= MagmaElement( m, 2 );  One( elm );  elm^2;
m2
m1
m1
gap> Inverse( elm );
m2
gap> AsGroup( m );
<group of size 4 with 2 generators>
gap> a:= [ [ 1, 2 ], [ 2, 2 ] ];
[ [ 1, 2 ], [ 2, 2 ] ]
gap> m:= MagmaByMultiplicationTable( a );
<magma with 2 generators>
gap> One( m );  Inverse( MagmaElement( m, 2 ) );
m1
fail
```

35.4 Attributes and Properties for Magmas

Note that `IsAssociative` (35.4.7) and `IsCommutative` (35.4.9) always refer to the multiplication of a domain. If a magma *M* has also an *additive structure*, e.g., if *M* is a ring (see 56), then the addition + is always assumed to be associative and commutative, see 31.12.

35.4.1 GeneratorsOfMagma

▷ `GeneratorsOfMagma(`*M*`)` (attribute)

is a list *gens* of elements of the magma *M* that generates *M* as a magma, that is, the closure of *gens* under multiplication `\*` (31.12.1) is *M*.

For a free magma, each generator can also be accessed using the . operator (see `GeneratorsOfDomain` (31.9.2)).

35.4.2 GeneratorsOfMagmaWithOne

▷ `GeneratorsOfMagmaWithOne(`*M*`)` (attribute)

is a list *gens* of elements of the magma-with-one *M* that generates *M* as a magma-with-one, that is, the closure of *gens* under multiplication `\*` (31.12.1) and `One` (31.10.2) is *M*.

For a free magma with one, each generator can also be accessed using the . operator (see `GeneratorsOfDomain` (31.9.2)).

35.4.3 GeneratorsOfMagmaWithInverses

▷ GeneratorsOfMagmaWithInverses(*M*) (attribute)

is a list *gens* of elements of the magma-with-inverses *M* that generates *M* as a magma-with-inverses, that is, the closure of *gens* under multiplication * (31.12.1) and taking inverses (see Inverse (31.10.8)) is *M*.

35.4.4 Centralizer (for a magma and an element)

▷ Centralizer(*M*, *elm*) (operation)
▷ Centralizer(*M*, *S*) (operation)
▷ Centralizer(*class*) (attribute)

For an element *elm* of the magma *M* this operation returns the *centralizer* of *elm*. This is the domain of those elements $m \in M$ that commute with *elm*.

For a submagma *S* it returns the domain of those elements that commute with *all* elements *s* of *S*.

If *class* is a class of objects of a magma (this magma then is stored as the ActingDomain of *class*) such as given by ConjugacyClass (39.10.1), Centralizer returns the centralizer of Representative(*class*) (which is a slight abuse of the notation).

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> Centralizer(g,(1,2,3));
Group([ (1,2,3) ])
gap> Centralizer(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3) ])
gap> Centralizer(g,Subgroup(g,[(1,2,3),(1,2)]));
Group(())
```

35.4.5 Centre

▷ Centre(*M*) (attribute)
▷ Center(*M*) (attribute)

Centre returns the *centre* of the magma *M*, i.e., the domain of those elements $m \in M$ that commute and associate with all elements of *M*. That is, the set $\{m \in M; \forall a,b \in M : ma = am, (ma)b = m(ab), (am)b = a(mb), (ab)m = a(bm)\}$.

Center is just a synonym for Centre.

For associative magmas we have that Centre(*M*) = Centralizer(*M*, *M*), see Centralizer (35.4.4).

The centre of a magma is always commutative (see IsCommutative (35.4.9)). (When one installs a new method for Centre, one should set the IsCommutative (35.4.9) value of the result to true, in order to make this information available.)

35.4.6 Idempotents

▷ Idempotents(*M*) (attribute)

The set of elements of `M` which are their own squares.

35.4.7 IsAssociative

▷ `IsAssociative(M)` (property)

A magma `M` is *associative* if for all elements $a,b,c \in$ `M` the equality $(a$ `*` $b)$ `*` $c = a$ `*` $(b$ `*` $c)$ holds.

An associative magma is called a *semigroup* (see 51), an associative magma-with-one is called a *monoid* (see 51), and an associative magma-with-inverses is called a *group* (see 39).

35.4.8 IsCentral

▷ `IsCentral(M, obj)` (operation)

`IsCentral` returns `true` if the object `obj`, which must either be an element or a magma, commutes with all elements in the magma `M`.

35.4.9 IsCommutative

▷ `IsCommutative(M)` (property)
▷ `IsAbelian(M)` (property)

A magma `M` is *commutative* if for all elements $a,b \in$ `M` the equality a `*` $b = b$ `*` a holds. `IsAbelian` is a synonym of `IsCommutative`.

Note that the commutativity of the *addition* `\+` (31.12.1) in an additive structure can be tested with `IsAdditivelyCommutative` (55.3.1).

35.4.10 MultiplicativeNeutralElement

▷ `MultiplicativeNeutralElement(M)` (attribute)

returns the element e in the magma `M` with the property that e `*` $m = m = m$ `*` e holds for all $m \in$ `M`, if such an element exists. Otherwise `fail` is returned.

A magma that is not a magma-with-one can have a multiplicative neutral element e; in this case, e *cannot* be obtained as `One( M )`, see `One` (31.10.2).

35.4.11 MultiplicativeZero

▷ `MultiplicativeZero(M)` (attribute)
▷ `IsMultiplicativeZero(M, z)` (operation)

`MultiplicativeZero` returns the multiplicative zero of the magma `M` which is the element `z` in `M` such that `z * m = m * z = z` for all `m` in `M`.

`IsMultiplicativeZero` returns `true` if the element `z` of the magma `M` equals the multiplicative zero of `M`.

Example

```
gap> S:=Semigroup( Transformation( [ 1, 1, 1 ] ),
> Transformation( [ 2, 3, 1 ] ) );
<transformation semigroup on 3 pts with 2 generators>
gap> MultiplicativeZero(S);
fail
gap> S:=Semigroup( Transformation( [ 1, 1, 1 ] ),
> Transformation( [ 1, 3, 2 ] ) );
<transformation semigroup on 3 pts with 2 generators>
gap> MultiplicativeZero(S);
Transformation( [ 1, 1, 1 ] )
```

35.4.12 SquareRoots

▷ `SquareRoots(`*M*`, `*elm*`)` (operation)

is the proper set of all elements r in the magma *M* such that $r * r =$ *elm* holds.

35.4.13 TrivialSubmagmaWithOne

▷ `TrivialSubmagmaWithOne(`*M*`)` (attribute)

is the magma-with-one that has the identity of the magma-with-one *M* as only element.

Chapter 36

Words

This chapter describes categories of *words* and *nonassociative words*, and operations for them. For information about *associative words*, which occur for example as elements in free groups, see Chapter 37.

36.1 Categories of Words and Nonassociative Words

36.1.1 IsWord

▷ `IsWord(obj)` (Category)
▷ `IsWordWithOne(obj)` (Category)
▷ `IsWordWithInverse(obj)` (Category)

Given a free multiplicative structure M that is freely generated by a subset X, any expression of an element in M as an iterated product of elements in X is called a *word* over X.

Interesting cases of free multiplicative structures are those of free semigroups, free monoids, and free groups, where the multiplication is associative (see `IsAssociative` (35.4.7)), which are described in Chapter 37, and also the case of free magmas, where the multiplication is nonassociative (see `IsNonassocWord` (36.1.3)).

Elements in free magmas (see `FreeMagma` (36.4.1)) lie in the category `IsWord`; similarly, elements in free magmas-with-one (see `FreeMagmaWithOne` (36.4.2)) lie in the category `IsWordWithOne`, and so on.

`IsWord` is mainly a "common roof" for the two *disjoint* categories `IsAssocWord` (37.1.1) and `IsNonassocWord` (36.1.3) of associative and nonassociative words. This means that associative words are *not* regarded as special cases of nonassociative words. The main reason for this setup is that we are interested in different external representations for associative and nonassociative words (see 36.5 and 37.7).

Note that elements in finitely presented groups and also elements in polycyclic groups in GAP are *not* in `IsWord` although they are usually called words, see Chapters 47 and 46.

Words are *constants* (see 12.6), that is, they are not copyable and not mutable.

The usual way to create words is to form them as products of known words, starting from *generators* of a free structure such as a free magma or a free group (see `FreeMagma` (36.4.1), `FreeGroup` (37.2.1)).

Words are also used to implement free algebras, in the same way as group elements are used to implement group algebras (see 62.3 and Chapter 65).

Example

```
gap> m:= FreeMagmaWithOne( 2 );;  gens:= GeneratorsOfMagmaWithOne( m );
[ x1, x2 ]
gap> w1:= gens[1] * gens[2] * gens[1];
((x1*x2)*x1)
gap> w2:= gens[1] * ( gens[2] * gens[1] );
(x1*(x2*x1))
gap> w1 = w2;  IsAssociative( m );
false
false
gap> IsWord( w1 );  IsAssocWord( w1 );  IsNonassocWord( w1 );
true
false
true
gap> s:= FreeMonoid( 2 );;  gens:= GeneratorsOfMagmaWithOne( s );
[ m1, m2 ]
gap> u1:= ( gens[1] * gens[2] ) * gens[1];
m1*m2*m1
gap> u2:= gens[1] * ( gens[2] * gens[1] );
m1*m2*m1
gap> u1 = u2;  IsAssociative( s );
true
true
gap> IsWord( u1 );  IsAssocWord( u1 );  IsNonassocWord( u1 );
true
true
false
gap> a:= (1,2,3);;  b:= (1,2);;
gap> w:= a*b*a;;  IsWord( w );
false
```

36.1.2 IsWordCollection

▷ `IsWordCollection(`*obj*`)` (Category)

`IsWordCollection` is the collections category (see `CategoryCollections` (30.2.4)) of `IsWord` (36.1.1).

Example

```
gap> IsWordCollection( m );  IsWordCollection( s );
true
true
gap> IsWordCollection( [ "a", "b" ] );
false
```

36.1.3 IsNonassocWord

▷ `IsNonassocWord(`*obj*`)` (Category)

▷ `IsNonassocWordWithOne(`*obj*`)` (Category)

A *nonassociative word* in GAP is an element in a free magma or a free magma-with-one (see 36.4).

The default methods for `ViewObj` (6.3.5) and `PrintObj` (6.3.5) show nonassociative words as products of letters, where the succession of multiplications is determined by round brackets.

In this sense each nonassociative word describes a "program" to form a product of generators. (Also associative words can be interpreted as such programs, except that the exact succession of multiplications is not prescribed due to the associativity.) The function `MappedWord` (36.3.1) implements a way to apply such a program. A more general way is provided by straight line programs (see 37.8).

Note that associative words (see Chapter 37) are *not* regarded as special cases of nonassociative words (see `IsWord` (36.1.1)).

36.1.4 IsNonassocWordCollection

▷ `IsNonassocWordCollection(`*obj*`)` (Category)
▷ `IsNonassocWordWithOneCollection(`*obj*`)` (Category)

`IsNonassocWordCollection` is the collections category (see `CategoryCollections` (30.2.4)) of `IsNonassocWord` (36.1.3), and `IsNonassocWordWithOneCollection` is the collections category of `IsNonassocWordWithOne` (36.1.3).

36.2 Comparison of Words

36.2.1 \= (for nonassociative words)

▷ `\=(`*w1*`,` *w2*`)` (function)

Two words are equal if and only if they are words over the same alphabet and with equal external representations (see 36.5 and 37.7). For nonassociative words, the latter means that the words arise from the letters of the alphabet by the same sequence of multiplications.

36.2.2 \< (for nonassociative words)

▷ `\<(`*w1*`,` *w2*`)` (function)

Words are ordered according to their external representation. More precisely, two words can be compared if they are words over the same alphabet, and the word with smaller external representation is smaller. For nonassociative words, the ordering is defined in 36.5; associative words are ordered by the shortlex ordering via < (see 37.7).

Note that the alphabet of a word is determined by its family (see 13.1), and that the result of each call to `FreeMagma` (36.4.1), `FreeGroup` (37.2.1) etc. consists of words over a new alphabet. In particular, there is no "universal" empty word, every families of words in `IsWordWithOne` (36.1.1) has its own empty word.

Example

```
gap> m:= FreeMagma( "a", "b" );;
gap> x:= FreeMagma( "a", "b" );;
gap> mgens:= GeneratorsOfMagma( m );
[ a, b ]
gap> xgens:= GeneratorsOfMagma( x );
```

```
[ a, b ]
gap> a:= mgens[1];;  b:= mgens[2];;
gap> a = xgens[1];
false
gap> a*(a*a) = (a*a)*a;  a*b = b*a;  a*a = a*a;
false
false
true
gap> a < b;  b < a;  a < a*b;
true
false
true
```

36.3 Operations for Words

Two words can be multiplied via * only if they are words over the same alphabet (see 36.2).

36.3.1 MappedWord

▷ MappedWord(*w, gens, imgs*) (operation)

MappedWord returns the object that is obtained by replacing each occurrence in the word *w* of a generator in the list *gens* by the corresponding object in the list *imgs*. The lists *gens* and *imgs* must of course have the same length.

MappedWord needs to do some preprocessing to get internal generator numbers etc. When mapping many (several thousand) words, an explicit loop over the words syllables might be faster.

For example, if the elements in *imgs* are all *associative words* (see Chapter 37) in the same family as the elements in *gens*, and some of them are equal to the corresponding generators in *gens*, then those may be omitted from *gens* and *imgs*. In this situation, the special case that the lists *gens* and *imgs* have only length 1 is handled more efficiently by EliminatedWord (37.4.6).

Example

```
gap> m:= FreeMagma( "a", "b" );;  gens:= GeneratorsOfMagma( m );;
gap> a:= gens[1];  b:= gens[2];
a
b
gap> w:= (a*b)*((b*a)*a)*b;
(((a*b)*((b*a)*a))*b)
gap> MappedWord( w, gens, [ (1,2), (1,2,3,4) ] );
(2,4,3)
gap> a:= (1,2);; b:= (1,2,3,4);;  (a*b)*((b*a)*a)*b;
(2,4,3)
gap> f:= FreeGroup( "a", "b" );;
gap> a:= GeneratorsOfGroup(f)[1];;  b:= GeneratorsOfGroup(f)[2];;
gap> w:= a^5*b*a^2/b^4*a;
a^5*b*a^2*b^-4*a
gap> MappedWord( w, [ a, b ], [ (1,2), (1,2,3,4) ] );
(1,3,4,2)
gap> (1,2)^5*(1,2,3,4)*(1,2)^2/(1,2,3,4)^4*(1,2);
(1,3,4,2)
```

```
gap> MappedWord( w, [ a ], [ a^2 ] );
a^10*b*a^4*b^-4*a^2
```

36.4 Free Magmas

The easiest way to create a family of words is to construct the free object generated by these words. Each such free object defines a unique alphabet, and its generators are simply the words of length one over this alphabet; These generators can be accessed via GeneratorsOfMagma (35.4.1) in the case of a free magma, and via GeneratorsOfMagmaWithOne (35.4.2) in the case of a free magma-with-one.

36.4.1 FreeMagma

▷ FreeMagma(*rank[, name]*) (function)
▷ FreeMagma(*name1, name2, ...*) (function)
▷ FreeMagma(*names*) (function)
▷ FreeMagma(*infinity, name, init*) (function)

Called with a positive integer *rank*, FreeMagma returns a free magma on *rank* generators. If the optional argument *name* is given then the generators are printed as *name*1, *name*2 etc., that is, each name is the concatenation of the string *name* and an integer from 1 to *range*. The default for *name* is the string "m".

Called in the second form, FreeMagma returns a free magma on as many generators as arguments, printed as *name1*, *name2* etc.

Called in the third form, FreeMagma returns a free magma on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*[*i*].

Called in the fourth form, FreeMagma returns a free magma on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

36.4.2 FreeMagmaWithOne

▷ FreeMagmaWithOne(*rank[, name]*) (function)
▷ FreeMagmaWithOne(*name1, name2, ...*) (function)
▷ FreeMagmaWithOne(*names*) (function)
▷ FreeMagmaWithOne(*infinity, name, init*) (function)

Called with a positive integer *rank*, FreeMagmaWithOne returns a free magma-with-one on *rank* generators. If the optional argument *name* is given then the generators are printed as *name*1, *name*2 etc., that is, each name is the concatenation of the string *name* and an integer from 1 to *range*. The default for *name* is the string "m".

Called in the second form, FreeMagmaWithOne returns a free magma-with-one on as many generators as arguments, printed as *name1*, *name2* etc.

Called in the third form, FreeMagmaWithOne returns a free magma-with-one on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*[*i*].

Called in the fourth form, FreeMagmaWithOne returns a free magma-with-one on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

Example

```
gap> FreeMagma( 3 );
<free magma on the generators [ x1, x2, x3 ]>
gap> FreeMagma( "a", "b" );
<free magma on the generators [ a, b ]>
gap> FreeMagma( infinity );
<free magma with infinity generators>
gap> FreeMagmaWithOne( 3 );
<free magma-with-one on the generators [ x1, x2, x3 ]>
gap> FreeMagmaWithOne( "a", "b" );
<free magma-with-one on the generators [ a, b ]>
gap> FreeMagmaWithOne( infinity );
<free magma-with-one with infinity generators>
```

Remember that the names of generators used for printing do not necessarily distinguish letters of the alphabet; so it is possible to create arbitrarily weird situations by choosing strange letter names.

Example

```
gap> m:= FreeMagma( "x", "x" );  gens:= GeneratorsOfMagma( m );;
<free magma on the generators [ x, x ]>
gap> gens[1] = gens[2];
false
```

36.5 External Representation for Nonassociative Words

The external representation of nonassociative words is defined as follows. The i-th generator of the family of elements in question has external representation i, the identity (if exists) has external representation 0, the inverse of the i-th generator (if exists) has external representation $-i$. If v and w are nonassociative words with external representations e_v and e_w, respectively then the product $v * w$ has external representation $[e_v, e_w]$. So the external representation of any nonassociative word is either an integer or a nested list of integers and lists, where each list has length two.

One can create a nonassociative word from a family of words and the external representation of a nonassociative word using ObjByExtRep (79.16.1).

Example

```
gap> m:= FreeMagma( 2 );;  gens:= GeneratorsOfMagma( m );
[ x1, x2 ]
gap> w:= ( gens[1] * gens[2] ) * gens[1];
((x1*x2)*x1)
gap> ExtRepOfObj( w );  ExtRepOfObj( gens[1] );
[ [ 1, 2 ], 1 ]
1
gap>  ExtRepOfObj( w*w );
[ [ [ 1, 2 ], 1 ], [ [ 1, 2 ], 1 ] ]
gap> ObjByExtRep( FamilyObj( w ), 2 );
x2
gap> ObjByExtRep( FamilyObj( w ), [ 1, [ 2, 1 ] ] );
(x1*(x2*x1))
```

Chapter 37

Associative Words

37.1 Categories of Associative Words

Associative words are used to represent elements in free groups, semigroups and monoids in GAP (see 37.2). An associative word is just a sequence of letters, where each letter is an element of an alphabet (in the following called a *generator*) or its inverse. Associative words can be multiplied; in free monoids also the computation of an identity is permitted, in free groups also the computation of inverses (see 37.4).

Different alphabets correspond to different families of associative words. There is no relation whatsoever between words in different families.

Example

```
gap> f:= FreeGroup( "a", "b", "c" );
<free group on the generators [ a, b, c ]>
gap> gens:= GeneratorsOfGroup(f);
[ a, b, c ]
gap> w:= gens[1]*gens[2]/gens[3]*gens[2]*gens[1]/gens[1]*gens[3]/gens[2];
a*b*c^-1*b*c*b^-1
gap> w^-1;
b*c^-1*b^-1*c*b^-1*a^-1
```

Words are displayed as products of letters. The letters are usually printed like `f1`, `f2`, ..., but it is possible to give user defined names to them, which can be arbitrary strings. These names do not necessarily identify a unique letter (generator), it is possible to have several letters –even in the same family– that are displayed in the same way. Note also that *there is no relation between the names of letters and variable names*. In the example above, we might have typed

Example

```
gap> a:= f.1;; b:= f.2;; c:= f.3;;
```

(*Interactively*, the function `AssignGeneratorVariables` (37.2.3) provides a shorthand for this.) This allows us to define `w` more conveniently:

Example

```
gap> w := a*b/c*b*a/a*c/b;
a*b*c^-1*b*c*b^-1
```

Using homomorphisms it is possible to express elements of a group as words in terms of generators, see 39.5.

37.1.1 IsAssocWord

▷ IsAssocWord(*obj*) (Category)
▷ IsAssocWordWithOne(*obj*) (Category)
▷ IsAssocWordWithInverse(*obj*) (Category)

IsAssocWord is the category of associative words in free semigroups, IsAssocWordWithOne is the category of associative words in free monoids (which admit the operation One (31.10.2) to compute an identity), IsAssocWordWithInverse is the category of associative words in free groups (which have an inverse). See IsWord (36.1.1) for more general categories of words.

37.2 Free Groups, Monoids and Semigroups

Usually a family of associative words will be generated by constructing the free object generated by them. See FreeMonoid (51.2.9), FreeSemigroup (51.1.10) for details.

37.2.1 FreeGroup

▷ FreeGroup(*[wfilt,]rank[, name]*) (function)
▷ FreeGroup(*[wfilt,]name1, name2, ...*) (function)
▷ FreeGroup(*[wfilt,]names*) (function)
▷ FreeGroup(*[wfilt,]infinity, name, init*) (function)

Called with a positive integer *rank*, FreeGroup returns a free group on *rank* generators. If the optional argument *name* is given then the generators are printed as *name*1, *name*2 etc., that is, each name is the concatenation of the string *name* and an integer from 1 to *range*. The default for *name* is the string "f".

Called in the second form, FreeGroup returns a free group on as many generators as arguments, printed as *name1*, *name2* etc.

Called in the third form, FreeGroup returns a free group on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*[*i*].

Called in the fourth form, FreeGroup returns a free group on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either IsSyllableWordsFamily or IsLetterWordsFamily or IsWLetterWordsFamily or IsBLetterWordsFamily. This filter then specifies the representation used for the elements of the free group (see 37.6). If no such filter is given, a letter representation is used.

(For interfacing to old code that omits the representation flag, use of the syllable representation is also triggered by setting the option FreeGroupFamilyType to the string "syllable".)

37.2.2 IsFreeGroup

▷ IsFreeGroup(*obj*) (Category)

Any group consisting of elements in IsAssocWordWithInverse (37.1.1) lies in the filter IsFreeGroup; this holds in particular for any group created with FreeGroup (37.2.1), or any subgroup of such a group.

Also see Chapter 47.

37.2.3 AssignGeneratorVariables

▷ AssignGeneratorVariables(*G*) (function)

If *G* is a group, whose generators are represented by symbols (for example a free group, a finitely presented group or a pc group) this function assigns these generators to global variables with the same names.

The aim of this function is to make it easy in interactive use to work with (for example) a free group. It is a shorthand for a sequence of assignments of the form

Example

```
var1:=GeneratorsOfGroup(G)[1];
var2:=GeneratorsOfGroup(G)[2];
...
varn:=GeneratorsOfGroup(G)[n];
```

However, since overwriting global variables can be very dangerous, *it is not permitted to use this function within a function*. (If –despite this warning– this is done, the result is undefined.)

If the assignment overwrites existing variables a warning is given, if any of the variables if write protected, or any of the generator names would not be a proper variable name, an error is raised.

37.3 Comparison of Associative Words

37.3.1 \= (for associative words)

▷ \=(*w1, w2*) (function)

Two associative words are equal if they are words over the same alphabet and if they are sequences of the same letters. This is equivalent to saying that the external representations of the words are equal, see 37.7 and 36.2.

There is no “universal” empty word, every alphabet (that is, every family of words) has its own empty word.

Example

```
gap> f:= FreeGroup( "a", "b", "b" );;
gap> gens:= GeneratorsOfGroup(f);
[ a, b, b ]
gap> gens[2] = gens[3];
false
gap> x:= gens[1]*gens[2];
a*b
gap> y:= gens[2]/gens[2]*gens[1]*gens[2];
a*b
gap> x = y;
true
gap> z:= gens[2]/gens[2]*gens[1]*gens[3];
```

```
a*b
gap> x = z;
false
```

37.3.2 \< (for associative words)

▷ `\<(w1, w2)` (function)

The ordering of associative words is defined by length and lexicography (this ordering is called *short-lex* ordering), that is, shorter words are smaller than longer words, and words of the same length are compared w.r.t. the lexicographical ordering induced by the ordering of generators. Generators are sorted according to the order in which they were created. If the generators are invertible then each generator g is larger than its inverse `g^-1`, and `g^-1` is larger than every generator that is smaller than g.

Example

```
gap> f:= FreeGroup( 2 );;  gens:= GeneratorsOfGroup( f );;
gap> a:= gens[1];;  b:= gens[2];;
gap> One(f) < a^-1;  a^-1 < a;  a < b^-1;  b^-1 < b; b < a^2;  a^2 < a*b;
true
true
true
true
true
true
```

37.3.3 IsShortLexLessThanOrEqual

▷ `IsShortLexLessThanOrEqual(u, v)` (function)

returns `IsLessThanOrEqualUnder(ord, u, v)` where *ord* is the short less ordering for the family of `u` and `v`. (This is here for compatibility with GAP 4.2.)

37.3.4 IsBasicWreathLessThanOrEqual

▷ `IsBasicWreathLessThanOrEqual(u, v)` (function)

returns `IsLessThanOrEqualUnder(ord, u, v)` where *ord* is the basic wreath product ordering for the family of `u` and `v`. (This is here for compatibility with GAP 4.2.)

37.4 Operations for Associative Words

The product of two given associative words is defined as the freely reduced concatenation of the words. Besides the multiplication `\*` (31.12.1), the arithmetical operators `One` (31.10.2) (if the word lies in a family with identity) and (if the generators are invertible) `Inverse` (31.10.8), `\/` (31.12.1),`\^` (31.12.1), `Comm` (31.12.3), and `LeftQuotient` (31.12.2) are applicable to associative words, see 31.12.

See also `MappedWord` (36.3.1), an operation that is applicable to arbitrary words.

See Section 37.6 for a discussion of the internal representations of associative words that are supported by GAP. Note that operations to extract or act on parts of words (letter or syllables) can carry substantially different costs, depending on the representation the words are in.

37.4.1 Length (for a associative word)

▷ `Length(w)` (attribute)

For an associative word `w`, `Length` returns the number of letters in `w`.

Example

```
gap> f := FreeGroup("a","b");; gens := GeneratorsOfGroup(f);;
gap> a := gens[1];; b := gens[2];;w := a^5*b*a^2*b^-4*a;;
gap>  w; Length( w );  Length( a^17 );  Length( w^0 );
a^5*b*a^2*b^-4*a
13
17
0
```

37.4.2 ExponentSumWord

▷ `ExponentSumWord(w, gen)` (operation)

For an associative word `w` and a generator *gen*, `ExponentSumWord` returns the number of times *gen* appears in `w` minus the number of times its inverse appears in `w`. If both *gen* and its inverse do not occur in `w` then 0 is returned. *gen* may also be the inverse of a generator.

Example

```
gap> w;  ExponentSumWord( w, a );  ExponentSumWord( w, b );
a^5*b*a^2*b^-4*a
8
-3
gap> ExponentSumWord( (a*b*a^-1)^3, a );  ExponentSumWord( w, b^-1 );
0
3
```

37.4.3 Subword

▷ `Subword(w, from, to)` (operation)

For an associative word `w` and two positive integers *from* and *to*, `Subword` returns the subword of `w` that begins at position *from* and ends at position *to*. Indexing is done with origin 1.

Example

```
gap> w;  Subword( w, 3, 7 );
a^5*b*a^2*b^-4*a
a^3*b*a
```

37.4.4 PositionWord

▷ `PositionWord(w, sub, from)` (operation)

Let *w* and *sub* be associative words, and *from* a positive integer. PositionWord returns the position of the first occurrence of *sub* as a subword of *w*, starting at position *from*. If there is no such occurrence, fail is returned. Indexing is done with origin 1.

In other words, PositionWord(*w*, *sub*, *from*) is the smallest integer *i* larger than or equal to *from* such that Subword(*w*, *i*, *i*+Length(*sub*)-1) = *sub*, see Subword (37.4.3).

Example

```
gap> w;  PositionWord( w, a/b, 1 );
a^5*b*a^2*b^-4*a
8
gap> Subword( w, 8, 9 );
a*b^-1
gap> PositionWord( w, a^2, 1 );
1
gap> PositionWord( w, a^2, 2 );
2
gap> PositionWord( w, a^2, 6 );
7
gap> PositionWord( w, a^2, 8 );
fail
```

37.4.5 SubstitutedWord

▷ SubstitutedWord(*w, from, to, by*) (operation)
▷ SubstitutedWord(*w, sub, from, by*) (operation)

Let *w* be an associative word.

In the first form, SubstitutedWord returns the associative word obtained by replacing the subword of *w* that begins at position *from* and ends at position *to* by the associative word *by*. *from* and *to* must be positive integers, indexing is done with origin 1. In other words, SubstitutedWord(*w*, *from*, *to*, *by*) is the product of the three words Subword(*w*, 1, *from*-1), *by*, and Subword(*w*, *to*+1, Length(*w*)), see Subword (37.4.3).

In the second form, SubstitutedWord returns the associative word obtained by replacing the first occurrence of the associative word *sub* of *w*, starting at position *from*, by the associative word *by*; if there is no such occurrence, fail is returned.

Example

```
gap> w;  SubstitutedWord( w, 3, 7, a^19 );
a^5*b*a^2*b^-4*a
a^22*b^-4*a
gap> SubstitutedWord( w, a, 6, b^7 );
a^5*b^8*a*b^-4*a
gap> SubstitutedWord( w, a*b, 6, b^7 );
fail
```

37.4.6 EliminatedWord

▷ EliminatedWord(*w, gen, by*) (operation)

For an associative word *w*, a generator *gen*, and an associative word *by*, EliminatedWord returns the associative word obtained by replacing each occurrence of *gen* in *w* by *by*.

Example

```
gap> w;  EliminatedWord( w, a, a^2 );  EliminatedWord( w, a, b^-1 );
a^5*b*a^2*b^-4*a
a^10*b*a^4*b^-4*a^2
b^-11
```

37.5 Operations for Associative Words by their Syllables

For an associative word $w = x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k}$ over an alphabet containing $x_1, x_2, \ldots, x_k$, such that $x_i \neq x_{i+1}^{\pm 1}$ for $1 \leq i \leq k-1$, the subwords $x_i^{e_i}$ are uniquely determined; these powers of generators are called the *syllables* of w.

37.5.1 NumberSyllables

▷ `NumberSyllables(w)` (attribute)

`NumberSyllables` returns the number of syllables of the associative word `w`.

37.5.2 ExponentSyllable

▷ `ExponentSyllable(w, i)` (operation)

`ExponentSyllable` returns the exponent of the `i`-th syllable of the associative word `w`.

37.5.3 GeneratorSyllable

▷ `GeneratorSyllable(w, i)` (operation)

`GeneratorSyllable` returns the number of the generator that is involved in the `i`-th syllable of the associative word `w`.

37.5.4 SubSyllables

▷ `SubSyllables(w, from, to)` (operation)

`SubSyllables` returns the subword of the associative word `w` that consists of the syllables from positions *from* to *to*, where *from* and *to* must be positive integers, and indexing is done with origin 1.

Example

```
gap> w;  NumberSyllables( w );
a^5*b*a^2*b^-4*a
5
gap> ExponentSyllable( w, 3 );
2
gap> GeneratorSyllable( w, 3 );
1
gap> SubSyllables( w, 2, 3 );
b*a^2
```

37.6 Representations for Associative Words

GAP provides two different internal kinds of representations of associative words. The first one are "syllable representations" in which words are stored in syllable (i.e. generator,exponent) form. (Older versions of GAP only used this representation.) The second kind are "letter representations" in which each letter in a word is represented by its index number. Negative numbers are used for inverses. Unless the syllable representation is specified explicitly when creating the free group/monoid or semigroup, a letter representation is used by default.

Depending on the task in mind, either of these two representations will perform better in time or in memory use and algorithms that are syllable or letter based (for example `GeneratorSyllable` (37.5.3) and `Subword` (37.4.3)) perform substantially better in the corresponding representation. For example when creating pc groups (see 46), it is advantageous to use a syllable representation while calculations in free groups usually benefit from using a letter representation.

37.6.1 IsLetterAssocWordRep

▷ `IsLetterAssocWordRep(`*obj*`)` (Representation)

A word in letter representation stores a list of generator/inverses numbers (as given by `LetterRepAssocWord` (37.6.8)). Letter access is fast, syllable access is slow for such words.

37.6.2 IsLetterWordsFamily

▷ `IsLetterWordsFamily(`*obj*`)` (Category)

A letter word family stores words by default in letter form.

Internally, there are letter representations that use integers (4 Byte) to represent a generator and letter representations that use single bytes to represent a character. The latter are more memory efficient, but can only be used if there are less than 128 generators (in which case they are used by default).

37.6.3 IsBLetterAssocWordRep

▷ `IsBLetterAssocWordRep(`*obj*`)` (Representation)
▷ `IsWLetterAssocWordRep(`*obj*`)` (Representation)

these two subrepresentations of `IsLetterAssocWordRep` (37.6.1) indicate whether the word is stored as a list of bytes (in a string) or as a list of integers).

37.6.4 IsBLetterWordsFamily

▷ `IsBLetterWordsFamily(`*obj*`)` (Category)
▷ `IsWLetterWordsFamily(`*obj*`)` (Category)

These two subcategories of `IsLetterWordsFamily` (37.6.2) specify the type of letter representation to be used.

37.6.5 IsSyllableAssocWordRep

▷ IsSyllableAssocWordRep(*obj*) (Representation)

A word in syllable representation stores generator/exponents pairs (as given by ExtRepOfObj (79.16.1). Syllable access is fast, letter access is slow for such words.

37.6.6 IsSyllableWordsFamily

▷ IsSyllableWordsFamily(*obj*) (Category)

A syllable word family stores words by default in syllable form. There are also different versions of syllable representations, which compress a generator exponent pair in 8, 16 or 32 bits or use a pair of integers. Internal mechanisms try to make this as memory efficient as possible.

37.6.7 Is16BitsFamily

▷ Is16BitsFamily(*obj*) (Category)
▷ Is32BitsFamily(*obj*) (Category)
▷ IsInfBitsFamily(*obj*) (Category)

Regardless of the internal representation used, it is possible to convert a word in a list of numbers in letter or syllable representation and vice versa.

37.6.8 LetterRepAssocWord

▷ LetterRepAssocWord(*w[, gens]*) (operation)

The *letter representation* of an associated word is as a list of integers, each entry corresponding to a group generator. Inverses of the generators are represented by negative numbers. The generator numbers are as associated to the family.

This operation returns the letter representation of the associative word *w*.

In the call with two arguments, the generator numbers correspond to the generator order given in the list *gens*.

(For words stored in syllable form the letter representation has to be computed.)

37.6.9 AssocWordByLetterRep

▷ AssocWordByLetterRep(*Fam, lrep[, gens]*) (operation)

takes a letter representation *lrep* (see LetterRepAssocWord (37.6.8)) and returns an associative word in family *fam* corresponding to this letter representation.

If *gens* is given, the numbers in the letter representation correspond to *gens*.

Example

```
gap> w:=AssocWordByLetterRep( FamilyObj(a), [-1,2,1,-2,-2,-2,1,1,1,1]);
a^-1*b*a*b^-3*a^4
gap> LetterRepAssocWord( w^2 );
[ -1, 2, 1, -2, -2, -2, 1, 1, 1, 2, 1, -2, -2, -2, 1, 1, 1, 1 ]
```

The external representation (see section 37.7) can be used if a syllable representation is needed.

37.7 The External Representation for Associative Words

The external representation of the associative word w is defined as follows. If $w = g_{i_1}^{e_1} * g_{i_2}^{e_2} * \cdots * g_{i_k}^{e_k}$ is a word over the alphabet $g_1, g_2, \ldots$, i.e., g_i denotes the i-th generator of the family of w, then w has external representation $[i_1, e_1, i_2, e_2, \ldots, i_k, e_k]$. The empty list describes the identity element (if exists) of the family. Exponents may be negative if the family allows inverses. The external representation of an associative word is guaranteed to be freely reduced; for example, $g_1 * g_2 * g_2^{-1} * g_1$ has the external representation `[ 1, 2 ]`.

Regardless of the family preference for letter or syllable representations (see 37.6), `ExtRepOfObj` and `ObjByExtRep` can be used and interface to this "syllable"-like representation.

Example

```
gap> w:= ObjByExtRep( FamilyObj(a), [1,5,2,-7,1,3,2,4,1,-2] );
a^5*b^-7*a^3*b^4*a^-2
gap> ExtRepOfObj( w^2 );
[ 1, 5, 2, -7, 1, 3, 2, 4, 1, 3, 2, -7, 1, 3, 2, 4, 1, -2 ]
```

37.8 Straight Line Programs

Straight line programs describe an efficient way for evaluating an abstract word at concrete generators, in a more efficient way than with `MappedWord` (36.3.1). For example, the associative word *ababbab* of length 7 can be computed from the generators a, b with only four multiplications, by first computing $c = ab$, then $d = cb$, and then cdc; Alternatively, one can compute $c = ab$, $e = bc$, and aee. In each step of these computations, one forms words in terms of the words computed in the previous steps.

A straight line program in GAP is represented by an object in the category `IsStraightLineProgram` (37.8.1)) that stores a list of "lines" each of which has one of the following three forms.

1. a nonempty dense list l of integers,
2. a pair $[l, i]$ where l is a list of form 1. and i is a positive integer,
3. a list $[l_1, l_2, \ldots, l_k]$ where each l_i is a list of form 1.; this may occur only for the last line of the program.

The lists of integers that occur are interpreted as external representations of associative words (see Section 37.7); for example, the list $[1, 3, 2, -1]$ represents the word $g_1^3 g_2^{-1}$, with g_1 and g_2 the first and second abstract generator, respectively.

For the meaning of the list of lines, see `ResultOfStraightLineProgram` (37.8.5).

Straight line programs can be constructed using `StraightLineProgram` (37.8.2).

Defining attributes for straight line programs are `NrInputsOfStraightLineProgram` (37.8.4) and `LinesOfStraightLineProgram` (37.8.3). Another operation for straight line programs is `ResultOfStraightLineProgram` (37.8.5).

Special methods applicable to straight line programs are installed for the operations `Display` (6.3.6), `IsInternallyConsistent` (12.8.3), `PrintObj` (6.3.5), and `ViewObj` (6.3.5).

For a straight line program *prog*, the default `Display` (6.3.6) method prints the interpretation of *prog* as a sequence of assignments of associative words; a record with components `gensnames`

(with value a list of strings) and `listname` (a string) may be entered as second argument of `Display` (6.3.6), in this case these names are used, the default for `gensnames` is `[ g1, g2, ... ]`, the default for `listname` is *r*.

37.8.1 IsStraightLineProgram

▷ `IsStraightLineProgram(`*obj*`)` (Category)

Each straight line program in GAP lies in the category `IsStraightLineProgram`.

37.8.2 StraightLineProgram (for a list of lines (and the number of generators))

▷ `StraightLineProgram(`*lines[, nrgens]*`)` (function)
▷ `StraightLineProgram(`*string, gens*`)` (function)
▷ `StraightLineProgramNC(`*lines[, nrgens]*`)` (function)
▷ `StraightLineProgramNC(`*string, gens*`)` (function)

In the first form, *lines* must be a list of lists that defines a unique straight line program (see `IsStraightLineProgram` (37.8.1)); in this case `StraightLineProgram` returns this program, otherwise an error is signalled. The optional argument *nrgens* specifies the number of input generators of the program; if a line of form 1. (that is, a list of integers) occurs in *lines* except in the last position, this number is not determined by *lines* and therefore *must* be specified by the argument *nrgens*; if not then `StraightLineProgram` returns `fail`.

In the second form, *string* must be a string describing an arithmetic expression in terms of the strings in the list *gens*, where multiplication is denoted by concatenation, powering is denoted by `^`, and round brackets `(`, `)` may be used. Each entry in *gens* must consist only of uppercase or lowercase letters (i.e., letters in `IsAlphaChar` (27.4.4)) such that no entry is an initial part of another one. Called with this input, `StraightLineProgram` returns a straight line program that evaluates to the word corresponding to *string* when called with generators corresponding to *gens*.

The `NC` variant does the same, except that the internal consistency of the program is not checked.

37.8.3 LinesOfStraightLineProgram

▷ `LinesOfStraightLineProgram(`*prog*`)` (attribute)

For a straight line program *prog*, `LinesOfStraightLineProgram` returns the list of program lines. There is no default method to compute these lines if they are not stored.

37.8.4 NrInputsOfStraightLineProgram

▷ `NrInputsOfStraightLineProgram(`*prog*`)` (attribute)

For a straight line program *prog*, `NrInputsOfStraightLineProgram` returns the number of generators that are needed as input.

If a line of form 1. (that is, a list of integers) occurs in the lines of *prog* except the last line then the number of generators is not determined by the lines, and must be set in the construction of the straight line program (see `StraightLineProgram` (37.8.2)). So if *prog* contains a line of form 1. other than

the last line and does *not* store the number of generators then `NrInputsOfStraightLineProgram` signals an error.

37.8.5 ResultOfStraightLineProgram

▷ `ResultOfStraightLineProgram(`*prog, gens*`)` (operation)

`ResultOfStraightLineProgram` evaluates the straight line program (see `IsStraightLineProgram` (37.8.1)) *prog* at the group elements in the list *gens*.

The *result* of a straight line program with lines $p_1, p_2, \ldots, p_k$ when applied to *gens* is defined as follows.

(a) First a list r of intermediate results is initialized with a shallow copy of *gens*.

(b) For $i < k$, before the i-th step, let r be of length n. If p_i is the external representation of an associative word in the first n generators then the image of this word under the homomorphism that is given by mapping r to these first n generators is added to r; if p_i is a pair $[l, j]$, for a list l, then the same element is computed, but instead of being added to r, it replaces the j-th entry of r.

(c) For $i = k$, if p_k is the external representation of an associative word then the element described in (b) is the result of the program, if p_k is a pair $[l, j]$, for a list l, then the result is the element described by l, and if p_k is a list $[l_1, l_2, \ldots, l_k]$ of lists then the result is a list of group elements, where each l_i is treated as in (b).

Example

```
gap> f:= FreeGroup( "x", "y" );;  gens:= GeneratorsOfGroup( f );;
gap> x:= gens[1];;  y:= gens[2];;
gap> prg:= StraightLineProgram( [ [] ] );
<straight line program>
gap> ResultOfStraightLineProgram( prg, [] );
[  ]
```

The above straight line program `prg` returns –for *any* list of input generators– an empty list.

Example

```
gap> StraightLineProgram( [ [1,2,2,3], [3,-1] ] );
fail
gap> prg:= StraightLineProgram( [ [1,2,2,3], [3,-1] ], 2 );
<straight line program>
gap> LinesOfStraightLineProgram( prg );
[ [ 1, 2, 2, 3 ], [ 3, -1 ] ]
gap> prg:= StraightLineProgram( "(a^2b^3)^-1", [ "a", "b" ] );
<straight line program>
gap> LinesOfStraightLineProgram( prg );
[ [ [ 1, 2, 2, 3 ], 3 ], [ [ 3, -1 ], 4 ] ]
gap> res:= ResultOfStraightLineProgram( prg, gens );
y^-3*x^-2
gap> res = (x^2 * y^3)^-1;
true
gap> NrInputsOfStraightLineProgram( prg );
2
gap> Print( prg, "\n" );
```

```
StraightLineProgram( [ [ [ 1, 2, 2, 3 ], 3 ], [ [ 3, -1 ], 4 ] ], 2 )
gap> Display( prg );
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[1]^2*r[2]^3;
r[4]:= r[3]^-1;
# return value:
r[4]
gap> IsInternallyConsistent( StraightLineProgramNC( [ [1,2] ] ) );
true
gap> IsInternallyConsistent( StraightLineProgramNC( [ [1,2,3] ] ) );
false
gap> prg1:= StraightLineProgram( [ [1,1,2,2], [3,3,1,1] ], 2 );;
gap> prg2:= StraightLineProgram( [ [ [1,1,2,2], 2 ], [2,3,1,1] ] );;
gap> res1:= ResultOfStraightLineProgram( prg1, gens );
(x*y^2)^3*x
gap> res1 = (x*y^2)^3*x;
true
gap> res2:= ResultOfStraightLineProgram( prg2, gens );
(x*y^2)^3*x
gap> res2 = (x*y^2)^3*x;
true
gap> prg:= StraightLineProgram( [ [2,3], [ [3,1,1,4], [1,2,3,1] ] ], 2 );;
gap> res:= ResultOfStraightLineProgram( prg, gens );
[ y^3*x^4, x^2*y^3 ]
```

37.8.6 StringOfResultOfStraightLineProgram

▷ StringOfResultOfStraightLineProgram(*prog, gensnames[, "LaTeX"]*) (function)

StringOfResultOfStraightLineProgram returns a string that describes the result of the straight line program (see IsStraightLineProgram (37.8.1)) *prog* as word(s) in terms of the strings in the list *gensnames*. If the result of *prog* is a single element then the return value of StringOfResultOfStraightLineProgram is a string consisting of the entries of *gensnames*, opening and closing brackets (and), and powering by integers via ^. If the result of *prog* is a list of elements then the return value of StringOfResultOfStraightLineProgram is a comma separated concatenation of the strings of the single elements, enclosed in square brackets [,].

Example

```
gap> prg:= StraightLineProgram( [ [ 1, 2, 2, 3 ], [ 3, -1 ] ], 2 );;
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ] );
"(a^2b^3)^-1"
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ], "LaTeX" );
"(a^{2}b^{3})^{-1}"
```

37.8.7 CompositionOfStraightLinePrograms

▷ CompositionOfStraightLinePrograms(*prog2, prog1*) (function)

For two straight line programs *prog1* and *prog2*, CompositionOfStraightLinePrograms returns a straight line program *prog* with the properties that *prog1* and *prog* have the same number of inputs, and the result of *prog* when applied to given generators *gens* equals the result of *prog2* when this is applied to the output of *prog1* applied to *gens*.

(Of course the number of outputs of *prog1* must be the same as the number of inputs of *prog2*.)

Example

```
gap> prg1:= StraightLineProgram( "a^2b", [ "a","b" ] );;
gap> prg2:= StraightLineProgram( "c^5", [ "c" ] );;
gap> comp:= CompositionOfStraightLinePrograms( prg2, prg1 );
<straight line program>
gap> StringOfResultOfStraightLineProgram( comp, [ "a", "b" ] );
"(a^2b)^5"
gap> prg:= StraightLineProgram( [ [2,3], [ [3,1,1,4], [1,2,3,1] ] ], 2 );;
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ] );
"[ b^3a^4, a^2b^3 ]"
gap> comp:= CompositionOfStraightLinePrograms( prg, prg );
<straight line program>
gap> StringOfResultOfStraightLineProgram( comp, [ "a", "b" ] );
"[ (a^2b^3)^3(b^3a^4)^4, (b^3a^4)^2(a^2b^3)^3 ]"
```

37.8.8 IntegratedStraightLineProgram

▷ IntegratedStraightLineProgram(*listofprogs*) (function)

For a nonempty dense list *listofprogs* of straight line programs that have the same number *n*, say, of inputs (see NrInputsOfStraightLineProgram (37.8.4)) and for which the results (see ResultOfStraightLineProgram (37.8.5)) are single elements (i.e., *not* lists of elements), IntegratedStraightLineProgram returns a straight line program *prog* with *n* inputs such that for each *n*-tuple *gens* of generators, ResultOfStraightLineProgram(*prog*, *gens*) is equal to the list List(*listofprogs*, *p* -> ResultOfStraightLineProgram(*p*, *gens*).

Example

```
gap> f:= FreeGroup( "x", "y" );;  gens:= GeneratorsOfGroup( f );;
gap> prg1:= StraightLineProgram([ [ [ 1, 2 ], 1 ], [ 1, 2, 2, -1 ] ], 2);;
gap> prg2:= StraightLineProgram([ [ [ 2, 2 ], 3 ], [ 1, 3, 3, 2 ] ], 2);;
gap> prg3:= StraightLineProgram([ [ 2, 2 ], [ 1, 3, 3, 2 ] ], 2);;
gap> prg:= IntegratedStraightLineProgram( [ prg1, prg2, prg3 ] );;
gap> ResultOfStraightLineProgram( prg, gens );
[ x^4*y^-1, x^3*y^4, x^3*y^4 ]
gap> prg:= IntegratedStraightLineProgram( [ prg2, prg3, prg1 ] );;
gap> ResultOfStraightLineProgram( prg, gens );
[ x^3*y^4, x^3*y^4, x^4*y^-1 ]
gap> prg:= IntegratedStraightLineProgram( [ prg3, prg1, prg2 ] );;
gap> ResultOfStraightLineProgram( prg, gens );
[ x^3*y^4, x^4*y^-1, x^3*y^4 ]
```

37.8.9 RestrictOutputsOfSLP

▷ RestrictOutputsOfSLP(*slp*, *k*) (function)

slp must be a straight line program returning a tuple of values. This function returns a new slp that calculates only those outputs specified by *k*. The argument *k* may be an integer or a list of integers. If *k* is an integer, the resulting slp calculates only the result with that number in the original output tuple. If *k* is a list of integers, the resulting slp calculates those results with indices *k* in the original output tuple. In both cases the resulting slp does only what is necessary. Obviously, the slp must have a line with enough expressions (lists) for the supplied *k* as its last line. *slp* is either an slp or a pair where the first entry are the lines of the slp and the second is the number of inputs.

37.8.10 IntermediateResultOfSLP

▷ `IntermediateResultOfSLP(`*slp, k*`)` (function)

Returns a new slp that calculates only the value of slot *k* at the end of *slp* doing only what is necessary. slp is either an slp or a pair where the first entry are the lines of the slp and the second is the number of inputs. Note that this assumes a general SLP with possible overwriting. If you know that your SLP does not overwrite slots, please use `IntermediateResultOfSLPWithoutOverwrite` (37.8.11), which is much faster in this case.

37.8.11 IntermediateResultOfSLPWithoutOverwrite

▷ `IntermediateResultOfSLPWithoutOverwrite(`*slp, k*`)` (function)

Returns a new slp that calculates only the value of slot *k*, which must be an integer. Note that *slp* must not overwrite slots but only append!!! Use `IntermediateResultOfSLP` (37.8.10) in the other case! *slp* is either an slp or a pair where the first entry is the lines of the slp and the second is the number of inputs.

37.8.12 IntermediateResultsOfSLPWithoutOverwrite

▷ `IntermediateResultsOfSLPWithoutOverwrite(`*slp, k*`)` (function)

Returns a new slp that calculates only the value of slots contained in the list k. Note that *slp* must not overwrite slots but only append!!! Use `IntermediateResultOfSLP` (37.8.10) in the other case! *slp* is either a slp or a pair where the first entry is the lines of the slp and the second is the number of inputs.

37.8.13 ProductOfStraightLinePrograms

▷ `ProductOfStraightLinePrograms(`*s1, s2*`)` (function)

s1 and *s2* must be two slps that return a single element with the same number of inputs. This function constructs an slp that returns the product of the two results the slps *s1* and *s2* would produce with the same input.

37.8.14 SlotUsagePattern

▷ `SlotUsagePattern(`*s*`)` (attribute)

Analyses the straight line program *s* for more efficient evaluation. This means in particular two things, when this attribute is known: First of all, intermediate results which are not actually needed later on are not computed at all, and once an intermediate result is used for the last time in this SLP, it is discarded. The latter leads to the fact that the evaluation of the SLP needs less memory.

37.9 Straight Line Program Elements

When computing with very large (in terms of memory) elements, for example permutations of degree a few hundred thousands, it can be helpful (in terms of memory usage) to represent them via straight line programs in terms of an original generator set. (So every element takes only small extra storage for the straight line program.)

A straight line program element has a *seed* (a list of group elements) and a straight line program on the same number of generators as the length of this seed, its value is the value of the evaluated straight line program.

At the moment, the entries of the straight line program have to be simple lists (i.e. of the first form).

Straight line program elements are in the same categories and families as the elements of the seed, so they should work together with existing algorithms.

Note however, that due to the different way of storage some normally very cheap operations (such as testing for element equality) can become more expensive when dealing with straight line program elements. This is essentially the tradeoff for using less memory.

See also Section 43.13.

37.9.1 IsStraightLineProgElm

▷ `IsStraightLineProgElm(`*obj*`)` (Representation)

A straight line program element is a group element given (for memory reasons) as a straight line program. Straight line program elements are positional objects, the first component is a record with a component `seeds`, the second component the straight line program.

37.9.2 StraightLineProgElm

▷ `StraightLineProgElm(`*seed, prog*`)` (function)

Creates a straight line program element for seed *seed* and program *prog*.

37.9.3 StraightLineProgGens

▷ `StraightLineProgGens(`*gens[, base]*`)` (function)

returns a set of straight line program elements corresponding to the generators in *gens*. If *gens* is a set of permutations then *base* can be given which must be a base for the group generated by *gens*. (Such a base will be used to speed up equality tests.)

37.9.4 EvalStraightLineProgElm

▷ `EvalStraightLineProgElm(`*`slpel`*`)` (function)

evaluates a straight line program element *slpel* from its seeds.

37.9.5 StretchImportantSLPElement

▷ `StretchImportantSLPElement(`*`elm`*`)` (operation)

If *elm* is a straight line program element whose straight line representation is very long, this operation changes the representation of *elm* to a straight line program element, equal to *elm*, whose seed contains the evaluation of *elm* and whose straight line program has length 1.

For other objects nothing happens.

This operation permits to designate "important" elements within an algorithm (elements that will be referred to often), which will be represented by guaranteed short straight line program elements.

Example

```
gap> gens:=StraightLineProgGens([(1,2,3,4),(1,2)]);
[ <[ [ 2, 1 ] ]|(1,2,3,4)>, <[ [ 1, 1 ] ]|(1,2)> ]
gap> g:=Group(gens);;
gap> (gens[1]^3)^gens[2];
<[ [ 1, -1, 2, 3, 1, 1 ] ]|(1,2,4,3)>
gap> Size(g);
24
gap> Random(g);
<
[ [ 1, -1, 2, -1, 1, 1, 2, -1, 1, -1, 2, 1, 1, 1, 2, 1, 1, -1, 2, 2,
      1, 1 ],
  [ 3, -2, 2, -2, 1, -1, 2, -2, 1, 1, 2, -1, 1, -1, 2, -2, 1, 1, 2,
      -1, 1, -1, 2, -1, 1, 1, 2, 1, 1, -1, 2, 1, 1, 1 ] ]>
```

Chapter 38

Rewriting Systems

Rewriting systems in GAP are a framework for dealing with the very general task of rewriting elements of a free (or *term*) algebra in some normal form. Although most rewriting systems currently in use are *string rewriting systems* (where the algebra has only one binary operation which is associative) the framework in GAP is general enough to encompass the task of rewriting algebras of any signature from groups to semirings.

Rewriting systems are already implemented in GAP for finitely presented semigroups and for pc groups. The use of these particular rewriting systems is described in the corresponding chapters. We describe here only the general framework of rewriting systems with a particular emphasis on material which would be helpful for a developer implementing a rewriting system.

We fix some definitions and terminology for the rest of this chapter. Let T be a term algebra in some signature. A *term rewriting system* for T is a set of ordered pairs of elements of T of the form (l, r). Viewed as a set of relations, the rewriting system determines a presentation for a quotient algebra A of T.

When we take into account the fact that the relations are expressed as *ordered* pairs, we have a way of *reducing* the elements of T. Suppose an element u of T has a subword l and (l, r) is a rule of the rewriting system, then we can replace the subterm l of u by the term r and obtain a new word v. We say that we have *rewritten* u as v. Note that u and v represent the same element of A. If u can not be rewritten using any rule of the rewriting system we sat that u is *reduced*.

38.1 Operations on rewriting systems

38.1.1 IsRewritingSystem

▷ `IsRewritingSystem(obj)` (Category)

This is the category in which all rewriting systems lie.

38.1.2 Rules

▷ `Rules(rws)` (attribute)

The rules comprising the rewriting system. Note that these may change through the life of the rewriting system, however they will always be a set of defining relations of the algebra described by the rewriting system.

38.1.3 OrderOfRewritingSystem

▷ `OrderOfRewritingSystem(rws)` (attribute)
▷ `OrderingOfRewritingSystem(rws)` (attribute)

return the ordering of the rewriting system `rws`.

38.1.4 ReducedForm

▷ `ReducedForm(rws, u)` (operation)

Given an element `u` in the free (or term) algebra T over which `rws` is defined, rewrite `u` by successive applications of the rules of `rws` until no further rewriting is possible, and return the resulting element of T.

38.1.5 IsConfluent

▷ `IsConfluent(rws)` (property)
▷ `IsConfluent(A)` (property)

For a rewriting system `rws`, `IsConfluent` returns `true` if and only if `rws` is confluent. A rewriting system is *confluent* if, for every two words u and v in the free algebra T which represent the same element of the algebra A defined by `rws`, `ReducedForm( rws , u ) = ReducedForm( rws , v)` as words in the free algebra T. This element is the *unique normal form* of the element represented by u.

For an algebra `A` with a canonical rewriting system associated with it, `IsConfluent` checks whether that rewriting system is confluent.

Also see `IsConfluent` (46.4.7).

38.1.6 ConfluentRws

▷ `ConfluentRws(rws)` (attribute)

Return a new rewriting system defining the same algebra as `rws` which is confluent.

38.1.7 IsReduced

▷ `IsReduced(rws)` (property)

A rewriting system is reduced if for each rule (l, r), l and r are both reduced.

38.1.8 ReduceRules

▷ `ReduceRules(rws)` (operation)

Reduce rules and remove redundant rules to make `rws` reduced.

38.1.9 AddRule

▷ AddRule(*rws*, *rule*) (operation)

Add *rule* to a rewriting system *rws*.

38.1.10 AddRuleReduced

▷ AddRuleReduced(*rws*, *rule*) (operation)

Add *rule* to rewriting system *rws*. Performs a reduction operation on the resulting system, so that if *rws* is reduced it will remain reduced.

38.1.11 MakeConfluent

▷ MakeConfluent(*rws*) (operation)

Add rules (and perhaps reduce) in order to make *rws* confluent

38.1.12 GeneratorsOfRws

▷ GeneratorsOfRws(*rws*) (attribute)

Returns the list of generators of the rewriting system *rws*.

38.2 Operations on elements of the algebra

In this section let u denote an element of the term algebra T representing $[u]$ in the quotient algebra A.

38.2.1 ReducedProduct

▷ ReducedProduct(*rws*, *u*, *v*) (operation)
▷ ReducedSum(*rws*, *left*, *right*) (operation)
▷ ReducedOne(*rws*) (operation)
▷ ReducedAdditiveInverse(*rws*, *obj*) (operation)
▷ ReducedComm(*rws*, *left*, *right*) (operation)
▷ ReducedConjugate(*rws*, *left*, *right*) (operation)
▷ ReducedDifference(*rws*, *left*, *right*) (operation)
▷ ReducedInverse(*rws*, *obj*) (operation)
▷ ReducedLeftQuotient(*rws*, *left*, *right*) (operation)
▷ ReducedPower(*rws*, *obj*, *pow*) (operation)
▷ ReducedQuotient(*rws*, *left*, *right*) (operation)
▷ ReducedScalarProduct(*rws*, *left*, *right*) (operation)
▷ ReducedZero(*rws*) (operation)

The result of ReducedProduct is w where $[w]$ equals $[u][v]$ in A and w is in reduced form.

The remaining operations are defined similarly when they are defined (as determined by the signature of the term algebra).

38.3 Properties of rewriting systems

38.3.1 IsBuiltFromAdditiveMagmaWithInverses

▷ IsBuiltFromAdditiveMagmaWithInverses(*obj*) (property)
▷ IsBuiltFromMagma(*obj*) (property)
▷ IsBuiltFromMagmaWithOne(*obj*) (property)
▷ IsBuiltFromMagmaWithInverses(*obj*) (property)
▷ IsBuiltFromSemigroup(*obj*) (property)
▷ IsBuiltFromGroup(*obj*) (property)

These properties may be used to identify the type of term algebra over which the rewriting system is defined.

38.4 Rewriting in Groups and Monoids

One application of rewriting is to reduce words in finitely presented groups and monoids. The rewriting system still has to be built for a finitely presented monoid (using IsomorphismFpMonoid for conversion). Rewriting then can take place for words in the underlying free monoid. (These can be obtained from monoid elements with the command UnderlyingElement.)

Example

```
gap> f:=FreeGroup(3);;
gap> rels:=[f.1*f.2^2/f.3,f.2*f.3^2/f.1,f.3*f.1^2/f.2];;
gap> g:=f/rels;
<fp group on the generators [ f1, f2, f3 ]>
gap> mhom:=IsomorphismFpMonoid(g);
MappingByFunction( <fp group on the generators
[ f1, f2, f3 ]>, <fp monoid on the generators
[ f1, f1^-1, f2, f2^-1, f3, f3^-1
 ]>, function( x ) ... end, function( x ) ... end )
gap> mon:=Image(mhom);
<fp monoid on the generators [ f1, f1^-1, f2, f2^-1, f3, f3^-1 ]>
gap> k:=KnuthBendixRewritingSystem(mon);
Knuth Bendix Rewriting System for Monoid(
[ f1, f1^-1, f2, f2^-1, f3, f3^-1 ] ) with rules
[ [ f1*f1^-1, <identity ...> ], [ f1^-1*f1, <identity ...> ],
  [ f2*f2^-1, <identity ...> ], [ f2^-1*f2, <identity ...> ],
  [ f3*f3^-1, <identity ...> ], [ f3^-1*f3, <identity ...> ],
  [ f1*f2^2*f3^-1, <identity ...> ], [ f2*f3^2*f1^-1, <identity ...> ]
    , [ f3*f1^2*f2^-1, <identity ...> ] ]
gap> MakeConfluent(k);
gap> a:=Product(GeneratorsOfMonoid(mon));
f1*f1^-1*f2*f2^-1*f3*f3^-1
gap> ReducedForm(k,UnderlyingElement(a));
<identity ...>
```

To rewrite a word in the finitely presented group, one has to convert it to a word in the monoid first, rewrite in the underlying free monoid and convert back (by forming first again an element of the fp monoid) to the finitely presented group.

Example

```
gap> r:=PseudoRandom(g);;
gap> Length(r);
3704
gap> melm:=Image(mhom,r);;
gap> red:=ReducedForm(k,UnderlyingElement(melm));
f1^-1^3*f2^-1*f1^2
gap> melm:=ElementOfFpMonoid(FamilyObj(One(mon)),red);
f1^-1^3*f2^-1*f1^2
gap> gpelm:=PreImagesRepresentative(mhom,melm);
f1^-3*f2^-1*f1^2
gap> r=gpelm;
true
gap> CategoriesOfObject(red);
[ "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsAssociativeElement", "IsWord" ]
gap> CategoriesOfObject(melm);
[ "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsAssociativeElement",
  "IsElementOfFpMonoid" ]
gap> CategoriesOfObject(gpelm);
[ "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsMultiplicativeElementWithInverse",
  "IsAssociativeElement", "IsElementOfFpGroup" ]
```

Note, that the elements `red` (free monoid) `melm` (fp monoid) and `gpelm` (group) differ, though they are displayed identically.

Under Unix, it is possible to use the kbmag package to replace the built-in rewriting by this packages efficient C implementation. You can do this (after loading the kbmag package) by assigning the variable `KB_REW` (52.6.2) to `KBMAG_REW`. Assignment to `GAPKB_REW` reverts to the built-in implementation.

Example

```
gap> LoadPackage("kbmag");
true
gap> KB_REW:=KBMAG_REW;;
```

38.5 Developing rewriting systems

The key point to note about rewriting systems is that they have properties such as `IsConfluent` (38.1.5) and attributes such as `Rules` (38.1.2), however they are rarely stored, but rather computed afresh each time they are asked for, from data stored in the private members of the rewriting system object. This is because a rewriting system often evolves through a session, starting with some rules which define the algebra *A* as relations, and then adding more rules to make the system confluent. For example, in the case of Knuth-Bendix rewriting systems (see Chapter 52), the function `CreateKnuthBendixRewritingSystem` creating the rewriting system (in the file `lib/kbsemi.gi`) uses

Example

```
kbrws := Objectify(NewType(rwsfam,
  IsMutable and IsKnuthBendixRewritingSystem and
```

```
    IsKnuthBendixRewritingSystemRep),
    rec(family:= fam,
    reduced:=false,
    tzrules:=List(relwco,i->
     [LetterRepAssocWord(i[1]),LetterRepAssocWord(i[2])]),
    pairs2check:=CantorList(Length(r)),
    ordering:=wordord,
    freefam:=freefam));
```

In particular, since we don't use the filter `IsAttributeStoringRep` in the `Objectify` (79.9.1), whenever `IsConfluent` (38.1.5) is called, the appropriate method to determine confluence is called.

Chapter 39

Groups

This chapter explains how to create groups and defines operations for groups, that is operations whose definition does not depend on the representation used. However methods for these operations in most cases will make use of the representation.

If not otherwise specified, in all examples in this chapter the group g will be the symmetric group S_4 acting on the letters $\{1,\ldots,4\}$.

39.1 Group Elements

Groups in GAP are written multiplicatively. The elements from which a group can be generated must permit multiplication and multiplicative inversion (see 31.14).

Example

```
gap> a:=(1,2,3);;b:=(2,3,4);;
gap> One(a);
()
gap> Inverse(b);
(2,4,3)
gap> a*b;
(1,3)(2,4)
gap> Order(a*b);
2
gap> Order( [ [ 1, 1 ], [ 0, 1 ] ] );
infinity
```

The next example may run into an infinite loop because the given matrix in fact has infinite order.

Example

```
gap> Order( [ [ 1, 1 ], [ 0, 1 ] ] * Indeterminate( Rationals ) );
#I  Order: warning, order of <mat> might be infinite
```

Since groups are domains, the recommended command to compute the order of a group is `Size` (30.4.6). For convenience, group orders can also be computed with `Order` (31.10.10).

The operation `Comm` (31.12.3) can be used to compute the commutator of two elements, the operation `LeftQuotient` (31.12.2) computes the product $x^{-1}y$.

39.2 Creating Groups

When groups are created from generators, this means that the generators must be elements that can be multiplied and inverted (see also 31.3). For creating a free group on a set of symbols, see FreeGroup (37.2.1).

39.2.1 Group (for several generators)

▷ Group(*gen, ...*) (function)
▷ Group(*gens[, id]*) (function)

Group(*gen* , ...) is the group generated by the arguments *gen*, ...

If the only argument *gens* is a list that is not a matrix then Group(*gens*) is the group generated by the elements of that list.

If there are two arguments, a list *gens* and an element *id*, then Group(*gens* , *id*) is the group generated by the elements of *gens*, with identity *id*.

Note that the value of the attribute GeneratorsOfGroup (39.2.4) need not be equal to the list *gens* of generators entered as argument. Use GroupWithGenerators (39.2.3) if you want to be sure that the argument *gens* is stored as value of GeneratorsOfGroup (39.2.4).

Example

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
```

39.2.2 GroupByGenerators

▷ GroupByGenerators(*gens*) (operation)
▷ GroupByGenerators(*gens, id*) (operation)

GroupByGenerators returns the group G generated by the list *gens*. If a second argument *id* is present then this is stored as the identity element of the group.

The value of the attribute GeneratorsOfGroup (39.2.4) of G need not be equal to *gens*. GroupByGenerators is the underlying operation called by Group (39.2.1).

39.2.3 GroupWithGenerators

▷ GroupWithGenerators(*gens[, id]*) (operation)

GroupWithGenerators returns the group G generated by the list *gens*. If a second argument *id* is present then this is stored as the identity element of the group. The value of the attribute GeneratorsOfGroup (39.2.4) of G is equal to *gens*.

39.2.4 GeneratorsOfGroup

▷ GeneratorsOfGroup(*G*) (attribute)

returns a list of generators of the group *G*. If *G* has been created by the command GroupWithGenerators (39.2.3) with argument *gens*, then the list returned by GeneratorsOfGroup will be equal to *gens*. For such a group, each generator can also be accessed using the . operator

(see GeneratorsOfDomain (31.9.2)): for a positive integer *i*, *G*.i returns the *i*-th element of the list returned by GeneratorsOfGroup. Moreover, if *G* is a free group, and name is the name of a generator of *G* then *G*.name also returns this generator.

Example

```
gap> g:=GroupWithGenerators([(1,2,3,4),(1,2)]);
Group([ (1,2,3,4), (1,2) ])
gap> GeneratorsOfGroup(g);
[ (1,2,3,4), (1,2) ]
```

While in this example GAP displays the group via the generating set stored in the attribute GeneratorsOfGroup, the methods installed for View (6.3.3) will in general display only some information about the group which may even be just the fact that it is a group.

39.2.5 AsGroup

▷ AsGroup(*D*) (attribute)

if the elements of the collection *D* form a group the command returns this group, otherwise it returns fail.

Example

```
gap> AsGroup([(1,2)]);
fail
gap> AsGroup([(),(1,2)]);
Group([ (1,2) ])
```

39.2.6 ConjugateGroup

▷ ConjugateGroup(*G, obj*) (operation)

returns the conjugate group of *G*, obtained by applying the conjugating element *obj*.

To form a conjugate (group) by any object acting via ^, one can also use the infix operator ^.

Example

```
gap> ConjugateGroup(g,(1,5));
Group([ (2,3,4,5), (2,5) ])
```

39.2.7 IsGroup

▷ IsGroup(*obj*) (Category)

A group is a magma-with-inverses (see IsMagmaWithInverses (35.1.4)) and associative (see IsAssociative (35.4.7)) multiplication.

IsGroup tests whether the object *obj* fulfills these conditions, it does *not* test whether *obj* is a set of elements that forms a group under multiplication; use AsGroup (39.2.5) if you want to perform such a test. (See 13.3 for details about categories.)

Example

```
gap> IsGroup(g);
true
```

39.2.8 InfoGroup

▷ `InfoGroup` (info class)

is the info class for the generic group theoretic functions (see 7.4).

39.3 Subgroups

For the general concept of parents and subdomains, see 31.7 and 31.8. More functions that construct certain subgroups can be found in the sections 39.11, 39.12, 39.13, and 39.14.

If a group U is created as a subgroup of another group G, G becomes the parent of U. There is no "universal" parent group, parent-child chains can be arbitrary long. GAP stores the result of some operations (such as `Normalizer` (39.11.1)) with the parent as an attribute.

39.3.1 Subgroup

▷ `Subgroup(`*G, gens*`)` (function)
▷ `SubgroupNC(`*G, gens*`)` (function)
▷ `Subgroup(`*G*`)` (function)

creates the subgroup *U* of *G* generated by *gens*. The `Parent` (31.7.1) value of *U* will be *G*. The `NC` version does not check, whether the elements in *gens* actually lie in *G*.

The unary version of `Subgroup` creates a (shell) subgroup that does not even know generators but can be used to collect information about a particular subgroup over time.

Example

```
gap> u:=Subgroup(g,[(1,2,3),(1,2)]);
Group([ (1,2,3), (1,2) ])
```

39.3.2 Index (GAP operation)

▷ `Index(`*G, U*`)` (operation)
▷ `IndexNC(`*G, U*`)` (operation)

For a subgroup *U* of the group *G*, `Index` returns the index $[G : U] = |G|/|U|$ of *U* in *G*. The `NC` version does not test whether *U* is contained in *G*.

Example

```
gap> Index(g,u);
4
```

39.3.3 IndexInWholeGroup

▷ `IndexInWholeGroup(`*G*`)` (attribute)

If the family of elements of *G* itself forms a group *P*, this attribute returns the index of *G* in *P*. It is used primarily for free groups or finitely presented groups.

Example
```
gap> freegp:=FreeGroup(1);;
gap> freesub:=Subgroup(freegp,[freegp.1^5]);;
gap> IndexInWholeGroup(freesub);
5
```

39.3.4 AsSubgroup

▷ AsSubgroup(*G*, *U*) (operation)

creates a subgroup of *G* which contains the same elements as *U*

Example
```
gap> v:=AsSubgroup(g,Group((1,2,3),(1,4)));
Group([ (1,2,3), (1,4) ])
gap> Parent(v);
Group([ (1,2,3,4), (1,2) ])
```

39.3.5 IsSubgroup

▷ IsSubgroup(*G*, *U*) (function)

IsSubgroup returns true if *U* is a group that is a subset of the domain *G*. This is actually checked by calling IsGroup(*U*) and IsSubset(*G*, *U*); note that special methods for IsSubset (30.5.1) are available that test only generators of *U* if *G* is closed under the group operations. So in most cases, for example whenever one knows already that *U* is a group, it is better to call only IsSubset (30.5.1).

Example
```
gap> IsSubgroup(g,u);
true
gap> v:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> u=v;
true
gap> IsSubgroup(g,v);
true
```

39.3.6 IsNormal

▷ IsNormal(*G*, *U*) (operation)

returns true if the group *G* normalizes the group *U* and false otherwise.

A group *G* *normalizes* a group *U* if and only if for every $g \in G$ and $u \in U$ the element u^g is a member of *U*. Note that *U* need not be a subgroup of *G*.

Example
```
gap> IsNormal(g,u);
false
```

39.3.7 IsCharacteristicSubgroup

▷ IsCharacteristicSubgroup(*G, N*) (operation)

tests whether *N* is invariant under all automorphisms of *G*.

Example

```
gap> IsCharacteristicSubgroup(g,u);
false
```

39.3.8 ConjugateSubgroup

▷ ConjugateSubgroup(*G, g*) (operation)

For a group *G* which has a parent group P (see Parent (31.7.1)), returns the subgroup of P, obtained by conjugating *G* using the conjugating element *g*.

If *G* has no parent group, it just delegates to the call to ConjugateGroup (39.2.6) with the same arguments.

To form a conjugate (subgroup) by any object acting via ^, one can also use the infix operator ^.

39.3.9 ConjugateSubgroups

▷ ConjugateSubgroups(*G, U*) (operation)

returns a list of all images of the group *U* under conjugation action by *G*.

39.3.10 IsSubnormal

▷ IsSubnormal(*G, U*) (operation)

A subgroup *U* of the group *G* is subnormal if it is contained in a subnormal series of *G*.

Example

```
gap> IsSubnormal(g,Group((1,2,3)));
false
gap> IsSubnormal(g,Group((1,2)(3,4)));
true
```

39.3.11 SubgroupByProperty

▷ SubgroupByProperty(*G, prop*) (function)

creates a subgroup of *G* consisting of those elements fulfilling *prop* (which is a tester function). No test is done whether the property actually defines a subgroup.

Note that currently very little functionality beyond an element test exists for groups created this way.

39.3.12 SubgroupShell

▷ SubgroupShell(*G*) (function)

creates a subgroup of *G* which at this point is not yet specified further (but will be later, for example by assigning a generating set).

Example

```
gap> u:=SubgroupByProperty(g,i->3^i=3);
<subgrp of Group([ (1,2,3,4), (1,2) ]) by property>
gap> (1,3) in u; (1,4) in u; (1,5) in u;
false
true
false
gap> GeneratorsOfGroup(u);
[ (1,2), (1,4,2) ]
gap> u:=SubgroupShell(g);
<group>
```

39.4 Closures of (Sub)groups

39.4.1 ClosureGroup

▷ ClosureGroup(*G, obj*) (operation)

creates the group generated by the elements of *G* and *obj*. *obj* can be either an element or a collection of elements, in particular another group.

Example

```
gap> g:=SmallGroup(24,12);;u:=Subgroup(g,[g.3,g.4]);
Group([ f3, f4 ])
gap> ClosureGroup(u,g.2);
Group([ f2, f3, f4 ])
gap> ClosureGroup(u,[g.1,g.2]);
Group([ f1, f2, f3, f4 ])
gap> ClosureGroup(u,Group(g.2*g.1));
Group([ f1*f2^2, f3, f4 ])
```

39.4.2 ClosureGroupAddElm

▷ ClosureGroupAddElm(*G, elm*) (function)
▷ ClosureGroupCompare(*G, elm*) (function)
▷ ClosureGroupIntest(*G, elm*) (function)

These three functions together with ClosureGroupDefault (39.4.3) implement the main methods for ClosureGroup (39.4.1). In the ordering given, they just add *elm* to the generators, remove duplicates and identity elements, and test whether *elm* is already contained in *G*.

39.4.3 ClosureGroupDefault

▷ ClosureGroupDefault(*G, elm*) (function)

This functions returns the closure of the group *G* with the element *elm*. If *G* has the attribute AsSSortedList (30.3.10) then also the result has this attribute. This is used to implement the default method for Enumerator (30.3.2) and EnumeratorSorted (30.3.3).

39.4.4 ClosureSubgroup

▷ ClosureSubgroup(*G, obj*) (function)
▷ ClosureSubgroupNC(*G, obj*) (function)

For a group *G* that stores a parent group (see 31.7), ClosureSubgroup calls ClosureGroup (39.4.1) with the same arguments; if the result is a subgroup of the parent of *G* then the parent of *G* is set as parent of the result, otherwise an error is raised. The check whether the result is contained in the parent of *G* is omitted by the NC version. As a wrong parent might imply wrong properties this version should be used with care.

39.5 Expressing Group Elements as Words in Generators

Using homomorphisms (see chapter 40) is is possible to express group elements as words in given generators: Create a free group (see FreeGroup (37.2.1)) on the correct number of generators and create a homomorphism from this free group onto the group *G* in whose generators you want to factorize. Then the preimage of an element of *G* is a word in the free generators, that will map on this element again.

39.5.1 EpimorphismFromFreeGroup

▷ EpimorphismFromFreeGroup(*G*) (attribute)

For a group *G* with a known generating set, this attribute returns a homomorphism from a free group that maps the free generators to the groups generators.

The option names can be used to prescribe a (print) name for the free generators.

The following example shows how to decompose elements of S_4 in the generators (1,2,3,4) and (1,2):

Example

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> hom:=EpimorphismFromFreeGroup(g:names:=["x","y"]);
[ x, y ] -> [ (1,2,3,4), (1,2) ]
gap> PreImagesRepresentative(hom,(1,4));
y^-1*x^-1*(x^-1*y^-1)^2*x
```

The following example stems from a real request to the GAP Forum. In September 2000 a GAP user working with puzzles wanted to express the permutation (1,2) as a word as short as possible in particular generators of the symmetric group S_{16}.

Example

```
gap> perms := [ (1,2,3,7,11,10,9,5), (2,3,4,8,12,11,10,6),
>   (5,6,7,11,15,14,13,9), (6,7,8,12,16,15,14,10) ];;
gap> puzzle := Group( perms );;Size( puzzle );
20922789888000
```

```
gap> hom:=EpimorphismFromFreeGroup(puzzle:names:=["a", "b", "c", "d"]);;
gap> word := PreImagesRepresentative( hom, (1,2) );
a^-1*c*b*c^-1*a*b^-1*a^-2*c^-1*a*b^-1*c*b
gap> Length( word );
13
```

39.5.2 Factorization

▷ `Factorization(G, elm)` (operation)

returns a factorization of *elm* as word in the generators of the group *G* given in the attribute GeneratorsOfGroup (39.2.4). The attribute EpimorphismFromFreeGroup (39.5.1) of *G* will contain a map from the group *G* to the free group in which the word is expressed. The attribute MappingGeneratorsImages (40.10.2) of this map gives a list of generators and corresponding letters.

The algorithm used forms all elements of the group to ensure a short word is found. Therefore this function should *not* be used when the group *G* has more than a few million elements. Because of this, one should not call this function within algorithms, but use homomorphisms instead.

Example

```
gap> G:=SymmetricGroup( 6 );;
gap> r:=(3,4);; s:=(1,2,3,4,5,6);;
gap> # create subgroup to force the system to use the generators r and s:
gap> H:= Subgroup(G, [ r, s ] );
Group([ (3,4), (1,2,3,4,5,6) ])
gap> Factorization( H, (1,2,3) );
(x2*x1)^2*x2^-2
gap> s*r*s*r*s^-2;
(1,2,3)
gap> MappingGeneratorsImages(EpimorphismFromFreeGroup(H));
[ [ x1, x2 ], [ (3,4), (1,2,3,4,5,6) ] ]
```

39.5.3 GrowthFunctionOfGroup

▷ `GrowthFunctionOfGroup(G)` (operation)

▷ `GrowthFunctionOfGroup(G, radius)` (operation)

For a group *G* with a generating set given in GeneratorsOfGroup (39.2.4), this function calculates the number of elements whose shortest expression as words in the generating set is of a particular length. It returns a list *L*, whose $i+1$ entry counts the number of elements whose shortest word expression has length i. If a maximal length *radius* is given, only words up to length *radius* are counted. Otherwise the group must be finite and all elements are enumerated.

Example

```
gap> GrowthFunctionOfGroup(MathieuGroup(12));
[ 1, 5, 19, 70, 255, 903, 3134, 9870, 25511, 38532, 16358, 382 ]
gap> GrowthFunctionOfGroup(MathieuGroup(12),2);
[ 1, 5, 19 ]
gap> GrowthFunctionOfGroup(MathieuGroup(12),99);
[ 1, 5, 19, 70, 255, 903, 3134, 9870, 25511, 38532, 16358, 382 ]
gap> free:=FreeGroup("a","b");
```

```
<free group on the generators [ a, b ]>
gap> product:=free/ParseRelators(free,"a2,b3");
<fp group on the generators [ a, b ]>
gap> SetIsFinite(product,false);
gap> GrowthFunctionOfGroup(product,10);
[ 1, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64 ]
```

39.6 Structure Descriptions

39.6.1 StructureDescription

▷ `StructureDescription(`*G*`)` (attribute)

The method for `StructureDescription` exhibits a structure of the given group *G* to some extent, using the strategy outlined below. The idea is to return a possibly short string which gives some insight in the structure of the considered group. It is intended primarily for small groups (order less than 100) or groups with few normal subgroups, in other cases, in particular large p-groups, it can be very costly. Furthermore, the string returned is – as the action on chief factors is not described – often not the most useful way to describe a group.

The string returned by `StructureDescription` is NOT an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings. The value returned by `StructureDescription` is a string of the following form:

```
StructureDescription(<G>) ::=
    1                                   ; trivial group
  | C<size>                             ; cyclic group
  | A<degree>                           ; alternating group
  | S<degree>                           ; symmetric group
  | D<size>                             ; dihedral group
  | Q<size>                             ; quaternion group
  | QD<size>                            ; quasidihedral group
  | PSL(<n>,<q>)                        ; projective special linear group
  | SL(<n>,<q>)                         ; special linear group
  | GL(<n>,<q>)                         ; general linear group
  | PSU(<n>,<q>)                        ; proj. special unitary group
  | O(2<n>+1,<q>)                       ; orthogonal group, type B
  | O+(2<n>,<q>)                        ; orthogonal group, type D
  | O-(2<n>,<q>)                        ; orthogonal group, type 2D
  | PSp(2<n>,<q>)                       ; proj. special symplectic group
  | Sz(<q>)                             ; Suzuki group
  | Ree(<q>)                            ; Ree group (type 2F or 2G)
  | E(6,<q>) | E(7,<q>) | E(8,<q>)      ; Lie group of exceptional type
  | 2E(6,<q>) | F(4,<q>) | G(2,<q>)
  | 3D(4,<q>)                           ; Steinberg triality group
  | M11 | M12 | M22 | M23 | M24
  | J1 | J2 | J3 | J4 | Co1 | Co2
  | Co3 | Fi22 | Fi23 | Fi24' | Suz
  | HS | McL | He | HN | Th | B
  | M | ON | Ly | Ru                   ; sporadic simple group
```

```
    | 2F(4,2)'                          ; Tits group
    | PerfectGroup(<size>,<id>)         ; the indicated group from the
                                        ; library of perfect groups
    | A x B                             ; direct product
    | N : H                             ; semidirect product
    | C(G) . G/C(G) = G' . G/G'         ; non-split extension
                                        ; (equal alternatives and
                                        ; trivial extensions omitted)
    | Phi(G) . G/Phi(G)                 ; non-split extension:
                                        ; Frattini subgroup and
                                        ; Frattini factor group
```

Note that the `StructureDescription` is only *one* possible way of building up the given group from smaller pieces.

The option "short" is recognized - if this option is set, an abbreviated output format is used (e.g. `"6x3"` instead of `"C6 x C3"`).

If the `Name` (12.8.2) attribute is not bound, but `StructureDescription` is, `View` (6.3.3) prints the value of the attribute `StructureDescription`. The `Print` (6.3.4)ed representation of a group is not affected by computing a `StructureDescription`.

The strategy used to compute a `StructureDescription` is as follows:

1. Lookup in a precomputed list, if the order of `G` is not larger than 100 and not equal to 64.

2. If `G` is abelian, then decompose it into cyclic factors in "elementary divisors style". For example, `"C2 x C3 x C3"` is `"C6 x C3"`.

3. Recognize alternating groups, symmetric groups, dihedral groups, quasidihedral groups, quaternion groups, PSL's, SL's, GL's and simple groups not listed so far as basic building blocks.

4. Decompose `G` into a direct product of irreducible factors.

5. Recognize semidirect products `G`=N:H, where N is normal. Select a pair N, H with the following preferences:

- **1.** H is abelian
- **2.** N is abelian
- **2a.** N has many abelian invariants
- **3.** N is a direct product
- **3a.** N has many direct factors
- **4.** $\phi : H \to \mathrm{Aut}(N)$, $h \mapsto (n \mapsto n^h)$ is injective.

6. Fall back to non-splitting extensions: If the centre or the commutator factor group is non-trivial, write `G` as Z(`G`).`G`/Z(`G`) or `G'`.`G`/`G'`, respectively. Otherwise if the Frattini subgroup is non-trivial, write `G` as Φ(`G`).`G`/Φ(`G`).

7. If no decomposition is found (maybe this is not the case for any finite group), try to identify `G` in the perfect groups library. If this fails also, then return a string describing this situation.

Note that `StructureDescription` is *not* intended to be a research tool, but rather an educational tool. The reasons for this are as follows:

1. "Most" groups do not have "nice" decompositions. This is in some contrast to what is often taught in elementary courses on group theory, where it is sometimes suggested that basically every group can be written as iterated direct or semidirect product of cyclic groups and non-abelian simple groups.

2. In particular many p-groups have very "similar" structure, and `StructureDescription` can only exhibit a little of it. Changing this would likely make the output not essentially easier to read than a pc presentation.

Example

```
gap> l := AllSmallGroups(12);;
gap> List(l,StructureDescription);; l;
[ C3 : C4, C12, A4, D12, C6 x C2 ]
gap> List(AllSmallGroups(40),G->StructureDescription(G:short));
[ "5:8", "40", "5:8", "5:Q8", "4xD10", "D40", "2x(5:4)", "(10x2):2",
  "20x2", "5xD8", "5xQ8", "2x(5:4)", "2^2xD10", "10x2^2" ]
gap> List(AllTransitiveGroups(DegreeAction,6),
>         G->StructureDescription(G:short));
[ "6", "S3", "D12", "A4", "3xS3", "2xA4", "S4", "S4", "S3xS3",
  "(3^2):4", "2xS4", "A5", "(S3xS3):2", "S5", "A6", "S6" ]
gap> StructureDescription(PSL(4,2));
"A8"
```

39.7 Cosets

39.7.1 RightCoset

▷ `RightCoset(`*U*`, `*g*`)` (operation)

returns the right coset of U with representative g, which is the set of all elements of the form ug for all $u \in U$. g must be an element of a larger group G which contains U. For element operations such as `in` a right coset behaves like a set of group elements.

Right cosets are external orbits for the action of U which acts via `OnLeftInverse` (41.2.3). Of course the action of a larger group G on right cosets is via `OnRight` (41.2.2).

Example

```
gap> u:=Group((1,2,3), (1,2));;
gap> c:=RightCoset(u,(2,3,4));
RightCoset(Group( [ (1,2,3), (1,2) ] ),(2,3,4))
gap> ActingDomain(c);
Group([ (1,2,3), (1,2) ])
gap> Representative(c);
(2,3,4)
gap> Size(c);
6
gap> AsList(c);
[ (2,3,4), (1,4,2), (1,3,4,2), (1,3)(2,4), (2,4), (1,4,2,3) ]
```

39.7.2 RightCosets

▷ `RightCosets(`*G*`, `*U*`)` (function)

▷ `RightCosetsNC(`*G*`, `*U*`)` (operation)

computes a duplicate free list of right cosets $U\,g$ for $g \in G$. A set of representatives for the elements in this list forms a right transversal of U in G. (By inverting the representatives one obtains a list of representatives of the left cosets of U.) The NC version does not check whether U is a subgroup of G.

Example

```
gap> RightCosets(g,u);
[ RightCoset(Group( [ (1,2,3), (1,2) ] ),()),
  RightCoset(Group( [ (1,2,3), (1,2) ] ),(1,3)(2,4)),
  RightCoset(Group( [ (1,2,3), (1,2) ] ),(1,4)(2,3)),
  RightCoset(Group( [ (1,2,3), (1,2) ] ),(1,2)(3,4)) ]
```

39.7.3 CanonicalRightCosetElement

▷ `CanonicalRightCosetElement(U, g)` (operation)

returns a "canonical" representative of the right coset $U\,g$ which is independent of the given representative g. This can be used to compare cosets by comparing their canonical representatives.

The representative chosen to be the "canonical" one is representation dependent and only guaranteed to remain the same within one GAP session.

Example

```
gap> CanonicalRightCosetElement(u,(2,4,3));
(3,4)
```

39.7.4 IsRightCoset

▷ `IsRightCoset(obj)` (Category)

The category of right cosets.

GAP does not provide left cosets as a separate data type, but as the left coset gU consists of exactly the inverses of the elements of the right coset Ug^{-1} calculations with left cosets can be emulated using right cosets by inverting the representatives.

39.7.5 CosetDecomposition

▷ `CosetDecomposition(G, S)` (function)

For a finite group G and a subgroup $S \leq G$ this function returns a partition of the elements of G according to the (right) cosets of S. The result is a list of lists, each sublist corresponding to one coset. The first sublist is the elements list of the subgroup, the other lists are arranged accordingly.

Example

```
gap> CosetDecomposition(SymmetricGroup(4),SymmetricGroup(3));
[ [ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ],
  [ (1,4), (1,4)(2,3), (1,2,4), (1,2,3,4), (1,3,2,4), (1,3,4) ],
  [ (1,4,2), (1,4,2,3), (2,4), (2,3,4), (1,3)(2,4), (1,3,4,2) ],
  [ (1,4,3), (1,4,3,2), (1,2,4,3), (1,2)(3,4), (2,4,3), (3,4) ] ]
```

39.8 Transversals

39.8.1 RightTransversal

▷ RightTransversal(*G, U*) (operation)

A right transversal t is a list of representatives for the set $U \setminus G$ of right cosets (consisting of cosets Ug) of U in G.

The object returned by RightTransversal is not a plain list, but an object that behaves like an immutable list of length $[G : U]$, except if U is the trivial subgroup of G in which case RightTransversal may return the sorted plain list of coset representatives.

The operation PositionCanonical (21.16.3), called for a transversal t and an element g of G, will return the position of the representative in t that lies in the same coset of U as the element g does. (In comparison, Position (21.16.1) will return fail if the element is not equal to the representative.) Functions that implement group actions such as Action (41.7.2) or Permutation (41.9.1) (see Chapter 41) use PositionCanonical (21.16.3), therefore it is possible to "act" on a right transversal to implement the action on the cosets. This is often much more efficient than acting on cosets.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> u:=Subgroup(g,[(1,2,3),(1,2)]);;
gap> rt:=RightTransversal(g,u);
RightTransversal(Group([ (1,2,3,4), (1,2) ]),Group([ (1,2,3), (1,2) ]))
gap> Length(rt);
4
gap> Position(rt,(1,2,3));
fail
```

Note that the elements of a right transversal are not necessarily "canonical" in the sense of CanonicalRightCosetElement (39.7.3), but we may compute a list of canonical coset representatives by calling that function. (See also PositionCanonical (21.16.3).)

Example

```
gap> List(RightTransversal(g,u),i->CanonicalRightCosetElement(u,i));
[ (), (2,3,4), (1,2,3,4), (3,4) ]
gap> PositionCanonical(rt,(1,2,3));
1
gap> rt[1];
()
```

39.9 Double Cosets

39.9.1 DoubleCoset

▷ DoubleCoset(*U, g, V*) (operation)

The groups U and V must be subgroups of a common supergroup G of which g is an element. This command constructs the double coset $U\ g\ V$ which is the set of all elements of the form ugv for any $u \in U$, $v \in V$. For element operations such as in, a double coset behaves like a set of group elements. The double coset stores U in the attribute LeftActingGroup, g as Representative (30.4.7), and V as RightActingGroup.

39.9.2 RepresentativesContainedRightCosets

▷ RepresentativesContainedRightCosets(*D*) (attribute)

A double coset $D = UgV$ can be considered as a union of right cosets Uh_i. (It is the union of the orbit of Ug under right multiplication by V.) For a double coset *D* this function returns a set of representatives h_i such that $D = \bigcup_{h_i} Uh_i$. The representatives returned are canonical for U (see CanonicalRightCosetElement (39.7.3)) and form a set.

Example

```
gap> u:=Subgroup(g,[(1,2,3),(1,2)]);;v:=Subgroup(g,[(3,4)]);;
gap> c:=DoubleCoset(u,(2,4),v);
DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(2,4),Group( [ (3,4) ] ))
gap> (1,2,3) in c;
false
gap> (2,3,4) in c;
true
gap> LeftActingGroup(c);
Group([ (1,2,3), (1,2) ])
gap> RightActingGroup(c);
Group([ (3,4) ])
gap> RepresentativesContainedRightCosets(c);
[ (2,3,4) ]
```

39.9.3 DoubleCosets

▷ DoubleCosets(*G, U, V*) (operation)
▷ DoubleCosetsNC(*G, U, V*) (operation)

computes a duplicate free list of all double cosets *U g V* for $g \in G$. The groups *U* and *V* must be subgroups of the group *G*. The NC version does not check whether *U* and *V* are subgroups of *G*.

Example

```
gap> dc:=DoubleCosets(g,u,v);
[ DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(),Group( [ (3,4) ] )),
  DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(1,3)(2,4),Group(
    [ (3,4) ] )), DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(1,4)
    (2,3),Group( [ (3,4) ] )) ]
gap> List(dc,Representative);
[ (), (1,3)(2,4), (1,4)(2,3) ]
```

39.9.4 IsDoubleCoset (operation)

▷ IsDoubleCoset(*obj*) (Category)

The category of double cosets.

39.9.5 DoubleCosetRepsAndSizes

▷ DoubleCosetRepsAndSizes(*G, U, V*) (operation)

returns a list of double coset representatives and their sizes, the entries are lists of the form $[r,n]$ where r and n are an element of the double coset and the size of the coset, respectively. This operation is faster than DoubleCosetsNC (39.9.3) because no double coset objects have to be created.

Example

```
gap> dc:=DoubleCosetRepsAndSizes(g,u,v);
[ [ (), 12 ], [ (1,3)(2,4), 6 ], [ (1,4)(2,3), 6 ] ]
```

39.9.6 InfoCoset

▷ InfoCoset (info class)

The information function for coset and double coset operations is InfoCoset.

39.10 Conjugacy Classes

39.10.1 ConjugacyClass

▷ ConjugacyClass(*G*, *g*) (operation)

creates the conjugacy class in *G* with representative *g*. This class is an external set, so functions such as Representative (30.4.7) (which returns *g*), ActingDomain (41.12.3) (which returns *G*), StabilizerOfExternalSet (41.12.10) (which returns the centralizer of *g*) and AsList (30.3.8) work for it.

A conjugacy class is an external orbit (see ExternalOrbit (41.12.9)) of group elements with the group acting by conjugation on it. Thus element tests or operation representatives can be computed. The attribute Centralizer (35.4.4) gives the centralizer of the representative (which is the same result as StabilizerOfExternalSet (41.12.10)). (This is a slight abuse of notation: This is *not* the centralizer of the class as a *set* which would be the standard behaviour of Centralizer (35.4.4).)

39.10.2 ConjugacyClasses (attribute)

▷ ConjugacyClasses(*G*) (attribute)

returns the conjugacy classes of elements of *G* as a list of class objects of *G* (see ConjugacyClass (39.10.1) for details). It is guaranteed that the class of the identity is in the first position, the further arrangement depends on the method chosen (and might be different for equal but not identical groups).

For very small groups (of size up to 500) the classes will be computed by the conjugation action of *G* on itself (see ConjugacyClassesByOrbits (39.10.4)). This can be deliberately switched off using the "noaction" option shown below.

For solvable groups, the default method to compute the classes is by homomorphic lift (see section 45.17).

For other groups the method of [Hul00] is employed.

ConjugacyClasses supports the following options that can be used to modify this strategy:

random

The classes are computed by random search. See ConjugacyClassesByRandomSearch (39.10.3) below.

`action`

The classes are computed by action of `G` on itself. See `ConjugacyClassesByOrbits` (39.10.4) below.

`noaction`

Even for small groups `ConjugacyClassesByOrbits` (39.10.4) is not used as a default. This can be useful if the elements of the group use a lot of memory.

Example

```
gap> g:=SymmetricGroup(4);;
gap> cl:=ConjugacyClasses(g);
[ ()^G, (1,2)^G, (1,2)(3,4)^G, (1,2,3)^G, (1,2,3,4)^G ]
gap> Representative(cl[3]);Centralizer(cl[3]);
(1,2)(3,4)
Group([ (1,2), (1,3)(2,4), (3,4) ])
gap> Size(Centralizer(cl[5]));
4
gap> Size(cl[2]);
6
```

In general, you will not need to have to influence the method, but simply call `ConjugacyClasses` –GAP will try to select a suitable method on its own. The method specifications are provided here mainly for expert use.

39.10.3 ConjugacyClassesByRandomSearch

▷ `ConjugacyClassesByRandomSearch(G)` (function)

computes the classes of the group `G` by random search. This works very efficiently for almost simple groups.

This function is also accessible via the option `random` to the function `ConjugacyClass` (39.10.1).

39.10.4 ConjugacyClassesByOrbits

▷ `ConjugacyClassesByOrbits(G)` (function)

computes the classes of the group `G` as orbits of `G` on its elements. This can be quick but unsurprisingly may also take a lot of memory if `G` becomes larger. All the classes will store their element list and thus a membership test will be quick as well.

This function is also accessible via the option `action` to the function `ConjugacyClass` (39.10.1).

Typically, for small groups (roughly of order up to 10^3) the computation of classes as orbits under the action is fastest; memory restrictions (and the increasing cost of eliminating duplicates) make this less efficient for larger groups.

Calculation by random search has the smallest memory requirement, but in generally performs worse, the more classes are there.

The following example shows the effect of this for a small group with many classes:

Example

```
gap> h:=Group((4,5)(6,7,8),(1,2,3)(5,6,9));;ConjugacyClasses(h:noaction);;time;
110
gap> h:=Group((4,5)(6,7,8),(1,2,3)(5,6,9));;ConjugacyClasses(h:random);;time;
```

```
300
gap> h:=Group((4,5)(6,7,8),(1,2,3)(5,6,9));;ConjugacyClasses(h:action);;time;
30
```

39.10.5 NrConjugacyClasses

▷ `NrConjugacyClasses(G)` (attribute)

returns the number of conjugacy classes of `G`.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> NrConjugacyClasses(g);
5
```

39.10.6 RationalClass

▷ `RationalClass(G, g)` (operation)

creates the rational class in `G` with representative `g`. A rational class consists of all elements that are conjugate to `g` or to an i-th power of `g` where i is coprime to the order of `g`. Thus a rational class can be interpreted as a conjugacy class of cyclic subgroups. A rational class is an external set (`IsExternalSet` (41.12.1)) of group elements with the group acting by conjugation on it, but not an external orbit.

39.10.7 RationalClasses

▷ `RationalClasses(G)` (attribute)

returns a list of the rational classes of the group `G`. (See `RationalClass` (39.10.6).)

Example

```
gap> RationalClasses(DerivedSubgroup(g));
[ RationalClass( AlternatingGroup( [ 1 .. 4 ] ), () ),
  RationalClass( AlternatingGroup( [ 1 .. 4 ] ), (1,2)(3,4) ),
  RationalClass( AlternatingGroup( [ 1 .. 4 ] ), (1,2,3) ) ]
```

39.10.8 GaloisGroup (of rational class of a group)

▷ `GaloisGroup(ratcl)` (attribute)

Suppose that `ratcl` is a rational class of a group G with representative g. The exponents i for which g^i lies already in the ordinary conjugacy class of g, form a subgroup of the *prime residue class group* P_n (see `PrimitiveRootMod` (15.3.3)), the so-called *Galois group* of the rational class. The prime residue class group P_n is obtained in GAP as `Units( Integers mod n )`, the unit group of a residue class ring. The Galois group of a rational class `ratcl` is stored in the attribute `GaloisGroup` as a subgroup of this group.

39.10.9 IsConjugate

▷ IsConjugate(*G*, *x*, *y*) (operation)
▷ IsConjugate(*G*, *U*, *V*) (operation)

tests whether the elements *x* and *y* or the subgroups *U* and *V* are conjugate under the action of *G*. (They do not need to be *contained in* *G*.) This command is only a shortcut to RepresentativeAction (41.6.1).

Example

```
gap> IsConjugate(g,Group((1,2,3,4),(1,3)),Group((1,3,2,4),(1,2)));
true
```

RepresentativeAction (41.6.1) can be used to obtain conjugating elements.

Example

```
gap> RepresentativeAction(g,(1,2),(3,4));
(1,3)(2,4)
```

39.10.10 NthRootsInGroup

▷ NthRootsInGroup(*G*, *e*, *n*) (function)

Let *e* be an element in the group *G*. This function returns a list of all those elements in *G* whose *n*-th power is *e*.

Example

```
gap> NthRootsInGroup(g,(1,2)(3,4),2);
[ (1,3,2,4), (1,4,2,3) ]
```

39.11 Normal Structure

For the operations Centralizer (35.4.4) and Centre (35.4.5), see Chapter 35.

39.11.1 Normalizer

▷ Normalizer(*G*, *U*) (operation)
▷ Normalizer(*G*, *g*) (operation)

For two groups *G*, *U*, Normalizer computes the normalizer $N_G(U)$, that is, the stabilizer of *U* under the conjugation action of *G*.

For a group *G* and a group element *g*, Normalizer computes $N_G(\langle g \rangle)$.

Example

```
gap> Normalizer(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3), (2,3) ])
```

39.11.2 Core

▷ `Core(S, U)` (operation)

If `S` and `U` are groups of elements in the same family, this operation returns the core of `U` in `S`, that is the intersection of all `S`-conjugates of `U`.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> Core(g,Subgroup(g,[(1,2,3,4)]));
Group(())
```

39.11.3 PCore

▷ `PCore(G, p)` (function)

The *p-core* of `G` is the largest normal `p`-subgroup of `G`. It is the core of a Sylow `p` subgroup of `G`, see `Core` (39.11.2).

Example

```
gap> PCore(g,2);
Group([ (1,4)(2,3), (1,2)(3,4) ])
```

39.11.4 NormalClosure

▷ `NormalClosure(G, U)` (operation)

The normal closure of `U` in `G` is the smallest normal subgroup of the closure of `G` and `U` which contains `U`.

Example

```
gap> NormalClosure(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3), (1,3,4) ])
gap> NormalClosure(g,Group((3,4,5)));
Group([ (3,4,5), (1,5,4), (1,2,5) ])
```

39.11.5 NormalIntersection

▷ `NormalIntersection(G, U)` (operation)

computes the intersection of `G` and `U`, assuming that `G` is normalized by `U`. This works faster than `Intersection`, but will not produce the intersection if `G` is not normalized by `U`.

Example

```
gap> NormalIntersection(Group((1,2)(3,4),(1,3)(2,4)),Group((1,2,3,4)));
Group([ (1,3)(2,4) ])
```

39.11.6 ComplementClassesRepresentatives

▷ `ComplementClassesRepresentatives(G, N)` (operation)

Let `N` be a normal subgroup of `G`. This command returns a set of representatives for the conjugacy classes of complements of `N` in `G`. Complements are subgroups of `G` which intersect trivially with `N` and together with `N` generate `G`.

At the moment only methods for a solvable `N` are available.

Example

```
gap> ComplementClassesRepresentatives(g,Group((1,2)(3,4),(1,3)(2,4)));
[ Group([ (3,4), (2,4,3) ]) ]
```

39.11.7 InfoComplement

▷ `InfoComplement` (info class)

Info class for the complement routines.

39.12 Specific and Parametrized Subgroups

The centre of a group (the subgroup of those elements that commute with all other elements of the group) can be computed by the operation `Centre` (35.4.5).

39.12.1 TrivialSubgroup

▷ `TrivialSubgroup(G)` (attribute)

Example

```
gap> TrivialSubgroup(g);
Group(())
```

39.12.2 CommutatorSubgroup

▷ `CommutatorSubgroup(G, H)` (operation)

If `G` and `H` are two groups of elements in the same family, this operation returns the group generated by all commutators $[g,h] = g^{-1}h^{-1}gh$ (see `Comm` (31.12.3)) of elements $g \in G$ and $h \in H$, that is the group $\langle [g,h] \mid g \in G, h \in H \rangle$.

Example

```
gap> CommutatorSubgroup(Group((1,2,3),(1,2)),Group((2,3,4),(3,4)));
Group([ (1,4)(2,3), (1,3,4) ])
gap> Size(last);
12
```

39.12.3 DerivedSubgroup

▷ `DerivedSubgroup(G)` (attribute)

The derived subgroup G' of `G` is the subgroup generated by all commutators of pairs of elements of `G`. It is normal in `G` and the factor group G/G' is the largest abelian factor group of `G`.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> DerivedSubgroup(g);
Group([ (1,3,2), (1,4,3) ])
```

39.12.4 CommutatorLength

▷ `CommutatorLength(G)` (attribute)

returns the minimal number n such that each element in the derived subgroup (see `DerivedSubgroup` (39.12.3)) of the group `G` can be written as a product of (at most) n commutators of elements in `G`.

Example

```
gap> CommutatorLength( g );
1
```

39.12.5 FittingSubgroup

▷ `FittingSubgroup(G)` (attribute)

The Fitting subgroup of a group `G` is its largest nilpotent normal subgroup.

Example

```
gap> FittingSubgroup(g);
Group([ (1,2)(3,4), (1,4)(2,3) ])
```

39.12.6 FrattiniSubgroup

▷ `FrattiniSubgroup(G)` (attribute)

The Frattini subgroup of a group `G` is the intersection of all maximal subgroups of `G`.

Example

```
gap> FrattiniSubgroup(g);
Group(())
```

39.12.7 PrefrattiniSubgroup

▷ `PrefrattiniSubgroup(G)` (attribute)

returns a Prefrattini subgroup of the finite solvable group `G`.

A factor M/N of `G` is called a Frattini factor if M/N is contained in the Frattini subgroup of G/N. A subgroup P is a Prefrattini subgroup of `G` if P covers each Frattini chief factor of `G`, and if for each maximal subgroup of `G` there exists a conjugate maximal subgroup, which contains P. In a finite solvable group `G` the Prefrattini subgroups form a characteristic conjugacy class of subgroups and the intersection of all these subgroups is the Frattini subgroup of `G`.

Example

```
gap> G := SmallGroup( 60, 7 );
<pc group of size 60 with 4 generators>
gap> P := PrefrattiniSubgroup(G);
```

```
Group([ f2 ])
gap> Size(P);
2
gap> IsNilpotent(P);
true
gap> Core(G,P);
Group([  ])
gap> FrattiniSubgroup(G);
Group([  ])
```

39.12.8 PerfectResiduum

▷ `PerfectResiduum(G)` (attribute)

is the smallest normal subgroup of `G` that has a solvable factor group.

Example

```
gap> PerfectResiduum(Group((1,2,3,4,5),(1,2)));
Group([ (1,3,2), (2,4,3), (1,3)(4,5) ])
```

39.12.9 RadicalGroup

▷ `RadicalGroup(G)` (attribute)

is the radical of `G`, i.e., the largest solvable normal subgroup of `G`.

Example

```
gap> RadicalGroup(SL(2,5));
<group of 2x2 matrices of size 2 over GF(5)>
gap> Size(last);
2
```

39.12.10 Socle

▷ `Socle(G)` (attribute)

The socle of the group `G` is the subgroup generated by all minimal normal subgroups.

Example

```
gap> Socle(g);
Group([ (1,4)(2,3), (1,2)(3,4) ])
```

39.12.11 SupersolvableResiduum

▷ `SupersolvableResiduum(G)` (attribute)

is the supersolvable residuum of the group `G`, that is, its smallest normal subgroup N such that the factor group G/N is supersolvable.

Example

```
gap> SupersolvableResiduum(g);
Group([ (1,3)(2,4), (1,4)(2,3) ])
```

39.12.12 PRump

▷ `PRump(G, p)` (function)

For a prime p, the *p-rump* of a group G is the subgroup $G'G^p$.
@example missing!@

39.13 Sylow Subgroups and Hall Subgroups

With respect to the following GAP functions, please note that by theorems of P. Hall, a group G is solvable if and only if one of the following conditions holds.

1. For each prime p dividing the order of G, there exists a p-complement (see `SylowComplement` (39.13.2)).
2. For each set P of primes dividing the order of G, there exists a P-Hall subgroup (see `HallSubgroup` (39.13.3)).
3. G has a Sylow system (see `SylowSystem` (39.13.4)).
4. G has a complement system (see `ComplementSystem` (39.13.5)).

39.13.1 SylowSubgroup

▷ `SylowSubgroup(G, p)` (function)

returns a Sylow p subgroup of the finite group G. This is a p-subgroup of G whose index in G is coprime to p. `SylowSubgroup` computes Sylow subgroups via the operation `SylowSubgroupOp`.

Example

```
gap> g:=SymmetricGroup(4);;
gap> SylowSubgroup(g,2);
Group([ (1,2), (3,4), (1,3)(2,4) ])
```

39.13.2 SylowComplement

▷ `SylowComplement(G, p)` (function)

returns a Sylow p-complement of the finite group G. This is a subgroup U of order coprime to p such that the index $[G : U]$ is a p-power.

At the moment methods exist only if G is solvable and GAP will issue an error if G is not solvable.

Example

```
gap> SylowComplement(g,3);
Group([ (1,2), (3,4), (1,3)(2,4) ])
```

39.13.3 HallSubgroup

▷ HallSubgroup(*G*, *P*) (function)

computes a *P*-Hall subgroup for a set *P* of primes. This is a subgroup the order of which is only divisible by primes in *P* and whose index is coprime to all primes in *P*. Such a subgroup is unique up to conjugacy if *G* is solvable. The function computes Hall subgroups via the operation HallSubgroupOp.

If *G* is solvable this function always returns a subgroup. If *G* is not solvable this function might return a subgroup (if it is unique up to conjugacy), a list of subgroups (which are representatives of the conjugacy classes in case there are several such classes) or fail if no such subgroup exists.

Example

```
gap> h:=SmallGroup(60,10);;
gap> u:=HallSubgroup(h,[2,3]);
Group([ f1, f2, f3 ])
gap> Size(u);
12
gap> h:=PSL(3,5);;
gap> HallSubgroup(h,[2,3]);
[ <permutation group of size 96 with 6 generators>,
  <permutation group of size 96 with 6 generators> ]
gap> HallSubgroup(h,[3,31]);
Group([ (2,29,7)(3,30,11)(4,28,8)(5,27,9)(6,31,10)(12,24,21)(13,25,20)
(14,23,17)(15,22,19)(16,26,18), (1,20,31,23,11,25,29,3,21,24,6,19,5,
17,12,14,4,18,8,10,30,27,16,26,22,13,7,15,28,9,2) ])
gap> HallSubgroup(h,[5,31]);
fail
```

39.13.4 SylowSystem

▷ SylowSystem(*G*) (attribute)

A Sylow system of a group *G* is a set of Sylow subgroups of *G* such that every pair of subgroups from this set commutes as subgroups. Sylow systems exist only for solvable groups. The operation returns fail if the group *G* is not solvable.

Example

```
gap> h:=SmallGroup(60,10);;
gap> SylowSystem(h);
[ Group([ f1, f2 ]), Group([ f3 ]), Group([ f4 ]) ]
gap> List(last,Size);
[ 4, 3, 5 ]
```

39.13.5 ComplementSystem

▷ ComplementSystem(*G*) (attribute)

A complement system of a group *G* is a set of Hall p'-subgroups of *G*, where p' runs through the subsets of prime factors of $|G|$ that omit exactly one prime. Every pair of subgroups from this set commutes as subgroups. Complement systems exist only for solvable groups, therefore ComplementSystem returns fail if the group *G* is not solvable.

Example

```
gap> ComplementSystem(h);
[ Group([ f3, f4 ]), Group([ f1, f2, f4 ]), Group([ f1, f2, f3 ]) ]
gap> List(last,Size);
[ 15, 20, 12 ]
```

39.13.6 HallSystem

▷ `HallSystem(G)` (attribute)

returns a list containing one Hall P-subgroup for each set P of prime divisors of the order of `G`. Hall systems exist only for solvable groups. The operation returns `fail` if the group `G` is not solvable.

Example

```
gap> HallSystem(h);
[ Group([  ]), Group([ f1, f2 ]), Group([ f1, f2, f3 ]),
  Group([ f1, f2, f3, f4 ]), Group([ f1, f2, f4 ]), Group([ f3 ]),
  Group([ f3, f4 ]), Group([ f4 ]) ]
gap> List(last,Size);
[ 1, 4, 12, 60, 20, 3, 15, 5 ]
```

39.14 Subgroups characterized by prime powers

39.14.1 Omega

▷ `Omega(G, p[, n])` (operation)

For a p-group `G`, one defines $\Omega_n(G) = \{g \in G \mid g^{p^n} = 1\}$. The default value for `n` is `1`.
@At the moment methods exist only for abelian `G` and `n=1`.@

Example

```
gap> h:=SmallGroup(16,10);
<pc group of size 16 with 4 generators>
gap> Omega(h,2);
Group([ f2, f3, f4 ])
```

39.14.2 Agemo

▷ `Agemo(G, p[, n])` (function)

For a p-group `G`, one defines $\mho_n(G) = \langle g^{p^n} \mid g \in G \rangle$. The default value for `n` is `1`.

Example

```
gap> Agemo(h,2);Agemo(h,2,2);
Group([ f4 ])
Group([  ])
```

39.15 Group Properties

Some properties of groups can be defined not only for groups but also for other structures. For example, nilpotency and solvability make sense also for algebras. Note that these names refer to different

definitions for groups and algebras, contrary to the situation with finiteness or commutativity. In such cases, the name of the function for groups got a suffix `Group` to distinguish different meanings for different structures.

Some functions, such as `IsPSolvable` (39.15.23) and `IsPNilpotent` (39.15.24), although they are mathematical properties, are not properties in the sense of GAP (see 13.5 and 13.7), as they depend on a parameter.

39.15.1 IsCyclic

▷ `IsCyclic(G)` (property)

A group is *cyclic* if it can be generated by one element. For a cyclic group, one can compute a generating set consisting of only one element using `MinimalGeneratingSet` (39.22.3).

39.15.2 IsElementaryAbelian

▷ `IsElementaryAbelian(G)` (property)

A group `G` is elementary abelian if it is commutative and if there is a prime p such that the order of each element in `G` divides p.

39.15.3 IsNilpotentGroup

▷ `IsNilpotentGroup(G)` (property)

A group is *nilpotent* if the lower central series (see `LowerCentralSeriesOfGroup` (39.17.11) for a definition) reaches the trivial subgroup in a finite number of steps.

39.15.4 NilpotencyClassOfGroup

▷ `NilpotencyClassOfGroup(G)` (attribute)

The nilpotency class of a nilpotent group `G` is the number of steps in the lower central series of `G` (see `LowerCentralSeriesOfGroup` (39.17.11));

If `G` is not nilpotent an error is issued.

39.15.5 IsPerfectGroup

▷ `IsPerfectGroup(G)` (property)

A group is *perfect* if it equals its derived subgroup (see `DerivedSubgroup` (39.12.3)).

39.15.6 IsSolvableGroup

▷ `IsSolvableGroup(G)` (property)

A group is *solvable* if the derived series (see `DerivedSeriesOfGroup` (39.17.7) for a definition) reaches the trivial subgroup in a finite number of steps.

For finite groups this is the same as being polycyclic (see IsPolycyclicGroup (39.15.7)), and each polycyclic group is solvable, but there are infinite solvable groups that are not polycyclic.

39.15.7 IsPolycyclicGroup

▷ IsPolycyclicGroup(*G*) (property)

A group is polycyclic if it has a subnormal series with cyclic factors. For finite groups this is the same as if the group is solvable (see IsSolvableGroup (39.15.6)).

39.15.8 IsSupersolvableGroup

▷ IsSupersolvableGroup(*G*) (property)

A finite group is *supersolvable* if it has a normal series with cyclic factors.

39.15.9 IsMonomialGroup

▷ IsMonomialGroup(*G*) (property)

A finite group is *monomial* if every irreducible complex character is induced from a linear character of a subgroup.

39.15.10 IsSimpleGroup

▷ IsSimpleGroup(*G*) (property)

A group is *simple* if it is nontrivial and has no nontrivial normal subgroups.

39.15.11 IsAlmostSimpleGroup

▷ IsAlmostSimpleGroup(*G*) (property)

A group `G` is *almost simple* if a nonabelian simple group S exists such that `G` is isomorphic to a subgroup of the automorphism group of S that contains all inner automorphisms of S.

Equivalently, `G` is almost simple if and only if it has a unique minimal normal subgroup N and if N is a nonabelian simple group.

Note that an almost simple group is *not* defined as an extension of a simple group by outer automorphisms, since we want to exclude extensions of groups of prime order. In particular, a *simple* group is *almost simple* if and only if it is nonabelian.

Example

```
gap> IsAlmostSimpleGroup( AlternatingGroup( 5 ) );
true
gap> IsAlmostSimpleGroup( SymmetricGroup( 5 ) );
true
gap> IsAlmostSimpleGroup( SymmetricGroup( 3 ) );
false
gap> IsAlmostSimpleGroup( SL( 2, 5 ) );
false
```

39.15.12 IsomorphismTypeInfoFiniteSimpleGroup

▷ `IsomorphismTypeInfoFiniteSimpleGroup(G)` (attribute)
▷ `IsomorphismTypeInfoFiniteSimpleGroup(n)` (attribute)

For a finite simple group `G`, `IsomorphismTypeInfoFiniteSimpleGroup` returns a record with the components `series`, `name` and possibly `parameter`, describing the isomorphism type of `G`. The component `name` is a string that gives name(s) for `G`, and `series` is a string that describes the following series.

(If different characterizations of `G` are possible only one is given by `series` and `parameter`, while `name` may give several names.)

`"A"` Alternating groups, `parameter` gives the natural degree.

`"L"` Linear groups (Chevalley type A), `parameter` is a list $[n,q]$ that indicates $L(n,q)$.

`"2A"`
Twisted Chevalley type ${}^2A$, `parameter` is a list $[n,q]$ that indicates ${}^2A(n,q)$.

`"B"` Chevalley type B, `parameter` is a list $[n,q]$ that indicates $B(n,q)$.

`"2B"`
Twisted Chevalley type ${}^2B$, `parameter` is a value q that indicates ${}^2B(2,q)$.

`"C"` Chevalley type C, `parameter` is a list $[n,q]$ that indicates $C(n,q)$.

`"D"` Chevalley type D, `parameter` is a list $[n,q]$ that indicates $D(n,q)$.

`"2D"`
Twisted Chevalley type ${}^2D$, `parameter` is a list $[n,q]$ that indicates ${}^2D(n,q)$.

`"3D"`
Twisted Chevalley type ${}^3D$, `parameter` is a value q that indicates ${}^3D(4,q)$.

`"E"` Exceptional Chevalley type E, `parameter` is a list $[n,q]$ that indicates $E_n(q)$. The value of `n` is 6, 7, or 8.

`"2E"`
Twisted exceptional Chevalley type E_6, `parameter` is a value q that indicates ${}^2E_6(q)$.

`"F"` Exceptional Chevalley type F, `parameter` is a value q that indicates $F(4,q)$.

`"2F"`
Twisted exceptional Chevalley type ${}^2F$ (Ree groups), `parameter` is a value q that indicates ${}^2F(4,q)$.

`"G"` Exceptional Chevalley type G, `parameter` is a value q that indicates $G(2,q)$.

`"2G"`
Twisted exceptional Chevalley type ${}^2G$ (Ree groups), `parameter` is a value q that indicates ${}^2G(2,q)$.

"Spor"
Sporadic simple groups, name gives the name.

"Z" Cyclic groups of prime size, parameter gives the size.

An equal sign in the name denotes different naming schemes for the same group, a tilde sign abstract isomorphisms between groups constructed in a different way.

Example

```
gap> IsomorphismTypeInfoFiniteSimpleGroup(
>                               Group((4,5)(6,7),(1,2,4)(3,5,6)));
rec(
  name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~ 2A(1,\
7) = U(2,7) ~ A(2,2) = L(3,2)", parameter := [ 2, 7 ], series := "L" )
```

For a positive integer *n*, IsomorphismTypeInfoFiniteSimpleGroup returns fail if *n* is not the order of a finite simple group, and a record as described for the case of a group *G* otherwise. If more than one simple group of order *n* exists then the result record contains only the name component, a string that lists the two possible isomorphism types of simple groups of this order.

Example

```
gap> IsomorphismTypeInfoFiniteSimpleGroup( 5 );
rec( name := "Z(5)", parameter := 5, series := "Z" )
gap> IsomorphismTypeInfoFiniteSimpleGroup( 6 );
fail
gap> IsomorphismTypeInfoFiniteSimpleGroup(Size(SymplecticGroup(6,3))/2);
rec(
  name := "cannot decide from size alone between B(3,3) = O(7,3) and C\
(3,3) = S(6,3)", parameter := [ 3, 3 ] )
```

39.15.13 SimpleGroup

▷ SimpleGroup(*id[, param]*) (function)

This function will construct AN instance of the specified simple group. Groups are specified via their name in ATLAS style notation, with parameters added if necessary. The intelligence applied to parsing the name is limited, and at the moment no proper extensions can be constructed. For groups who do not have a permutation representation of small degree the ATLASREP package might need to be installed to construct theses groups.

Example

```
gap> g:=SimpleGroup("M(23)");
M23
gap> Size(g);
10200960
gap> g:=SimpleGroup("PSL",3,5);
PSL(3,5)
gap> Size(g);
372000
gap> g:=SimpleGroup("PSp6",2);
PSp(6,2)
```

39.15.14 SimpleGroupsIterator

▷ SimpleGroupsIterator(*[start[, end]]*) (function)

This function returns an iterator that will run over all simple groups, starting at order *start* if specified, up to order 10^{18} (or – if specified – order *end*). If the option *NOPSL2* is given, groups of type $PSL_2(q)$ are omitted.

Example

```
gap> it:=SimpleGroupsIterator(20000);
<iterator>
gap> List([1..8],x->NextIterator(it));
[ A8, PSL(3,4), PSL(2,37), PSp(4,3), Sz(8), PSL(2,32), PSL(2,41),
  PSL(2,43) ]
gap> it:=SimpleGroupsIterator(1,2000);;
gap> l:=[];;for i in it do Add(l,i);od;l;
[ A5, PSL(2,7), A6, PSL(2,8), PSL(2,11), PSL(2,13) ]
gap> it:=SimpleGroupsIterator(20000,100000:NOPSL2);;
gap> l:=[];;for i in it do Add(l,i);od;l;
[ A8, PSL(3,4), PSp(4,3), Sz(8), PSU(3,4), M12 ]
```

39.15.15 SmallSimpleGroup

▷ SmallSimpleGroup(*order[, i]*) (function)

Returns: The *i*th simple group of order *order* in the stored list, given in a small-degree permutation representation, or fail (20.2.1) if no such simple group exists.

If *i* is not given, it defaults to 1. Currently, all simple groups of order less than 10^6 are available via this function.

Example

```
gap> SmallSimpleGroup(60);
A5
gap> SmallSimpleGroup(20160,1);
A8
gap> SmallSimpleGroup(20160,2);
PSL(3,4)
```

39.15.16 AllSmallNonabelianSimpleGroups

▷ AllSmallNonabelianSimpleGroups(*orders*) (function)

Returns: A list of all nonabelian simple groups whose order lies in the range *orders*.

The groups are given in small-degree permutation representations. The returned list is sorted by ascending group order. Currently, all simple groups of order less than 10^6 are available via this function.

Example

```
gap> List(AllSmallNonabelianSimpleGroups([1..1000000]),
>         StructureDescription);
[ "A5", "PSL(3,2)", "A6", "PSL(2,8)", "PSL(2,11)", "PSL(2,13)",
  "PSL(2,17)", "A7", "PSL(2,19)", "PSL(2,16)", "PSL(3,3)",
  "PSU(3,3)", "PSL(2,23)", "PSL(2,25)", "M11", "PSL(2,27)",
  "PSL(2,29)", "PSL(2,31)", "A8", "PSL(3,4)", "PSL(2,37)", "O(5,3)",
  "Sz(8)", "PSL(2,32)", "PSL(2,41)", "PSL(2,43)", "PSL(2,47)",
  "PSL(2,49)", "PSU(3,4)", "PSL(2,53)", "M12", "PSL(2,59)",
```

```
    "PSL(2,61)", "PSU(3,5)", "PSL(2,67)", "J1", "PSL(2,71)", "A9",
    "PSL(2,73)", "PSL(2,79)", "PSL(2,64)", "PSL(2,81)", "PSL(2,83)",
    "PSL(2,89)", "PSL(3,5)", "M22", "PSL(2,97)", "PSL(2,101)",
    "PSL(2,103)", "HJ", "PSL(2,107)", "PSL(2,109)", "PSL(2,113)",
    "PSL(2,121)", "PSL(2,125)", "O(5,4)" ]
```

39.15.17 IsFinitelyGeneratedGroup

▷ `IsFinitelyGeneratedGroup(`*G*`)` (property)

tests whether the group *G* can be generated by a finite number of generators. (This property is mainly used to obtain finiteness conditions.)

Note that this is a pure existence statement. Even if a group is known to be generated by a finite number of elements, it can be very hard or even impossible to obtain such a generating set if it is not known.

39.15.18 IsSubsetLocallyFiniteGroup

▷ `IsSubsetLocallyFiniteGroup(`*U*`)` (property)

A group is called locally finite if every finitely generated subgroup is finite. This property checks whether the group *U* is a subset of a locally finite group. This is used to check whether finite generation will imply finiteness, as it does for example for permutation groups.

39.15.19 IsPGroup

▷ `IsPGroup(`*G*`)` (property)

A *p-group* is a finite group whose order (see `Size` (30.4.6)) is of the form p^n for a prime integer p and a nonnegative integer n. `IsPGroup` returns `true` if *G* is a p-group, and `false` otherwise.

39.15.20 PrimePGroup

▷ `PrimePGroup(`*G*`)` (attribute)

If *G* is a nontrivial p-group (see `IsPGroup` (39.15.19)), `PrimePGroup` returns the prime integer p; if *G* is trivial then `PrimePGroup` returns `fail`. Otherwise an error is issued.

(One should avoid a common error of writing `if IsPGroup(g) then ... PrimePGroup(g) ...` where the code represented by dots assumes that `PrimePGroup(g)` is an integer.)

39.15.21 PClassPGroup

▷ `PClassPGroup(`*G*`)` (attribute)

The p-class of a p-group *G* (see `IsPGroup` (39.15.19)) is the length of the lower p-central series (see `PCentralSeries` (39.17.13)) of *G*. If *G* is not a p-group then an error is issued.

39.15.22 RankPGroup

▷ RankPGroup(*G*) (attribute)

For a *p*-group *G* (see IsPGroup (39.15.19)), RankPGroup returns the *rank* of *G*, which is defined as the minimal size of a generating system of *G*. If *G* is not a *p*-group then an error is issued.

Example

```
gap> h:=Group((1,2,3,4),(1,3));;
gap> PClassPGroup(h);
2
gap> RankPGroup(h);
2
```

39.15.23 IsPSolvable

▷ IsPSolvable(*G*, *p*) (function)

A finite group is *p*-solvable if every chief factor either has order not divisible by *p*, or is solvable.

39.15.24 IsPNilpotent

▷ IsPNilpotent(*G*, *p*) (function)

A group is *p*-nilpotent if it possesses a normal *p*-complement.

39.16 Numerical Group Attributes

This section gives only some examples of numerical group attributes, so it should not serve as a collection of all numerical group attributes. The manual contains more such attributes documented in this manual, for example, NrConjugacyClasses (39.10.5), NilpotencyClassOfGroup (39.15.4) and others.

Note also that some functions, such as EulerianFunction (39.16.3), are mathematical attributes, but not GAP attributes (see 13.5) as they are depending on a parameter.

39.16.1 AbelianInvariants

▷ AbelianInvariants(*G*) (attribute)

returns the abelian invariants (also sometimes called primary decomposition) of the commutator factor group of the group *G*. These are given as a list of prime-powers or zeroes and describe the structure of G/G' as a direct product of cyclic groups of prime power (or infinite) order.

(See IndependentGeneratorsOfAbelianGroup (39.22.5) to obtain actual generators).

Example

```
gap> g:=Group((1,2,3,4),(1,2),(5,6));;
gap> AbelianInvariants(g);
[ 2, 2 ]
gap> h:=FreeGroup(2);;h:=h/[h.1^3];;
gap> AbelianInvariants(h);
[ 0, 3 ]
```

39.16.2 Exponent

▷ `Exponent(G)` (attribute)

The exponent e of a group `G` is the lcm of the orders of its elements, that is, e is the smallest integer such that $g^e = 1$ for all $g \in$ `G`.

Example

```
gap> Exponent(g);
12
```

39.16.3 EulerianFunction

▷ `EulerianFunction(G, n)` (operation)

returns the number of `n`-tuples $(g_1, g_2, \ldots, g_n)$ of elements of the group `G` that generate the whole group `G`. The elements of such an `n`-tuple need not be different.

In [Hal36], the notation $\phi_n(G)$ is used for the value returned by `EulerianFunction`, and the quotient of $\phi_n(G)$ by the order of the automorphism group of `G` is called $d_n(G)$. If `G` is a nonabelian simple group then $d_n(G)$ is the greatest number d for which the direct product of d groups isomorphic with `G` can be generated by `n` elements.

If the Library of Tables of Marks (see Chapter 70) covers the group `G`, you may also use `EulerianFunctionByTom` (70.9.9).

Example

```
gap> EulerianFunction( g, 2 );
432
```

39.17 Subgroup Series

In group theory many subgroup series are considered, and GAP provides commands to compute them. In the following sections, there is always a series $G = U_1 > U_2 > \cdots > U_m = \langle 1 \rangle$ of subgroups considered. A series also may stop without reaching G or $\langle 1 \rangle$.

A series is called *subnormal* if every U_{i+1} is normal in U_i.

A series is called *normal* if every U_i is normal in G.

A series of normal subgroups is called *central* if U_i/U_{i+1} is central in G/U_{i+1}.

We call a series *refinable* if intermediate subgroups can be added to the series without destroying the properties of the series.

Unless explicitly declared otherwise, all subgroup series are descending. That is they are stored in decreasing order.

39.17.1 ChiefSeries

▷ `ChiefSeries(G)` (attribute)

is a series of normal subgroups of `G` which cannot be refined further. That is there is no normal subgroup N of `G` with $U_i > N > U_{i+1}$. This attribute returns *one* chief series (of potentially many possibilities).

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> ChiefSeries(g);
[ Group([ (1,2,3,4), (1,2) ]),
  Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
```

39.17.2 ChiefSeriesThrough

▷ `ChiefSeriesThrough(G, l)` (operation)

is a chief series of the group `G` going through the normal subgroups in the list `l`, which must be a list of normal subgroups of `G` contained in each other, sorted by descending size. This attribute returns *one* chief series (of potentially many possibilities).

39.17.3 ChiefSeriesUnderAction

▷ `ChiefSeriesUnderAction(H, G)` (operation)

returns a series of normal subgroups of `G` which are invariant under `H` such that the series cannot be refined any further. `G` must be a subgroup of `H`. This attribute returns *one* such series (of potentially many possibilities).

39.17.4 SubnormalSeries

▷ `SubnormalSeries(G, U)` (operation)

If `U` is a subgroup of `G` this operation returns a subnormal series that descends from `G` to a subnormal subgroup $V \geq U$. If `U` is subnormal, $V = U$.

Example

```
gap> s:=SubnormalSeries(g,Group((1,2)(3,4)));
[ Group([ (1,2,3,4), (1,2) ]), Group([ (1,2)(3,4), (1,3)(2,4) ]),
  Group([ (1,2)(3,4) ]) ]
```

39.17.5 CompositionSeries

▷ `CompositionSeries(G)` (attribute)

A composition series is a subnormal series which cannot be refined. This attribute returns *one* composition series (of potentially many possibilities).

39.17.6 DisplayCompositionSeries

▷ `DisplayCompositionSeries(G)` (function)

Displays a composition series of `G` in a nice way, identifying the simple factors.

Example

```
gap> CompositionSeries(g);
[ Group([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group([ (1,3)(2,4) ]), Group(())
 ]
gap> DisplayCompositionSeries(Group((1,2,3,4,5,6,7),(1,2)));
G (2 gens, size 5040)
 | Z(2)
S (5 gens, size 2520)
 | A(7)
1 (0 gens, size 1)
```

39.17.7 DerivedSeriesOfGroup

▷ `DerivedSeriesOfGroup(`*G*`)` (attribute)

The derived series of a group is obtained by $U_{i+1} = U_i'$. It stops if U_i is perfect.

39.17.8 DerivedLength

▷ `DerivedLength(`*G*`)` (attribute)

The derived length of a group is the number of steps in the derived series. (As there is always the group, it is the series length minus 1.)

Example

```
gap> List(DerivedSeriesOfGroup(g),Size);
[ 24, 12, 4, 1 ]
gap> DerivedLength(g);
3
```

39.17.9 ElementaryAbelianSeries

▷ `ElementaryAbelianSeries(`*G*`)` (attribute)
▷ `ElementaryAbelianSeriesLargeSteps(`*G*`)` (attribute)
▷ `ElementaryAbelianSeries(`*list*`)` (attribute)

returns a series of normal subgroups of *G* such that all factors are elementary abelian. If the group is not solvable (and thus no such series exists) it returns `fail`.

The variant `ElementaryAbelianSeriesLargeSteps` tries to make the steps in this series large (by eliminating intermediate subgroups if possible) at a small additional cost.

In the third variant, an elementary abelian series through the given series of normal subgroups in the list *list* is constructed.

Example

```
gap> List(ElementaryAbelianSeries(g),Size);
[ 24, 12, 4, 1 ]
```

39.17.10 InvariantElementaryAbelianSeries

▷ `InvariantElementaryAbelianSeries(G, morph[, N[, fine]])` (operation)

For a (solvable) group `G` and a list of automorphisms `morph` of `G`, this command finds a normal series of `G` with elementary abelian factors such that every group in this series is invariant under every automorphism in `morph`.

If a normal subgroup `N` of `G` which is invariant under `morph` is given, this series is chosen to contain `N`. No tests are performed to check the validity of the arguments.

The series obtained will be constructed to prefer large steps unless `fine` is given as `true`.

Example

```
gap> g:=Group((1,2,3,4),(1,3));
Group([ (1,2,3,4), (1,3) ])
gap> hom:=GroupHomomorphismByImages(g,g,GeneratorsOfGroup(g),
> [(1,4,3,2),(1,4)(2,3)]);
[ (1,2,3,4), (1,3) ] -> [ (1,4,3,2), (1,4)(2,3) ]
gap> InvariantElementaryAbelianSeries(g,[hom]);
[ Group([ (1,2,3,4), (1,3) ]), Group([ (1,3)(2,4) ]), Group(()) ]
```

39.17.11 LowerCentralSeriesOfGroup

▷ `LowerCentralSeriesOfGroup(G)` (attribute)

The lower central series of a group `G` is defined as $U_{i+1} := [G, U_i]$. It is a central series of normal subgroups. The name derives from the fact that U_i is contained in the i-th step subgroup of any central series.

39.17.12 UpperCentralSeriesOfGroup

▷ `UpperCentralSeriesOfGroup(G)` (attribute)

The upper central series of a group `G` is defined as an ending series $U_i/U_{i+1} := Z(G/U_{i+1})$. It is a central series of normal subgroups. The name derives from the fact that U_i contains every i-th step subgroup of a central series.

39.17.13 PCentralSeries

▷ `PCentralSeries(G, p)` (function)

The p-central series of `G` is defined by $U_1 := G$, $U_i := [G, U_{i-1}]U_{i-1}^p$.

39.17.14 JenningsSeries

▷ `JenningsSeries(G)` (attribute)

For a p-group `G`, this function returns its Jennings series. This series is defined by setting $G_1 = G$ and for $i \geq 0$, $G_{i+1} = [G_i, G]G_j^p$, where j is the smallest integer $\geq i/p$.

39.17.15 DimensionsLoewyFactors

▷ DimensionsLoewyFactors(*G*) (attribute)

This operation computes the dimensions of the factors of the Loewy series of *G*. (See [HB82, p. 157] for the slightly complicated definition of the Loewy Series.)

The dimensions are computed via the JenningsSeries (39.17.14) without computing the Loewy series itself.

Example

```
gap> G:= SmallGroup( 3^6, 100 );
<pc group of size 729 with 6 generators>
gap> JenningsSeries( G );
[ <pc group of size 729 with 6 generators>, Group([ f3, f4, f5, f6 ]),
  Group([ f4, f5, f6 ]), Group([ f5, f6 ]), Group([ f5, f6 ]),
  Group([ f5, f6 ]), Group([ f6 ]), Group([ f6 ]), Group([ f6 ]),
  Group([ <identity> of ... ]) ]
gap> DimensionsLoewyFactors(G);
[ 1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26,
  27, 27, 27, 27, 27, 27, 27, 27, 27, 26, 25, 23, 22, 20, 19, 17, 16,
  14, 13, 11, 10, 8, 7, 5, 4, 2, 1 ]
```

39.17.16 AscendingChain

▷ AscendingChain(*G, U*) (function)

This function computes an ascending chain of subgroups from *U* to *G*. This chain is given as a list whose first entry is *U* and the last entry is *G*. The function tries to make the links in this chain small.

The option refineIndex can be used to give a bound for refinements of steps to avoid GAP trying to enforce too small steps. The option cheap (if set to true) will overall limit the amount of heuristic searches.

39.17.17 IntermediateGroup

▷ IntermediateGroup(*G, U*) (function)

This routine tries to find a subgroup E of *G*, such that $G > E > U$ holds. If *U* is maximal in *G*, the function returns fail. This is done by finding minimal blocks for the operation of *G* on the right cosets of *U*.

39.17.18 IntermediateSubgroups

▷ IntermediateSubgroups(*G, U*) (operation)

returns a list of all subgroups of *G* that properly contain *U*; that is all subgroups between *G* and *U*. It returns a record with a component subgroups, which is a list of these subgroups, as well as a component inclusions, which lists all maximality inclusions among these subgroups. A maximality inclusion is given as a list $[i, j]$ indicating that the subgroup number i is a maximal subgroup of the subgroup number j, the numbers 0 and 1+ Length(subgroups) are used to denote *U* and *G*, respectively.

39.18 Factor Groups

39.18.1 NaturalHomomorphismByNormalSubgroup

▷ NaturalHomomorphismByNormalSubgroup(*G*, *N*) (function)
▷ NaturalHomomorphismByNormalSubgroupNC(*G*, *N*) (function)

returns a homomorphism from *G* to another group whose kernel is *N*. GAP will try to select the image group as to make computations in it as efficient as possible. As the factor group G/N can be identified with the image of *G* this permits efficient computations in the factor group. The homomorphism returned is not necessarily surjective, so ImagesSource (32.4.1) should be used instead of Range (32.3.7) to get a group isomorphic to the factor group. The NC variant does not check whether *N* is normal in *G*.

39.18.2 FactorGroup

▷ FactorGroup(*G*, *N*) (function)
▷ FactorGroupNC(*G*, *N*) (operation)

returns the image of the NaturalHomomorphismByNormalSubgroup(*G*,*N*). The homomorphism will be stored in the attribute NaturalHomomorphism of the result. The NC version does not test whether *N* is normal in *G*.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;n:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);;
gap> hom:=NaturalHomomorphismByNormalSubgroup(g,n);
[ (1,2,3,4), (1,2) ] -> [ f1*f2, f1 ]
gap> Size(ImagesSource(hom));
6
gap> FactorGroup(g,n);
Group([ f1, f2 ])
```

39.18.3 CommutatorFactorGroup

▷ CommutatorFactorGroup(*G*) (attribute)

computes the commutator factor group G/G' of the group *G*.

Example

```
gap> CommutatorFactorGroup(g);
Group([ f1 ])
```

39.18.4 MaximalAbelianQuotient

▷ MaximalAbelianQuotient(*G*) (attribute)

returns an epimorphism from *G* onto the maximal abelian quotient of *G*. The kernel of this epimorphism is the derived subgroup of *G*, see DerivedSubgroup (39.12.3).

39.18.5 HasAbelianFactorGroup

▷ `HasAbelianFactorGroup(G, N)` (operation)

tests whether G / N is abelian (without explicitly constructing the factor group).

39.18.6 HasElementaryAbelianFactorGroup

▷ `HasElementaryAbelianFactorGroup(G, N)` (operation)

tests whether G / N is elementary abelian (without explicitly constructing the factor group).

Example

```
gap> HasAbelianFactorGroup(g,n);
false
gap> HasAbelianFactorGroup(DerivedSubgroup(g),n);
true
```

39.18.7 CentralizerModulo

▷ `CentralizerModulo(G, N, elm)` (operation)

Computes the full preimage of the centralizer $C_{G/N}(elm \cdot N)$ in G (without necessarily constructing the factor group).

Example

```
gap> CentralizerModulo(g,n,(1,2));
Group([ (3,4), (1,3)(2,4), (1,4)(2,3) ])
```

39.19 Sets of Subgroups

39.19.1 ConjugacyClassSubgroups

▷ `ConjugacyClassSubgroups(G, U)` (operation)

generates the conjugacy class of subgroups of G with representative U. This class is an external set, so functions such as `Representative` (30.4.7), (which returns U), `ActingDomain` (41.12.3) (which returns G), `StabilizerOfExternalSet` (41.12.10) (which returns the normalizer of U), and `AsList` (30.3.8) work for it.

(The use of the `[]` list access to select elements of the class is considered obsolescent and will be removed in future versions. Use `ClassElementLattice` (39.20.2) instead.)

Example

```
gap> g:=Group((1,2,3,4),(1,2));;IsNaturalSymmetricGroup(g);;
gap> cl:=ConjugacyClassSubgroups(g,Subgroup(g,[(1,2)]));
Group( [ (1,2) ] )^G
gap> Size(cl);
6
gap> ClassElementLattice(cl,4);
Group([ (2,3) ])
```

39.19.2 IsConjugacyClassSubgroupsRep

▷ IsConjugacyClassSubgroupsRep(*obj*) (Representation)
▷ IsConjugacyClassSubgroupsByStabilizerRep(*obj*) (Representation)

Is the representation GAP uses for conjugacy classes of subgroups. It can be used to check whether an object is a class of subgroups. The second representation IsConjugacyClassSubgroupsByStabilizerRep in addition is an external orbit by stabilizer and will compute its elements via a transversal of the stabilizer.

39.19.3 ConjugacyClassesSubgroups

▷ ConjugacyClassesSubgroups(*G*) (attribute)

This attribute returns a list of all conjugacy classes of subgroups of the group *G*. It also is applicable for lattices of subgroups (see LatticeSubgroups (39.20.1)). The order in which the classes are listed depends on the method chosen by GAP. For each class of subgroups, a representative can be accessed using Representative (30.4.7).

Example

```
gap> ConjugacyClassesSubgroups(g);
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (3,4) ] )^G,
  Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
  Group( [ (3,4), (1,2)(3,4) ] )^G,
  Group( [ (1,3,2,4), (1,2)(3,4) ] )^G, Group( [ (3,4), (2,4,3) ] )^G,
  Group( [ (1,4)(2,3), (1,3)(2,4), (3,4) ] )^G,
  Group( [ (1,4)(2,3), (1,3)(2,4), (2,4,3) ] )^G,
  Group( [ (1,4)(2,3), (1,3)(2,4), (2,4,3), (3,4) ] )^G ]
```

39.19.4 ConjugacyClassesMaximalSubgroups

▷ ConjugacyClassesMaximalSubgroups(*G*) (attribute)

returns the conjugacy classes of maximal subgroups of *G*. Representatives of the classes can be computed directly by MaximalSubgroupClassReps (39.19.6).

Example

```
gap> ConjugacyClassesMaximalSubgroups(g);
[ AlternatingGroup( [ 1 .. 4 ] )^G, Group( [ (1,2,3), (1,2) ] )^G,
  Group( [ (1,2), (3,4), (1,3)(2,4) ] )^G ]
```

39.19.5 AllSubgroups

▷ AllSubgroups(*G*) (function)

For a finite group *G* AllSubgroups returns a list of all subgroups of *G*, intended primarily for use in class for small examples. This list will quickly get very long and in general use of ConjugacyClassesSubgroups (39.19.3) is recommended.

Example

```
gap> AllSubgroups(SymmetricGroup(3));
[ Group(()), Group([ (2,3) ]), Group([ (1,2) ]), Group([ (1,3) ]),
  Group([ (1,2,3) ]), Group([ (1,2,3), (2,3) ]) ]
```

39.19.6 MaximalSubgroupClassReps

▷ `MaximalSubgroupClassReps(G)` (attribute)

returns a list of conjugacy representatives of the maximal subgroups of `G`.

Example

```
gap> MaximalSubgroupClassReps(g);
[ Alt( [ 1 .. 4 ] ), Group([ (1,2,3), (1,2) ]),
  Group([ (1,2), (3,4), (1,3)(2,4) ]) ]
```

39.19.7 MaximalSubgroups

▷ `MaximalSubgroups(G)` (attribute)

returns a list of all maximal subgroups of `G`. This may take up much space, therefore the command should be avoided if possible. See `ConjugacyClassesMaximalSubgroups` (39.19.4).

Example

```
gap> MaximalSubgroups(Group((1,2,3),(1,2)));
[ Group([ (1,2,3) ]), Group([ (2,3) ]), Group([ (1,2) ]),
  Group([ (1,3) ]) ]
```

39.19.8 NormalSubgroups

▷ `NormalSubgroups(G)` (attribute)

returns a list of all normal subgroups of `G`.

Example

```
gap> g:=SymmetricGroup(4);;NormalSubgroups(g);
[ Sym( [ 1 .. 4 ] ), Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
```

The algorithm for the computation of normal subgroups is described in [Hul98].

39.19.9 MaximalNormalSubgroups

▷ `MaximalNormalSubgroups(G)` (attribute)

is a list containing those proper normal subgroups of the group `G` that are maximal among the proper normal subgroups.

Example

```
gap> MaximalNormalSubgroups( g );
[ Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]) ]
```

39.19.10 MinimalNormalSubgroups

▷ `MinimalNormalSubgroups(G)` (attribute)

is a list containing those nontrivial normal subgroups of the group `G` that are minimal among the nontrivial normal subgroups.

Example

```
gap> MinimalNormalSubgroups( g );
[ Group([ (1,4)(2,3), (1,3)(2,4) ]) ]
```

39.20 Subgroup Lattice

39.20.1 LatticeSubgroups

▷ LatticeSubgroups(*G*) (attribute)

computes the lattice of subgroups of the group *G*. This lattice has the conjugacy classes of subgroups as attribute ConjugacyClassesSubgroups (39.19.3) and permits one to test maximality/minimality relations.

Example

```
gap> g:=SymmetricGroup(4);;
gap> l:=LatticeSubgroups(g);
<subgroup lattice of Sym( [ 1 .. 4 ] ), 11 classes, 30 subgroups>
gap> ConjugacyClassesSubgroups(l);
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (3,4) ] )^G,
  Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
  Group( [ (3,4), (1,2)(3,4) ] )^G,
  Group( [ (1,3,2,4), (1,2)(3,4) ] )^G, Group( [ (3,4), (2,4,3) ] )^G,
  Group( [ (1,4)(2,3), (1,3)(2,4), (3,4) ] )^G,
  Group( [ (1,4)(2,3), (1,3)(2,4), (2,4,3) ] )^G,
  Group( [ (1,4)(2,3), (1,3)(2,4), (2,4,3), (3,4) ] )^G ]
```

39.20.2 ClassElementLattice

▷ ClassElementLattice(*C*, *n*) (operation)

For a class *C* of subgroups, obtained by a lattice computation, this operation returns the *n*-th conjugate subgroup in the class.

Because of other methods installed, calling AsList *(30.3.8) with* C *can give a different arrangement of the class elements!*

The GAP package XGAP permits a graphical display of the lattice of subgroups in a nice way.

39.20.3 DotFileLatticeSubgroups

▷ DotFileLatticeSubgroups(*L*, *file*) (function)

This function produces a graphical representation of the subgroup lattice *L* in file *file*. The output is in .dot (also known as GraphViz format). For details on the format, and information about how to display or edit this format see http://www.graphviz.org. (On the Macintosh, the program OmniGraffle is also able to read this format.)

Subgroups are labelled in the form *i*-*j* where *i* is the number of the class of subgroups and *j* the number within this class. Normal subgroups are represented by a box.

Example

```
gap> DotFileLatticeSubgroups(l,"s4lat.dot");
```

39.20.4 MaximalSubgroupsLattice

▷ MaximalSubgroupsLattice(*lat*) (attribute)

For a lattice *lat* of subgroups this attribute contains the maximal subgroup relations among the subgroups of the lattice. It is a list corresponding to the ConjugacyClassesSubgroups (39.19.3) value of the lattice, each entry giving a list of the maximal subgroups of the representative of this class. Every maximal subgroup is indicated by a list of the form $[c,n]$ which means that the n-th subgroup in class number c is a maximal subgroup of the representative.

The number n corresponds to access via ClassElementLattice (39.20.2) and *not* necessarily the AsList (30.3.8) arrangement! See also MinimalSupergroupsLattice (39.20.5).

Example

```
gap> MaximalSubgroupsLattice(l);
[ [  ], [ [ 1, 1 ] ], [ [ 1, 1 ] ], [ [ 1, 1 ] ],
  [ [ 2, 1 ], [ 2, 2 ], [ 2, 3 ] ], [ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ],
  [ [ 2, 3 ] ], [ [ 4, 1 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ] ],
  [ [ 7, 1 ], [ 6, 1 ], [ 5, 1 ] ],
  [ [ 5, 1 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ] ],
  [ [ 10, 1 ], [ 9, 1 ], [ 9, 2 ], [ 9, 3 ], [ 8, 1 ], [ 8, 2 ],
      [ 8, 3 ], [ 8, 4 ] ] ]
gap> last[6];
[ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ]
gap> u1:=Representative(ConjugacyClassesSubgroups(l)[6]);
Group([ (3,4), (1,2)(3,4) ])
gap> u2:=ClassElementLattice(ConjugacyClassesSubgroups(l)[3],1);;
gap> u3:=ClassElementLattice(ConjugacyClassesSubgroups(l)[3],6);;
gap> u4:=ClassElementLattice(ConjugacyClassesSubgroups(l)[2],3);;
gap> IsSubgroup(u1,u2);IsSubgroup(u1,u3);IsSubgroup(u1,u4);
true
true
true
```

39.20.5 MinimalSupergroupsLattice

▷ MinimalSupergroupsLattice(*lat*) (attribute)

For a lattice *lat* of subgroups this attribute contains the minimal supergroup relations among the subgroups of the lattice. It is a list corresponding to the ConjugacyClassesSubgroups (39.19.3) value of the lattice, each entry giving a list of the minimal supergroups of the representative of this class. Every minimal supergroup is indicated by a list of the form $[c,n]$, which means that the n-th subgroup in class number c is a minimal supergroup of the representative.

The number n corresponds to access via ClassElementLattice (39.20.2) and *not* necessarily the AsList (30.3.8) arrangement! See also MaximalSubgroupsLattice (39.20.4).

Example

```
gap> MinimalSupergroupsLattice(l);
[ [ [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ],
      [ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ],
      [ 4, 4 ] ], [ [ 5, 1 ], [ 6, 2 ], [ 7, 2 ] ],
  [ [ 6, 1 ], [ 8, 1 ], [ 8, 3 ] ], [ [ 8, 1 ], [ 10, 1 ] ],
  [ [ 9, 1 ], [ 9, 2 ], [ 9, 3 ], [ 10, 1 ] ], [ [ 9, 1 ] ],
```

```
    [ [ 9, 1 ] ], [ [ 11, 1 ] ], [ [ 11, 1 ] ], [ [ 11, 1 ] ], [  ] ]
gap> last[3];
[ [ 6, 1 ], [ 8, 1 ], [ 8, 3 ] ]
gap> u5:=ClassElementLattice(ConjugacyClassesSubgroups(l)[8],1);
Group([ (3,4), (2,4,3) ])
gap> u6:=ClassElementLattice(ConjugacyClassesSubgroups(l)[8],3);
Group([ (1,3), (1,3,4) ])
gap> IsSubgroup(u5,u2);
true
gap> IsSubgroup(u6,u2);
true
```

39.20.6 RepresentativesPerfectSubgroups

▷ `RepresentativesPerfectSubgroups(`*G*`)` (attribute)
▷ `RepresentativesSimpleSubgroups(`*G*`)` (attribute)

returns a list of conjugacy representatives of perfect (respectively simple) subgroups of *G*. This uses the library of perfect groups (see `PerfectGroup` (50.8.2)), thus it will issue an error if the library is insufficient to determine all perfect subgroups.

Example

```
gap> m11:=TransitiveGroup(11,6);
M(11)
gap> r:=RepresentativesPerfectSubgroups(m11);
[ Group([ (3,9,11)(4,8,6)(5,10,7), (1,3)(2,11)(4,6)(7,8) ]),
  Group([ (2,3,4)(5,6,8)(7,11,9), (1,3)(2,11)(4,6)(7,8) ]),
  Group([ (4,11,10,9)(5,7,8,6), (1,6,4)(2,9,7)(8,11,10) ]),
  Group([ (3,5,8)(4,11,7)(6,9,10), (1,3)(2,11)(4,6)(7,8) ]), M(11),
  Group(()) ]
gap> List(r,Size);
[ 60, 60, 360, 660, 7920, 1 ]
```

39.20.7 ConjugacyClassesPerfectSubgroups

▷ `ConjugacyClassesPerfectSubgroups(`*G*`)` (attribute)

returns a list of the conjugacy classes of perfect subgroups of *G*. (see `RepresentativesPerfectSubgroups` (39.20.6).)

Example

```
gap> ConjugacyClassesPerfectSubgroups(m11);
[ Group( [ ( 3, 9,11)( 4, 8, 6)( 5,10, 7),
      ( 1, 3)( 2,11)( 4, 6)( 7, 8) ] )^G,
  Group( [ ( 2, 3, 4)( 5, 6, 8)( 7,11, 9),
      ( 1, 3)( 2,11)( 4, 6)( 7, 8) ] )^G,
  Group( [ ( 4,11,10, 9)( 5, 7, 8, 6), ( 1, 6, 4)( 2, 9, 7)( 8,11,10)
     ] )^G,
  Group( [ ( 3, 5, 8)( 4,11, 7)( 6, 9,10),
      ( 1, 3)( 2,11)( 4, 6)( 7, 8) ] )^G, M(11)^G, Group( () )^G ]
```

39.20.8 Zuppos

▷ `Zuppos(G)` (attribute)

The *Zuppos* of a group are the cyclic subgroups of prime power order. (The name "Zuppo" derives from the German abbreviation for "zyklische Untergruppen von Primzahlpotenzordnung".) This attribute gives generators of all such subgroups of a group `G`. That is all elements of `G` of prime power order up to the equivalence that they generate the same cyclic subgroup.

39.20.9 InfoLattice

▷ `InfoLattice` (info class)

is the information class used by the cyclic extension methods for subgroup lattice calculations.

39.21 Specific Methods for Subgroup Lattice Computations

39.21.1 LatticeByCyclicExtension

▷ `LatticeByCyclicExtension(G[, func[, noperf]])` (function)

computes the lattice of `G` using the cyclic extension algorithm. If the function `func` is given, the algorithm will discard all subgroups not fulfilling `func` (and will also not extend them), returning a partial lattice. This can be useful to compute only subgroups with certain properties. Note however that this will *not* necessarily yield all subgroups that fulfill `func`, but the subgroups whose subgroups are used for the construction must also fulfill `func` as well. (In fact the filter `func` will simply discard subgroups in the cyclic extension algorithm. Therefore the trivial subgroup will always be included.) Also note, that for such a partial lattice maximality/minimality inclusion relations cannot be computed. (If `func` is a list of length 2, its first entry is such a discarding function, the second a function for discarding zuppos.)

The cyclic extension algorithm requires the perfect subgroups of `G`. However GAP cannot analyze the function `func` for its implication but can only apply it. If it is known that `func` implies solvability, the computation of the perfect subgroups can be avoided by giving a third parameter `noperf` set to `true`.

Example

```
gap> g:=WreathProduct(Group((1,2,3),(1,2)),Group((1,2,3,4)));;
gap> l:=LatticeByCyclicExtension(g,function(G)
> return Size(G) in [1,2,3,6];end);
<subgroup lattice of <permutation group of size 5184 with
9 generators>, 47 classes,
2628 subgroups, restricted under further condition l!.func>
```

The total number of classes in this example is much bigger, as the following example shows:

Example

```
gap> LatticeSubgroups(g);
<subgroup lattice of <permutation group of size 5184 with
9 generators>, 566 classes, 27134 subgroups>
```

##

39.21.2 InvariantSubgroupsElementaryAbelianGroup

▷ InvariantSubgroupsElementaryAbelianGroup(*G, homs[, dims]*) (function)

Let *G* be an elementary abelian group and *homs* be a set of automorphisms of *G*. Then this function computes all subspaces of *G* which are invariant under all automorphisms in *homs*. When considering *G* as a module for the algebra generated by *homs*, these are all submodules. If *homs* is empty, it computes all subgroups. If the optional parameter *dims* is given, only submodules of this dimension are computed.

Example

```
gap> g:=Group((1,2,3),(4,5,6),(7,8,9));
Group([ (1,2,3), (4,5,6), (7,8,9) ])
gap> hom:=GroupHomomorphismByImages(g,g,[(1,2,3),(4,5,6),(7,8,9)],
> [(7,8,9),(1,2,3),(4,5,6)]);
[ (1,2,3), (4,5,6), (7,8,9) ] -> [ (7,8,9), (1,2,3), (4,5,6) ]
gap> u:=InvariantSubgroupsElementaryAbelianGroup(g,[hom]);
[ Group(()), Group([ (1,2,3)(4,5,6)(7,8,9) ]),
  Group([ (1,3,2)(7,8,9), (1,3,2)(4,5,6) ]),
  Group([ (7,8,9), (4,5,6), (1,2,3) ]) ]
```

39.21.3 SubgroupsSolvableGroup

▷ SubgroupsSolvableGroup(*G[, opt]*) (function)

This function (implementing the algorithm published in [Hul99]) computes subgroups of a solvable group *G*, using the homomorphism principle. It returns a list of representatives up to *G*-conjugacy.

The optional argument *opt* is a record, which may be used to put restrictions on the subgroups computed. The following record components of *opt* are recognized and have the following effects:

`actions`
: must be a list of automorphisms of *G*. If given, only groups which are invariant under all these automorphisms are computed. The algorithm must know the normalizer in *G* of the group generated by `actions` (defined formally by embedding in the semidirect product of *G* with *actions*). This can be given in the component `funcnorm` and will be computed if this component is not given.

`normal`
: if set to `true` only normal subgroups are guaranteed to be returned (though some of the returned subgroups might still be not normal).

`consider`
: a function to restrict the groups computed. This must be a function of five parameters, C, A, N, B, M, that are interpreted as follows: The arguments are subgroups of a factor F of *G* in the relation $F \geq C > A > N > B > M$. N and M are normal subgroups. C is the full preimage of the normalizer of A/N in F/N. When computing modulo M and looking for subgroups U such that $U \cap N = B$ and $\langle U, N \rangle = A$, this function is called. If it returns `false` then all potential groups U (and therefore all groups later arising from them) are disregarded. This can be used for example to compute only subgroups of certain sizes.

(*This is just a restriction to speed up computations. The function may still return (invariant) subgroups which don't fulfill this condition!*) This parameter is used to permit calculations of some subgroups if the set of all subgroups would be too large to handle.

The actual groups C, A, N and B which are passed to this function are not necessarily subgroups of `G` but might be subgroups of a proper factor group $F = G/H$. Therefore the `consider` function may not relate the parameter groups to `G`.

`retnorm`
: if set to `true` the function not only returns a list `subs` of subgroups but also a corresponding list `norms` of normalizers in the form `[ subs, norms ]`.

`series`
: is an elementary abelian series of `G` which will be used for the computation.

`groups`
: is a list of groups to seed the calculation. Only subgroups of these groups are constructed.

Example

```
gap> g:=Group((1,2,3),(1,2),(4,5,6),(4,5),(7,8,9),(7,8));
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ])
gap> hom:=GroupHomomorphismByImages(g,g,
> [(1,2,3),(1,2),(4,5,6),(4,5),(7,8,9),(7,8)],
> [(4,5,6),(4,5),(7,8,9),(7,8),(1,2,3),(1,2)]);
[ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ] ->
[ (4,5,6), (4,5), (7,8,9), (7,8), (1,2,3), (1,2) ]
gap> l:=SubgroupsSolvableGroup(g,rec(actions:=[hom]));;
gap> List(l,Size);
[ 1, 3, 9, 27, 54, 2, 6, 18, 108, 4, 216, 8 ]
gap> Length(ConjugacyClassesSubgroups(g)); # to compare
162
```

39.21.4 SizeConsiderFunction

▷ `SizeConsiderFunction(`*size*`)` (function)

This function returns a function `consider` of four arguments that can be used in `SubgroupsSolvableGroup` (39.21.3) for the option `consider` to compute subgroups whose sizes are divisible by *size*.

Example

```
gap> l:=SubgroupsSolvableGroup(g,rec(actions:=[hom],
> consider:=SizeConsiderFunction(6)));;
gap> List(l,Size);
[ 1, 3, 9, 27, 54, 6, 18, 108, 216 ]
```

This example shows that in general the `consider` function does not provide a perfect filter. It is guaranteed that all subgroups fulfilling the condition are returned, but not all subgroups returned necessarily fulfill the condition.

39.21.5 ExactSizeConsiderFunction

▷ `ExactSizeConsiderFunction(size)` (function)

This function returns a function `consider` of four arguments that can be used in `SubgroupsSolvableGroup` (39.21.3) for the option `consider` to compute subgroups whose sizes are exactly *size*.

Example

```
gap> l:=SubgroupsSolvableGroup(g,rec(actions:=[hom],
> consider:=ExactSizeConsiderFunction(6)));;
gap> List(l,Size);
[ 1, 3, 9, 27, 54, 6, 108, 216 ]
```

Again, the `consider` function does not provide a perfect filter. It is guaranteed that all subgroups fulfilling the condition are returned, but not all subgroups returned necessarily fulfill the condition.

39.21.6 InfoPcSubgroup

▷ `InfoPcSubgroup` (info class)

Information function for the subgroup lattice functions using pcgs.

39.22 Special Generating Sets

39.22.1 GeneratorsSmallest

▷ `GeneratorsSmallest(G)` (attribute)

returns a "smallest" generating set for the group `G`. This is the lexicographically (using GAPs order of group elements) smallest list l of elements of `G` such that $G = \langle l \rangle$ and $l_i \notin \langle l_1, \ldots, l_{i-1} \rangle$ (in particular l_1 is not the identity element of the group). The comparison of two groups via lexicographic comparison of their sorted element lists yields the same relation as lexicographic comparison of their smallest generating sets.

Example

```
gap> g:=SymmetricGroup(4);;
gap> GeneratorsSmallest(g);
[ (3,4), (2,3), (1,2) ]
```

39.22.2 LargestElementGroup

▷ `LargestElementGroup(G)` (attribute)

returns the largest element of `G` with respect to the ordering < of the elements family.

39.22.3 MinimalGeneratingSet

▷ `MinimalGeneratingSet(G)` (attribute)

returns a generating set of `G` of minimal possible length.

Note that –apart from special cases– currently there are only efficient methods known to compute minimal generating sets of finite solvable groups and of finitely generated nilpotent groups. Hence so far these are the only cases for which methods are available. The former case is covered by a method implemented in the GAP library, while the second case requires the package Polycyclic.

If you do not really need a minimal generating set, but are satisfied with getting a reasonably small set of generators, you better use `SmallGeneratingSet` (39.22.4).

Information about the minimal generating sets of the finite simple groups of order less than 10^6 can be found in [MY79]. See also the package AtlasRep.

Example

```
gap> MinimalGeneratingSet(g);
[ (2,4,3), (1,4,2,3) ]
```

39.22.4 SmallGeneratingSet

▷ `SmallGeneratingSet(G)` (attribute)

returns a generating set of `G` which has few elements. As neither irredundancy, nor minimal length is proven it runs much faster than `MinimalGeneratingSet` (39.22.3). It can be used whenever a short generating set is desired which not necessarily needs to be optimal.

Example

```
gap> SmallGeneratingSet(g);
[ (1,2,3,4), (1,2) ]
```

39.22.5 IndependentGeneratorsOfAbelianGroup

▷ `IndependentGeneratorsOfAbelianGroup(A)` (attribute)

returns a list of generators $a_1, a_2, \ldots$ of prime power order or infinite order of the abelian group `A` such that `A` is the direct product of the cyclic groups generated by the a_i. The list of orders of the returned generators must match the result of `AbelianInvariants` (39.16.1) (taking into account that zero and `infinity` (18.2.1) are identified).

Example

```
gap> g:=AbelianGroup(IsPermGroup,[15,14,22,78]);;
gap> List(IndependentGeneratorsOfAbelianGroup(g),Order);
[ 2, 2, 2, 3, 3, 5, 7, 11, 13 ]
gap> AbelianInvariants(g);
[ 2, 2, 2, 3, 3, 5, 7, 11, 13 ]
```

39.22.6 IndependentGeneratorExponents

▷ `IndependentGeneratorExponents(G, g)` (operation)

For an abelian group `G`, with `IndependentGeneratorsOfAbelianGroup` (39.22.5) value the list $[a_1, \ldots, a_n]$, this operation returns the exponent vector $[e_1, \ldots, e_n]$ to represent $g = \prod_i a_i^{e_i}$.

Example

```
gap> g := AbelianGroup([16,9,625]);;
gap> gens := IndependentGeneratorsOfAbelianGroup(g);;
gap> List(gens, Order);
```

```
[ 9, 16, 625 ]
gap> AbelianInvariants(g);
[ 9, 16, 625 ]
gap> r:=gens[1]^4*gens[2]^12*gens[3]^128;;
gap> IndependentGeneratorExponents(g,r);
[ 4, 12, 128 ]
```

39.23 1-Cohomology

Let G be a finite group and M an elementary abelian normal p-subgroup of G. Then the group of 1-cocycles $Z^1(G/M,M)$ is defined as

$$Z^1(G/M,M) = \{\gamma : G/M \to M \mid \forall g_1,g_2 \in G : \gamma(g_1M \cdot g_2M) = \gamma(g_1M)^{g_2} \cdot \gamma(g_2M)\}$$

and is a $GF(p)$-vector space.

The group of 1-coboundaries $B^1(G/M,M)$ is defined as

$$B^1(G/M,M) = \{\gamma : G/M \to M \mid \exists m \in M \forall g \in G : \gamma(gM) = (m^{-1})^g \cdot m\}$$

It also is a $GF(p)$-vector space.

Let α be the isomorphism of M into a row vector space $\mathscr{W}$ and $(g_1,\ldots,g_l)$ representatives for a generating set of G/M. Then there exists a monomorphism β of $Z^1(G/M,M)$ in the l-fold direct sum of $\mathscr{W}$, such that $\beta(\gamma) = (\alpha(\gamma(g_1M)),\ldots,\alpha(\gamma(g_lM)))$ for every $\gamma \in Z^1(G/M,M)$.

39.23.1 OneCocycles

▷ `OneCocycles(G, M)` (operation)
▷ `OneCocycles(G, mpcgs)` (operation)
▷ `OneCocycles(gens, M)` (operation)
▷ `OneCocycles(gens, mpcgs)` (operation)

Computes the group of 1-cocycles $Z^1(G/M,M)$. The normal subgroup `M` may be given by a (Modulo)Pcgs *mpcgs*. In this case the whole calculation is performed modulo the normal subgroup defined by `DenominatorOfModuloPcgs(`*mpcgs*`)` (see 45.1). Similarly the group `G` may instead be specified by a set of elements *gens* that are representatives for a generating system for the factor group `G/M`. If this is done the 1-cocycles are computed with respect to these generators (otherwise the routines try to select suitable generators themselves). The current version of the code assumes that `G` is a permutation group or a pc group.

39.23.2 OneCoboundaries

▷ `OneCoboundaries(G, M)` (operation)

computes the group of 1-coboundaries. Syntax of input and output otherwise is the same as with `OneCocycles` (39.23.1) except that entries that refer to cocycles are not computed.

The operations `OneCocycles` (39.23.1) and `OneCoboundaries` return a record with (at least) the components:

`generators`
: Is a list of representatives for a generating set of `G/M`. Cocycles are represented with respect to these generators.

`oneCocycles`
: A space of row vectors over GF(p), representing Z^1. The vectors are represented in dimension $a \cdot b$ where a is the length of `generators` and p^b the size of `M`.

`oneCoboundaries`
: A space of row vectors that represents B^1.

`cocycleToList`
: is a function to convert a cocycle (a row vector in `oneCocycles`) to a corresponding list of elements of `M`.

`listToCocycle`
: is a function to convert a list of elements of `M` to a cocycle.

`isSplitExtension`
: indicates whether `G` splits over `M`. The following components are only bound if the extension splits. Note that if `M` is given by a modulo pcgs all subgroups are given as subgroups of `G` by generators corresponding to `generators` and thus may not contain the denominator of the modulo pcgs. In this case taking the closure with this denominator will give the full preimage of the complement in the factor group.

`complement`
: One complement to `M` in `G`.

`cocycleToComplement( cyc )`
: is a function that takes a cocycle from `oneCocycles` and returns the corresponding complement to `M` in `G` (with respect to the fixed complement `complement`).

`complementToCocycle(U)`
: is a function that takes a complement and returns the corresponding cocycle.

If the factor `G/M` is given by a (modulo) pcgs *`gens`* then special methods are used that compute a presentation for the factor implicitly from the pcgs.

Note that the groups of 1-cocycles and 1-coboundaries are not groups in the sense of `Group` (39.2.1) for GAP but vector spaces.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> n:=Group((1,2)(3,4),(1,3)(2,4));;
gap> oc:=OneCocycles(g,n);
rec( cocycleToComplement := function( c ) ... end,
  cocycleToList := function( c ) ... end,
  complement := Group([ (3,4), (2,4,3) ]),
  complementGens := [ (3,4), (2,4,3) ],
  complementToCocycle := function( K ) ... end,
  factorGens := [ (3,4), (2,4,3) ], generators := [ (3,4), (2,4,3) ],
  isSplitExtension := true, listToCocycle := function( L ) ... end,
  oneCoboundaries := <vector space over GF(2), with 2 generators>,
```

```
  oneCocycles := <vector space over GF(2), with 2 generators> )
gap> oc.cocycleToList([ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]);
[ (1,2)(3,4), (1,2)(3,4) ]
gap> oc.listToCocycle([(),(1,3)(2,4)]) = Z(2) * [ 0, 0, 1, 0];
true
gap> oc.cocycleToComplement([ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]);
Group([ (3,4), (1,3,4) ])
gap> oc.complementToCocycle(Group((1,2,4),(1,4))) = Z(2) * [ 0, 1, 1, 1 ];
true
```

The factor group $H^1(G/M,M) = Z^1(G/M,M)/B^1(G/M,M)$ is called the first cohomology group. Currently there is no function which explicitly computes this group. The easiest way to represent it is as a vector space complement to B^1 in Z^1.

If the only purpose of the calculation of H^1 is the determination of complements it might be desirable to stop calculations once it is known that the extension cannot split. This can be achieved via the more technical function `OCOneCocycles` (39.23.3).

39.23.3 OCOneCocycles

▷ `OCOneCocycles(`*ocr, onlySplit*`)` (operation)

is the more technical function to compute 1-cocycles. It takes an record *ocr* as first argument which must contain at least the components `group` for the group and `modulePcgs` for a (modulo) pcgs of the module. This record will also be returned with components as described under `OneCocycles` (39.23.1) (with the exception of `isSplitExtension` which is indicated by the existence of a `complement`) but components such as `oneCoboundaries` will only be computed if not already present.

If *onlySplit* is `true`, `OCOneCocycles` returns `false` as soon as possible if the extension does not split.

39.23.4 ComplementClassesRepresentativesEA

▷ `ComplementClassesRepresentativesEA(`*G, N*`)` (operation)

computes complement classes to an elementary abelian normal subgroup *N* via 1-Cohomology. Normally, a user program should call `ComplementClassesRepresentatives` (39.11.6) instead, which also works for a solvable (not necessarily elementary abelian) *N*.

39.23.5 InfoCoh

▷ `InfoCoh` (info class)

The info class for the cohomology calculations is `InfoCoh`.

39.24 Schur Covers and Multipliers

Additional attributes and properties of a group can be derived from computing its Schur cover. For example, if G is a finitely presented group, the derived subgroup of a Schur cover of G is invariant

and isomorphic to the NonabelianExteriorSquare (39.24.5) value of G, see [BJR87].

39.24.1 EpimorphismSchurCover

▷ EpimorphismSchurCover(*G[, pl]*) (operation)

returns an epimorphism *epi* from a group D onto *G*. The group D is one (of possibly several) Schur covers of *G*. The group D can be obtained as the Source (32.3.8) value of *epi*. The kernel of *epi* is the Schur multiplier of *G*. If *pl* is given as a list of primes, only the multiplier part for these primes is realized. At the moment, D is represented as a finitely presented group.

39.24.2 SchurCover

▷ SchurCover(*G*) (operation)

returns one (of possibly several) Schur covers of the group *G*.

At the moment this cover is represented as a finitely presented group and IsomorphismPermGroup (43.3.1) would be needed to convert it to a permutation group.

If also the relation to *G* is needed, EpimorphismSchurCover (39.24.1) should be used.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> epi:=EpimorphismSchurCover(g);
[ f1, f2, f3 ] -> [ (3,4), (2,4,3), (1,3)(2,4) ]
gap> Size(Source(epi));
48
```

If the group becomes bigger, Schur Cover calculations might become unfeasible.

There is another operation, AbelianInvariantsMultiplier (39.24.3), which only returns the structure of the Schur Multiplier, and which should work for larger groups as well.

39.24.3 AbelianInvariantsMultiplier

▷ AbelianInvariantsMultiplier(*G*) (attribute)

returns a list of the abelian invariants of the Schur multiplier of *G*.

At the moment, this operation will not give any information about how to extend the multiplier to a Schur Cover.

Example

```
gap> AbelianInvariantsMultiplier(g);
[ 2 ]
gap> AbelianInvariantsMultiplier(AlternatingGroup(6));
[ 2, 3 ]
gap> AbelianInvariantsMultiplier(SL(2,3));
[  ]
gap> AbelianInvariantsMultiplier(SL(3,2));
[ 2 ]
gap> AbelianInvariantsMultiplier(PSU(4,2));
[ 2 ]
```

(Note that the last command from the example will take some time.)

The GAP 4.4.12 manual contained examples for larger groups e.g. M_{22}. However, some issues that may very rarely (and not easily reproducibly) lead to wrong results were discovered in the code capable of handling larger groups, and in GAP 4.5 it was replaced by a more reliable basic method. To deal with larger groups, one can use the function `SchurMultiplier` (**cohomolo: SchurMultiplier**) from the cohomolo package. Also, additional methods for `AbelianInvariantsMultiplier` are installed in the Polycyclic package for pcp-groups.

39.24.4 Epicentre

▷ `Epicentre(G)` (attribute)
▷ `ExteriorCentre(G)` (attribute)

There are various ways of describing the epicentre of a group `G`. It is the smallest normal subgroup N of `G` such that G/N is a central quotient of a group. It is also equal to the Exterior Center of `G`, see [Ell98].

39.24.5 NonabelianExteriorSquare

▷ `NonabelianExteriorSquare(G)` (operation)

Computes the nonabelian exterior square $G \wedge G$ of the group `G`, which for a finitely presented group is the derived subgroup of any Schur cover of `G` (see [BJR87]).

39.24.6 EpimorphismNonabelianExteriorSquare

▷ `EpimorphismNonabelianExteriorSquare(G)` (operation)

Computes the mapping $G \wedge G \to G$. The kernel of this mapping is equal to the Schur multiplier of `G`.

39.24.7 IsCentralFactor

▷ `IsCentralFactor(G)` (property)

This function determines if there exists a group H such that `G` is isomorphic to the quotient $H/Z(H)$. A group with this property is called in literature *capable*. A group being capable is equivalent to the epicentre of `G` being trivial, see [BFS79].

39.24.8 Covering groups of symmetric groups

The covering groups of symmetric groups were classified in [Sch11]; an inductive procedure to construct faithful, irreducible representations of minimal degree over all fields was presented in [Maa10]. Methods for `EpimorphismSchurCover` (39.24.1) are provided for natural symmetric groups which use these representations. For alternating groups, the restriction of these representations are provided, but they may not be irreducible. In the case of degree 6 and 7, they are not the full covering groups and so matrix representations are just stored explicitly for the six-fold covers.

Example

```
gap> EpimorphismSchurCover(SymmetricGroup(15));
[ < immutable compressed matrix 64x64 over GF(9) >,
  < immutable compressed matrix 64x64 over GF(9) > ] ->
[ (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15), (1,2) ]
gap> EpimorphismSchurCover(AlternatingGroup(15));
[ < immutable compressed matrix 64x64 over GF(9) >,
  < immutable compressed matrix 64x64 over GF(9) > ] ->
[ (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15), (13,14,15) ]
gap> SchurCoverOfSymmetricGroup(12);
<matrix group of size 958003200 with 2 generators>
gap> DoubleCoverOfAlternatingGroup(12);
<matrix group of size 479001600 with 2 generators>
gap> BasicSpinRepresentationOfSymmetricGroup( 10, 3, -1 );
[ < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) >,
  < immutable compressed matrix 16x16 over GF(9) > ]
```

39.24.9 BasicSpinRepresentationOfSymmetricGroup

▷ `BasicSpinRepresentationOfSymmetricGroup(`*n, p, sign*`)` (function)

Constructs the image of the Coxeter generators in the basic spin (projective) representation of the symmetric group of degree *n* over a field of characteristic $p \geq 0$. There are two such representations and *sign* controls which is returned: +1 gives a group where the preimage of an adjacent transposition $(i, i+1)$ has order 4, -1 gives a group where the preimage of an adjacent transposition $(i, i+1)$ has order 2. If no *sign* is specified, +1 is used by default. If no *p* is specified, 3 is used by default.

39.24.10 SchurCoverOfSymmetricGroup

▷ `SchurCoverOfSymmetricGroup(`*n, p, sign*`)` (operation)

Constructs a Schur cover of `SymmetricGroup(`*n*`)` as a faithful, irreducible matrix group in characteristic *p* ($p \neq 2$). For $n \geq 4$, there are two such covers, and *sign* determines which is returned: +1 gives a group where the preimage of an adjacent transposition $(i, i+1)$ has order 4, -1 gives a group where the preimage of an adjacent transposition $(i, i+1)$ has order 2. If no *sign* is specified, +1 is used by default. If no *p* is specified, 3 is used by default. For $n \leq 3$, the symmetric group is its own Schur cover and *sign* is ignored. For $p = 2$, there is no faithful, irreducible representation of the Schur cover unless $n = 1$ or $n = 3$, so `fail` is returned if $p = 2$. For $p = 3$, $n = 3$, the representation is indecomposable, but reducible. The field of the matrix group is generally `GF(`*p*`^2)` if $p > 0$, and an abelian number field if $p = 0$.

39.24.11 DoubleCoverOfAlternatingGroup

▷ `DoubleCoverOfAlternatingGroup(n, p)` (operation)

Constructs a double cover of `AlternatingGroup(n)` as a faithful, completely reducible matrix group in characteristic `p` ($p \neq 2$) for $n \geq 4$. For $n \geq 3$, the symmetric group is its own Schur cover so `fail` is returned. For $p = 2$, there is no faithful, completely reducible representation of the double cover, so `fail` is returned. The field of the matrix group is generally `GF(p^2)` if $p > 0$, and an abelian number field if $p = 0$. If `p` is omitted, the default is 3.

39.25 Tests for the Availability of Methods

The following filters and operations indicate capabilities of GAP. They can be used in the method selection or algorithms to check whether it is feasible to compute certain operations for a given group. In general, they return `true` if good algorithms for the given arguments are available in GAP. An answer `false` indicates that no method for this group may exist, or that the existing methods might run into problems.

Typical examples when this might happen is with finitely presented groups, for which many of the methods cannot be guaranteed to succeed in all situations.

The willingness of GAP to perform certain operations may change, depending on which further information is known about the arguments. Therefore the filters used are not implemented as properties but as "other filters" (see 13.7 and 13.8).

39.25.1 CanEasilyTestMembership

▷ `CanEasilyTestMembership(G)` (function)

This filter indicates whether GAP can test membership of elements in the group `G` (via the operation `\in` (30.6.1)) in reasonable time. It is used by the method selection to decide whether an algorithm that relies on membership tests may be used.

39.25.2 CanEasilyComputeWithIndependentGensAbelianGroup

▷ `CanEasilyComputeWithIndependentGensAbelianGroup(G)` (function)

This filter indicates whether GAP can in reasonable time compute independent abelian generators of the group `G` (via `IndependentGeneratorsOfAbelianGroup` (39.22.5)) and then can decompose arbitrary group elements with respect to these generators using `IndependentGeneratorExponents` (39.22.6). It is used by the method selection to decide whether an algorithm that relies on these two operations may be used.

39.25.3 CanComputeSize

▷ `CanComputeSize(dom)` (function)

This filter indicates whether the size of the domain `dom` (which might be `infinity` (18.2.1)) can be computed.

39.25.4 CanComputeSizeAnySubgroup

▷ `CanComputeSizeAnySubgroup(`*G*`)` (function)

This filter indicates whether GAP can easily compute the size of any subgroup of the group *G*. (This is for example advantageous if one can test that a stabilizer index equals the length of the orbit computed so far to stop early.)

39.25.5 CanComputeIndex

▷ `CanComputeIndex(`*G, H*`)` (function)

This function indicates whether the index $[G : H]$ (which might be `infinity` (18.2.1)) can be computed. It assumes that $H \leq G$ (see `CanComputeIsSubset` (39.25.6)).

39.25.6 CanComputeIsSubset

▷ `CanComputeIsSubset(`*A, B*`)` (operation)

This filter indicates that GAP can test (via `IsSubset` (30.5.1)) whether *B* is a subset of *A*.

39.25.7 KnowsHowToDecompose

▷ `KnowsHowToDecompose(`*G[, gens]*`)` (property)

Tests whether the group *G* can decompose elements in the generators *gens*. If *gens* is not given it tests, whether it can decompose in the generators given in the `GeneratorsOfGroup` (39.2.4) value of *G*.

This property can be used for example to check whether a group homomorphism by images (see `GroupHomomorphismByImages` (40.1.1)) can be reasonably defined from this group.

Chapter 40

Group Homomorphisms

A group homomorphism is a mapping from one group to another that respects multiplication and inverses. They are implemented as a special class of mappings, so in particular all operations for mappings, such as `Image` (32.4.6), `PreImage` (32.5.6), `PreImagesRepresentative` (32.5.4), `KernelOfMultiplicativeGeneralMapping` (32.9.5), `Source` (32.3.8), `Range` (32.3.7), `IsInjective` (32.3.4) and `IsSurjective` (32.3.5) (see chapter 32, in particular section 32.9) are applicable to them.

Homomorphisms can be used to transfer calculations into isomorphic groups in another representation, for which better algorithms are available. Section 40.5 explains a technique how to enforce this automatically.

Homomorphisms are also used to represent group automorphisms, and section 40.6 explains explains GAP's facilities to work with automorphism groups.

Section 40.9 explains how to make GAP to search for all homomorphisms between two groups which fulfill certain specifications.

40.1 Creating Group Homomorphisms

The most important way of creating group homomorphisms is to give images for a set of group generators and to extend it to the group generated by them by the homomorphism property.

A second way to create homomorphisms is to give functions that compute image and preimage. (A similar case are homomorphisms that are induced by conjugation. Special constructors for such mappings are described in section 40.6).

The third class are epimorphisms from a group onto its factor group. Such homomorphisms can be constructed by `NaturalHomomorphismByNormalSubgroup` (39.18.1).

The fourth class is homomorphisms in a permutation group that are induced by an action on a set. Such homomorphisms are described in the context of group actions, see chapter 41 and in particular `ActionHomomorphism` (41.7.1).

40.1.1 GroupHomomorphismByImages

▷ `GroupHomomorphismByImages(`*G, H[[, gens], imgs]*`)` (function)

`GroupHomomorphismByImages` returns the group homomorphism with source *G* and range *H* that is defined by mapping the list *gens* of generators of *G* to the list *imgs* of images in *H*.

If omitted, the arguments *gens* and *imgs* default to the GeneratorsOfGroup (39.2.4) value of *G* and *H*, respectively. If *H* is not given the maopping is automatically considered as surjective.

If *gens* does not generate *G* or if the mapping of the generators does not extend to a homomorphism (i.e., if mapping the generators describes only a multi-valued mapping) then fail is returned.

This test can be quite expensive. If one is certain that the mapping of the generators extends to a homomorphism, one can avoid the checks by calling GroupHomomorphismByImagesNC (40.1.2). (There also is the possibility to construct potentially multi-valued mappings with GroupGeneralMappingByImages (40.1.3) and to test with IsMapping (32.3.3) whether they are indeed homomorphisms.)

40.1.2 GroupHomomorphismByImagesNC

▷ GroupHomomorphismByImagesNC(*G, H[[, gens], imgs]*) (operation)

GroupHomomorphismByImagesNC creates a homomorphism as GroupHomomorphismByImages (40.1.1) does, however it does not test whether *gens* generates *G* and that the mapping of *gens* to *imgs* indeed defines a group homomorphism. Because these tests can be expensive it can be substantially faster than GroupHomomorphismByImages (40.1.1). Results are unpredictable if the conditions do not hold.

If omitted, the arguments *gens* and *imgs* default to the GeneratorsOfGroup (39.2.4) value of *G* and *H*, respectively.

(For creating a possibly multi-valued mapping from *G* to *H* that respects multiplication and inverses, GroupGeneralMappingByImages (40.1.3) can be used.)

Example

```
gap> gens:=[(1,2,3,4),(1,2)];
[ (1,2,3,4), (1,2) ]
gap> g:=Group(gens);
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> hom:=GroupHomomorphismByImages(g,h,gens,[(1,2),(1,3)]);
[ (1,2,3,4), (1,2) ] -> [ (1,2), (1,3) ]
gap> Image(hom,(1,4));
(2,3)
gap> map:=GroupHomomorphismByImages(g,h,gens,[(1,2,3),(1,2)]);
fail
```

40.1.3 GroupGeneralMappingByImages

▷ GroupGeneralMappingByImages(*G, H, gens, imgs*) (operation)
▷ GroupGeneralMappingByImages(*G, gens, imgs*) (operation)
▷ GroupGeneralMappingByImagesNC(*G, H, gens, imgs*) (operation)
▷ GroupGeneralMappingByImagesNC(*G, gens, imgs*) (operation)

returns a general mapping defined by extending the mapping from *gens* to *imgs* homomorphically. If the range *H* is not given the mapping will be made automatically surjective. The NC version does not test whether *gens* are contained in *G* or *imgs* are contained in *H*.

(GroupHomomorphismByImages (40.1.1) creates a group general mapping by images and tests whether it is in IsMapping (32.3.3).)

Example

```
gap> map:=GroupGeneralMappingByImages(g,h,gens,[(1,2,3),(1,2)]);
[ (1,2,3,4), (1,2) ] -> [ (1,2,3), (1,2) ]
gap> IsMapping(map);
false
```

40.1.4 GroupHomomorphismByFunction

▷ GroupHomomorphismByFunction(*S, R, fun[, invfun]*) (function)

▷ GroupHomomorphismByFunction(*S, R, fun, false, prefun*) (function)

GroupHomomorphismByFunction returns a group homomorphism hom with source *S* and range *R*, such that each element s of *S* is mapped to the element *fun*(s), where *fun* is a GAP function.

If the argument *invfun* is bound then *hom* is a bijection between *S* and *R*, and the preimage of each element r of *R* is given by *invfun*(r), where *invfun* is a GAP function.

If five arguments are given and the fourth argument is false then the GAP function *prefun* can be used to compute a single preimage also if hom is not bijective.

No test is performed on whether the functions actually give an homomorphism between both groups because this would require testing the full multiplication table.

GroupHomomorphismByFunction creates a mapping which lies in IsSPGeneralMapping (32.14.1).

Example

```
gap> hom:=GroupHomomorphismByFunction(g,h,
> function(x) if SignPerm(x)=-1 then return (1,2); else return ();fi;end);
MappingByFunction( Group([ (1,2,3,4), (1,2) ]), Group(
[ (1,2,3), (1,2) ]), function( x ) ... end )
gap> ImagesSource(hom);
Group([ (1,2), (1,2) ])
gap> Image(hom,(1,2,3,4));
(1,2)
```

40.1.5 AsGroupGeneralMappingByImages

▷ AsGroupGeneralMappingByImages(*map*) (attribute)

If *map* is a mapping from one group to another this attribute returns a group general mapping that which implements the same abstract mapping. (Some operations can be performed more effective in this representation, see also IsGroupGeneralMappingByAsGroupGeneralMappingByImages (40.10.3).)

Example

```
gap> AsGroupGeneralMappingByImages(hom);
[ (1,2,3,4), (1,2) ] -> [ (1,2), (1,2) ]
```

40.2 Operations for Group Homomorphisms

Group homomorphisms are mappings, so all the operations and properties for mappings described in chapter 32 are applicable to them. (However often much better methods, than for general mappings are available.)

Group homomorphisms will map groups to groups by just mapping the set of generators.

`KernelOfMultiplicativeGeneralMapping` (32.9.5) can be used to compute the kernel of a group homomorphism.

Example

```
gap> hom:=GroupHomomorphismByImages(g,h,gens,[(1,2),(1,3)]);;
gap> Kernel(hom);
Group([ (1,4)(2,3), (1,2)(3,4) ])
```

Homomorphisms can map between groups in different representations and are also used to get isomorphic groups in a different representation.

Example

```
gap> m1:=[[0,-1],[1,0]];;m2:=[[0,-1],[1,1]];;
gap> sl2z:=Group(m1,m2);; # SL(2,Integers) as matrix group
gap> F:=FreeGroup(2);;
gap> psl2z:=F/[F.1^2,F.2^3]; #PSL(2,Z) as FP group
<fp group on the generators [ f1, f2 ]>
gap> phom:=GroupHomomorphismByImagesNC(sl2z,psl2z,[m1,m2],
> GeneratorsOfGroup(psl2z)); # the non NC-version would be expensive
[ [ [ 0, -1 ], [ 1, 0 ] ], [ [ 0, -1 ], [ 1, 1 ] ] ] -> [ f1, f2 ]
gap> Kernel(phom); # the diagonal matrices
Group([ [ [ -1, 0 ], [ 0, -1 ] ], [ [ -1, 0 ], [ 0, -1 ] ] ])
gap> p1:=(1,2)(3,4);;p2:=(2,4,5);;a5:=Group(p1,p2);;
gap> ahom:=GroupHomomorphismByImages(psl2z,a5,
> GeneratorsOfGroup(psl2z),[p1,p2]); # here homomorphism test is cheap.
[ f1, f2 ] -> [ (1,2)(3,4), (2,4,5) ]
gap> u:=PreImage(ahom,Group((1,2,3),(1,2)(4,5)));
Group(<fp, no generators known>)
gap> Index(psl2z,u);
10
gap> isofp:=IsomorphismFpGroup(u);; Image(isofp);
<fp group of size infinity on the generators [ F1, F2, F3, F4 ]>
gap> RelatorsOfFpGroup(Image(isofp));
[ F1^2, F4^2, F3^3 ]
gap> up:=PreImage(phom,u);;
gap> List(GeneratorsOfGroup(up),TraceMat);
[ -2, -2, 0, -4, 1, 0 ]
```

For an automorphism *aut*, `Inverse` (31.10.8) returns the inverse automorphism aut^{-1}. However if *hom* is a bijective homomorphism between different groups, or if *hom* is injective and considered to be a bijection to its image, the operation `InverseGeneralMapping` (32.2.3) should be used instead. (See `Inverse` (31.10.8) for a further discussion of this problem.)

Example

```
gap> iso:=IsomorphismPcGroup(g);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]) -> [ f1, f2, f3, f4 ]
gap> Inverse(iso);
```

```
#I  The mapping must be bijective and have source=range
#I  You might want to use 'InverseGeneralMapping'
fail
gap> InverseGeneralMapping(iso);
[ f1, f2, f3, f4 ] -> Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
```

40.3 Efficiency of Homomorphisms

GAP permits to create homomorphisms between arbitrary groups. This section considers the efficiency of the implementation and shows ways how to choose suitable representations. For permutation groups (see 43) or Pc groups (see 46) this is normally nothing to worry about, unless the groups get extremely large. For other groups however certain calculations might be expensive and some precaution might be needed to avoid unnecessarily expensive calculations.

In short, it is always worth to tell a mapping that it is a homomorphism (this can be done by calling `SetIsMapping`) (or to create it directly with `GroupHomomorphismByImagesNC` (40.1.2)).

The basic operations required are to compute image and preimage of elements and to test whether a mapping is a homomorphism. Their cost will differ depending on the type of the mapping.

40.3.1 Mappings given on generators

See `GroupHomomorphismByImages` (40.1.1) and `GroupGeneralMappingByImages` (40.1.3).

Computing images requires to express an element of the source as word in the generators. If it cannot be done effectively (this is determined by `KnowsHowToDecompose` (39.25.7) which returns `true` for example for arbitrary permutation groups, for Pc groups or for finitely presented groups with the images of the free generators) the span of the generators has to be computed elementwise which can be very expensive and memory consuming.

Computing preimages adheres to the same rules with swapped rôles of generators and their images.

The test whether a mapping is a homomorphism requires the computation of a presentation for the source and evaluation of its relators in the images of its generators. For larger groups this can be expensive and `GroupHomomorphismByImagesNC` (40.1.2) should be used if the mapping is known to be a homomorphism.

40.3.2 Action homomorphisms

See `ActionHomomorphism` (41.7.1).

The calculation of images is determined by the acting function used and –for large domains– is often dominated by the search for the position of an image in a list of the domain elements. This can be improved by sorting this list if an efficient method for `\<` (31.11.1) to compare elements of the domain is available.

Once the images of a generating set are computed, computing preimages (which is done via `AsGroupGeneralMappingByImages` (40.1.5)) and computing the kernel behaves the same as for a homomorphism created with `GroupHomomorphismByImages` (40.1.1) from a permutation group.

GAP will always assume that the acting function provided implements a proper group action and thus that the mapping is indeed a homomorphism.

40.3.3 Mappings given by functions

See `GroupHomomorphismByFunction` (40.1.4).

Computing images is wholly determined by the function that performs the image calculation. If no function to compute preimages is given, computing preimages requires mapping every element of the source to find an element that maps to the requested image. This is time and memory consuming.

40.3.4 Other operations

To compute the kernel of a homomorphism (unless the mapping is known to be injective) requires the capability to compute a presentation of the image and to evaluate the relators of this presentation in preimages of the presentations generators.

The calculation of the `Image` (32.4.6) (respectively `ImagesSource` (32.4.1)) value requires to map a generating set of the source, testing surjectivity is a comparison for equality with the range.

Testing injectivity is a test for triviality of the kernel.

The comparison of mappings is based on a lexicographic comparison of a sorted element list of the source. For group homomorphisms, this can be simplified, using `ImagesSmallestGenerators` (40.3.5)

40.3.5 ImagesSmallestGenerators

▷ `ImagesSmallestGenerators(`*map*`)` (attribute)

returns the list of images of `GeneratorsSmallest(Source(`*map*`))`. This list can be used to compare group homomorphisms. (The standard comparison is to compare the image lists on the set of elements of the source. If however x and y have the same images under a and b, certainly all their products have. Therefore it is sufficient to test this on the images of the smallest generators.)

40.4 Homomorphism for very large groups

Some homomorphisms (notably particular actions) transfer known information about the source group (such as a stabilizer chain) to the image group if this is substantially cheaper than to compute the information in the image group anew. In most cases this is no problem and in fact speeds up further calculations notably.

For a huge source group, however this can be time consuming or take a large amount of extra memory for storage. In this case it can be helpful to avoid as much automatism as possible.

The following list of tricks might be useful in such a case. (However you will lose much automatic deduction. So please restrict the use of these to cases where the standard approach does not work.)

- Compute only images (or the `PreImagesRepresentative` (32.5.4)) of group elements. Do not compute the images of (sub)groups or the full preimage of a subgroup.
- Create action homomorphisms as "surjective" (see `ActionHomomorphism` (41.7.1)), otherwise the range is set to be the full symmetric group. However do not compute `Range` (32.3.7) or `Image` (32.4.6) values, but only the images of a generator set.
- If you suspect an action homomorphism to do too much internally, replace the action function with a function that does the same; i.e. replace `OnPoints` (41.2.1) by `function( p, g )`

`return p^g; end;`. The action will be the same, but as the action function is not `OnPoints` (41.2.1), the extra processing for special cases is not triggered.

40.5 Nice Monomorphisms

GAP contains very efficient algorithms for some special representations of groups (for example pc groups or permutation groups) while for other representations only slow generic methods are available. In this case it can be worthwhile to do all calculations rather in an isomorphic image of the group, which is in a "better" representation. The way to achieve this in GAP is via *nice monomorphisms*.

For this mechanism to work, of course there must be effective methods to evaluate the `NiceMonomorphism` (40.5.2) value on elements and to take preimages under it. As by definition no good algorithms exist for the source group, normally this can only be achieved by using the the result of a call to `ActionHomomorphism` (41.7.1) or `GroupHomomorphismByFunction` (40.1.4) (see also section 40.3).

40.5.1 IsHandledByNiceMonomorphism

▷ `IsHandledByNiceMonomorphism(`*obj*`)` (property)

If this property is `true`, high-valued methods that translate all calculations in *obj* in the image under the `NiceMonomorphism` (40.5.2) value of *obj* become available for *obj*.

40.5.2 NiceMonomorphism

▷ `NiceMonomorphism(`*obj*`)` (attribute)

is a homomorphism that is defined (at least) on the whole of *obj* and whose restriction to *obj* is injective. The concrete morphism (and also the image group) will depend on the representation of *obj*.

40.5.3 NiceObject

▷ `NiceObject(`*obj*`)` (attribute)

The `NiceObject` value of *obj* is the image of *obj* under the mapping stored as the value of `NiceMonomorphism` (40.5.2) for *obj*.

A typical example are finite matrix groups, which use a faithful action on vectors to translate all calculations in a permutation group.

Example

```
gap> gl:=GL(3,2);
SL(3,2)
gap> IsHandledByNiceMonomorphism(gl);
true
gap> NiceObject(gl);
Group([ (5,7)(6,8), (2,3,5)(4,7,6) ])
gap> Image(NiceMonomorphism(gl),Z(2)*[[1,0,0],[0,1,1],[1,0,1]]);
(2,6)(3,4,7,8)
```

40.5.4 IsCanonicalNiceMonomorphism

▷ IsCanonicalNiceMonomorphism(*nhom*) (property)

A nice monomorphism (see NiceMonomorphism (40.5.2) *nhom* is canonical if the image set will only depend on the set of group elements but not on the generating set and \< (31.11.1) comparison of group elements translates through the nice monomorphism. This implies that equal objects will always have equal NiceObject (40.5.3) values. In some situations however this condition would be expensive to achieve, therefore it is not guaranteed for every nice monomorphism.

40.6 Group Automorphisms

Group automorphisms are bijective homomorphism from a group onto itself. An important subclass are automorphisms which are induced by conjugation of the group itself or a supergroup.

40.6.1 ConjugatorIsomorphism

▷ ConjugatorIsomorphism(*G*, *g*) (operation)

Let *G* be a group, and *g* an element in the same family as the elements of *G*. ConjugatorIsomorphism returns the isomorphism from *G* to *G*^*g* defined by $h \mapsto h^g$ for all $h \in G$.

If *g* normalizes *G* then ConjugatorIsomorphism does the same as ConjugatorAutomorphismNC (40.6.2).

40.6.2 ConjugatorAutomorphism

▷ ConjugatorAutomorphism(*G*, *g*) (function)
▷ ConjugatorAutomorphismNC(*G*, *g*) (operation)

Let *G* be a group, and *g* an element in the same family as the elements of *G* such that *g* normalizes *G*. ConjugatorAutomorphism returns the automorphism of *G* defined by $h \mapsto h^g$ for all $h \in G$.

If conjugation by *g* does *not* leave *G* invariant, ConjugatorAutomorphism returns fail; in this case, the isomorphism from *G* to *G*^*g* induced by conjugation with *g* can be constructed with ConjugatorIsomorphism (40.6.1).

ConjugatorAutomorphismNC does the same as ConjugatorAutomorphism, except that the check is omitted whether *g* normalizes *G* and it is assumed that *g* is chosen to be in *G* if possible.

40.6.3 InnerAutomorphism

▷ InnerAutomorphism(*G*, *g*) (function)
▷ InnerAutomorphismNC(*G*, *g*) (operation)

Let *G* be a group, and $g \in G$. InnerAutomorphism returns the automorphism of *G* defined by $h \mapsto h^g$ for all $h \in G$.

If *g* is *not* an element of *G*, InnerAutomorphism returns fail; in this case, the isomorphism from *G* to *G*^*g* induced by conjugation with *g* can be constructed with ConjugatorIsomorphism (40.6.1) or with ConjugatorAutomorphism (40.6.2).

InnerAutomorphismNC does the same as InnerAutomorphism, except that the check is omitted whether $g \in G$.

40.6.4 IsConjugatorIsomorphism

▷ IsConjugatorIsomorphism(*hom*) (property)
▷ IsConjugatorAutomorphism(*hom*) (property)
▷ IsInnerAutomorphism(*hom*) (property)

Let *hom* be a group general mapping (see IsGroupGeneralMapping (32.9.4)) with source G, say. IsConjugatorIsomorphism returns true if *hom* is induced by conjugation of G by an element g that lies in G or in a group into which G is naturally embedded in the sense described below, and false otherwise.

Natural embeddings are dealt with in the case that G is a permutation group (see Chapter 43), a matrix group (see Chapter 44), a finitely presented group (see Chapter 47), or a group given w.r.t. a polycyclic presentation (see Chapter 46). In all other cases, IsConjugatorIsomorphism may return false if *hom* is induced by conjugation but is not an inner automorphism.

If IsConjugatorIsomorphism returns true for *hom* then an element g that induces *hom* can be accessed as value of the attribute ConjugatorOfConjugatorIsomorphism (40.6.5).

IsConjugatorAutomorphism returns true if *hom* is an automorphism (see IsEndoGeneralMapping (32.13.3)) that is regarded as a conjugator isomorphism by IsConjugatorIsomorphism, and false otherwise.

IsInnerAutomorphism returns true if *hom* is a conjugator automorphism such that an element g inducing *hom* can be chosen in G, and false otherwise.

40.6.5 ConjugatorOfConjugatorIsomorphism

▷ ConjugatorOfConjugatorIsomorphism(*hom*) (attribute)

For a conjugator isomorphism *hom* (see ConjugatorIsomorphism (40.6.1)), ConjugatorOfConjugatorIsomorphism returns an element g such that mapping under *hom* is induced by conjugation with g.

To avoid problems with IsInnerAutomorphism (40.6.4), it is guaranteed that the conjugator is taken from the source of *hom* if possible.

Example

```
gap> hgens:=[(1,2,3),(1,2,4)];;h:=Group(hgens);;
gap> hom:=GroupHomomorphismByImages(h,h,hgens,[(1,2,3),(2,3,4)]);;
gap> IsInnerAutomorphism(hom);
true
gap> ConjugatorOfConjugatorIsomorphism(hom);
(1,2,3)
gap> hom:=GroupHomomorphismByImages(h,h,hgens,[(1,3,2),(1,4,2)]);
[ (1,2,3), (1,2,4) ] -> [ (1,3,2), (1,4,2) ]
gap> IsInnerAutomorphism(hom);
false
gap> IsConjugatorAutomorphism(hom);
true
gap> ConjugatorOfConjugatorIsomorphism(hom);
(1,2)
```

40.7 Groups of Automorphisms

Group automorphism can be multiplied and inverted and thus it is possible to form groups of automorphisms.

40.7.1 AutomorphismGroup

▷ `AutomorphismGroup(G)` (attribute)

returns the full automorphism group of the group `G`. The automorphisms act on `G` by the caret operator `^`. The automorphism group often stores a `NiceMonomorphism` (40.5.2) value whose image is a permutation group, obtained by the action on a subset of `G`.

Note that current methods for the calculation of the automorphism group of a group G require G to be a permutation group or a pc group to be efficient. For groups in other representations the calculation is likely very slow.

Also, the AutPGrp package installs enhanced methods for `AutomorphismGroup` for finite p-groups, and the FGA package - for finitely generated subgroups of free groups.

Methods may be installed for `AutomorphismGroup` for other domains, such as e.g. for linear codes in the GUAVA package, loops in the loops package and nilpotent Lie algebras in the Sophus package (see package manuals for their descriptions).

40.7.2 IsGroupOfAutomorphisms

▷ `IsGroupOfAutomorphisms(G)` (property)

indicates whether `G` consists of automorphisms of another group H, say. The group H can be obtained from `G` via the attribute `AutomorphismDomain` (40.7.3).

40.7.3 AutomorphismDomain

▷ `AutomorphismDomain(G)` (attribute)

If `G` consists of automorphisms of H, this attribute returns H.

40.7.4 IsAutomorphismGroup

▷ `IsAutomorphismGroup(G)` (property)

indicates whether `G` is the full automorphism group of another group H, this group is given as `AutomorphismDomain` (40.7.3) value of `G`.

Example

```
gap> g:=Group((1,2,3,4),(1,3));
Group([ (1,2,3,4), (1,3) ])
gap> au:=AutomorphismGroup(g);
<group of size 8 with 3 generators>
gap> GeneratorsOfGroup(au);
[ Pcgs([ (2,4), (1,2,3,4), (1,3)(2,4) ]) ->
    [ (1,2)(3,4), (1,2,3,4), (1,3)(2,4) ],
  Pcgs([ (2,4), (1,2,3,4), (1,3)(2,4) ]) ->
```

```
    [ (1,3), (1,2,3,4), (1,3)(2,4) ],
  Pcgs([ (2,4), (1,2,3,4), (1,3)(2,4) ]) ->
    [ (2,4), (1,4,3,2), (1,3)(2,4) ] ]
gap> NiceObject(au);
Group([ (1,2,3,4), (1,3)(2,4), (2,4) ])
```

40.7.5 InnerAutomorphismsAutomorphismGroup

▷ InnerAutomorphismsAutomorphismGroup(*autgroup*) (attribute)

For an automorphism group *autgroup* of a group this attribute stores the subgroup of inner automorphisms (automorphisms induced by conjugation) of the original group.

Example

```
gap> InnerAutomorphismsAutomorphismGroup(au);
<group with 2 generators>
```

40.7.6 InducedAutomorphism

▷ InducedAutomorphism(*epi, aut*) (operation)

Let *aut* be an automorphism of a group G and *epi* be a homomorphism from G to a group H such that the kernel of *epi* is fixed under *aut*. Let U be the image of *epi*. This command returns the automorphism of U induced by *aut* via *epi*, that is, the automorphism of U which maps *g^epi* to *(g^aut)^epi*, for $g \in G$.

Example

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> n:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);
Group([ (1,2)(3,4), (1,3)(2,4) ])
gap> epi:=NaturalHomomorphismByNormalSubgroup(g,n);
[ (1,2,3,4), (1,2) ] -> [ f1*f2, f1 ]
gap> aut:=InnerAutomorphism(g,(1,2,3));
^(1,2,3)
gap> InducedAutomorphism(epi,aut);
^f2
```

40.8 Calculating with Group Automorphisms

Usually the best way to calculate in a group of automorphisms is to translate all calculations to an isomorphic group in a representation, for which better algorithms are available, say a permutation group. This translation can be done automatically using NiceMonomorphism (40.5.2).

Once a group knows to be a group of automorphisms (this can be achieved by testing or setting the property IsGroupOfAutomorphisms (40.7.2)), GAP will try itself to find such a nice monomorphism once calculations in the automorphism group are done.

Note that nice homomorphisms inherit down to subgroups, but cannot necessarily be extended from a subgroup to the whole group. Thus when working with a group of automorphisms, it can be beneficial to enforce calculation of the nice monomorphism for the whole group (for example by explicitly calling Random (30.7.1) and ignoring the result –it will be stored internally) at the start of

the calculation. Otherwise GAP might first calculate a nice monomorphism for the subgroup, only to be forced to calculate a new nice monomorphism for the whole group later on.

If a good domain for a faithful permutation action is known already, a homomorphism for the action on it can be created using NiceMonomorphismAutomGroup (40.8.2). It might be stored by SetNiceMonomorphism (see NiceMonomorphism (40.5.2)).

Another nice way of representing automorphisms as permutations has been described in [Sim97]. It is not yet available in GAP, a description however can be found in section 87.3.

40.8.1 AssignNiceMonomorphismAutomorphismGroup

▷ AssignNiceMonomorphismAutomorphismGroup(*autgrp, group*) (function)

computes a nice monomorphism for *autgroup* acting on *group* and stores it as NiceMonomorphism (40.5.2) value of *autgrp*.

If the centre of AutomorphismDomain (40.7.3) of *autgrp* is trivial, the operation will first try to represent all automorphisms by conjugation (in *group* or in a natural parent of *group*).

If this fails the operation tries to find a small subset of *group* on which the action will be faithful.

The operation sets the attribute NiceMonomorphism (40.5.2) and does not return a value.

40.8.2 NiceMonomorphismAutomGroup

▷ NiceMonomorphismAutomGroup(*autgrp, elms, elmsgens*) (function)

This function creates a monomorphism for an automorphism group *autgrp* of a group by permuting the group elements in the list *elms*. This list must be chosen to yield a faithful representation. *elmsgens* is a list of generators which are a subset of *elms*. (They can differ from the group's original generators.) It does not yet assign it as NiceMonomorphism (40.5.2) value.

40.9 Searching for Homomorphisms

40.9.1 IsomorphismGroups

▷ IsomorphismGroups(*G, H*) (function)

computes an isomorphism between the groups *G* and *H* if they are isomorphic and returns fail otherwise.

With the existing methods the amount of time needed grows with the size of a generating system of *G*. (Thus in particular for *p*-groups calculations can be slow.) If you do only need to know whether groups are isomorphic, you might want to consider IdSmallGroup (50.7.5) or the random isomorphism test (see RandomIsomorphismTest (46.10.1)).

Example

```
gap> g:=Group((1,2,3,4),(1,3));;
gap> h:=Group((1,4,6,7)(2,3,5,8), (1,5)(2,6)(3,4)(7,8));;
gap> IsomorphismGroups(g,h);
[ (1,2,3,4), (1,3) ] -> [ (1,4,6,7)(2,3,5,8), (1,2)(3,7)(4,8)(5,6) ]
gap> IsomorphismGroups(g,Group((1,2,3,4),(1,2)));
fail
```

40.9.2 AllHomomorphismClasses

▷ `AllHomomorphismClasses(G, H)` (function)

For two groups G and H, this function returns representatives of all homomorphisms $GtoH$ up to H-conjugacy.

Example

```
gap> AllHomomorphismClasses(SymmetricGroup(4),SymmetricGroup(3));
[ [ (1,3,4,2), (1,2,4) ] -> [ (), () ],
  [ (1,3,4,2), (1,2,4) ] -> [ (1,2), () ],
  [ (1,3,4,2), (1,2,4) ] -> [ (2,3), (1,2,3) ] ]
```

40.9.3 AllHomomorphisms

▷ `AllHomomorphisms(G, H)` (function)
▷ `AllEndomorphisms(G)` (function)
▷ `AllAutomorphisms(G)` (function)

For two groups G and H, this function returns all homomorphisms $GtoH$. Since this number will grow quickly, `AllHomomorphismClasses` (40.9.2) should be used in most cases. `AllEndomorphisms` returns all homomorphisms from G to itself, `AllAutomorphisms` returns all bijective endomorphisms.

Example

```
gap> AllHomomorphisms(SymmetricGroup(3),SymmetricGroup(3));
[ [ (1,2,3), (1,2) ] -> [ (), () ],
  [ (1,2,3), (1,2) ] -> [ (), (1,2) ],
  [ (1,2,3), (1,2) ] -> [ (), (2,3) ],
  [ (1,2,3), (1,2) ] -> [ (), (1,3) ],
  [ (1,2,3), (1,2) ] -> [ (1,2,3), (1,2) ],
  [ (1,2,3), (1,2) ] -> [ (1,2,3), (2,3) ],
  [ (1,2,3), (1,2) ] -> [ (1,3,2), (1,2) ],
  [ (1,2,3), (1,2) ] -> [ (1,2,3), (1,3) ],
  [ (1,2,3), (1,2) ] -> [ (1,3,2), (1,3) ],
  [ (1,2,3), (1,2) ] -> [ (1,3,2), (2,3) ] ]
```

40.9.4 GQuotients

▷ `GQuotients(F, G)` (operation)

computes all epimorphisms from F onto G up to automorphisms of G. This classifies all factor groups of F which are isomorphic to G.

With the existing methods the amount of time needed grows with the size of a generating system of G. (Thus in particular for p-groups calculations can be slow.)

If the `findall` option is set to `false`, the algorithm will stop once one homomorphism has been found (this can be faster and might be sufficient if not all homomorphisms are needed).

Example

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((1,2,3),(1,2));
```

```
Group([ (1,2,3), (1,2) ])
gap> quo:=GQuotients(g,h);
[ [ (1,2,4,3), (1,2,3) ] -> [ (2,3), (1,2,3) ] ]
```

40.9.5 IsomorphicSubgroups

▷ `IsomorphicSubgroups(`*G, H*`)` (operation)

computes all monomorphisms from *H* into *G* up to *G*-conjugacy of the image groups. This classifies all *G*-classes of subgroups of *G* which are isomorphic to *H*.

With the existing methods, the amount of time needed grows with the size of a generating system of *G*. (Thus in particular for *p*-groups calculations can be slow.) A main use of `IsomorphicSubgroups` therefore is to find nonsolvable subgroups (which often can be generated by 2 elements).

(To find *p*-subgroups it is often faster to compute the subgroup lattice of the Sylow subgroup and to use `IdGroup` (50.7.5) to identify the type of the subgroups.)

If the `findall` option is set to `false`, the algorithm will stop once one homomorphism has been found (this can be faster and might be sufficient if not all homomorphisms are needed).

Example

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((3,4),(1,2));;
gap> emb:=IsomorphicSubgroups(g,h);
[ [ (3,4), (1,2) ] -> [ (1,2), (3,4) ],
  [ (3,4), (1,2) ] -> [ (1,3)(2,4), (1,2)(3,4) ] ]
```

40.9.6 MorClassLoop

▷ `MorClassLoop(`*range, classes, params, action*`)` (function)

This function loops over element tuples taken from *classes* and checks these for properties such as generating a given group, or fulfilling relations. This can be used to find small generating sets or all types of Morphisms. The element tuples are used only up to up to inner automorphisms as all images can be obtained easily from them by conjugation while running through all of them usually would take too long.

range is a group from which these elements are taken. The classes are given in a list *classes* which is a list of records with the following components.

`classes`
: list of conjugacy classes

`representative`
: One element in the union of these classes

`size`
: The sum of the sizes of these classes

params is a record containing the following optional components.

`gens`
: generators that are to be mapped (for testing morphisms). The length of this list determines the length of element tuples considered.

`from`
: a preimage group (that contains `gens`)

`to` image group (which might be smaller than `range`)

`free`
: free generators, a list of the same length than the generators `gens`.

`rels`
: some relations that hold among the generators `gens`. They are given as a list `[ word, order ]` where `word` is a word in the free generators `free`.

`dom` a set of elements on which automorphisms act faithfully (used to do element tests in partial automorphism groups).

`aut` Subgroup of already known automorphisms.

`condition`
: A function that will be applied to the homomorphism and must return `true` for the homomorphism to be accepted.

`action` is a number whose bit-representation indicates the requirements which are enforced on the element tuples found, as follows.

1 homomorphism

2 injective

4 surjective

8 find all (otherwise stops after the first find)

If the search is for homomorphisms, the function returns homomorphisms obtained by mapping the given generators `gens` instead of element tuples.

The "Morpheus" algorithm used to find homomorphisms is described in [Hul96, Section V.5].

40.10 Representations for Group Homomorphisms

The different representations of group homomorphisms are used to indicate from what type of group to what type of group they map and thus determine which methods are used to compute images and preimages.

The information in this section is mainly relevant for implementing new methods and not for using homomorphisms.

40.10.1 IsGroupGeneralMappingByImages

▷ IsGroupGeneralMappingByImages(*map*) (Representation)

Representation for mappings from one group to another that are defined by extending a mapping of group generators homomorphically. Instead of record components, the attribute MappingGeneratorsImages (40.10.2) is used to store generators and their images.

40.10.2 MappingGeneratorsImages

▷ MappingGeneratorsImages(*map*) (attribute)

This attribute contains a list of length 2, the first entry being a list of generators of the source of *map* and the second entry a list of their images. This attribute is used, for example, by GroupHomomorphismByImages (40.1.1) to store generators and images.

40.10.3 IsGroupGeneralMappingByAsGroupGeneralMappingByImages

▷ IsGroupGeneralMappingByAsGroupGeneralMappingByImages(*map*) (Representation)

Representation for mappings that delegate work on a GroupHomomorphismByImages (40.1.1).

40.10.4 IsPreimagesByAsGroupGeneralMappingByImages

▷ IsPreimagesByAsGroupGeneralMappingByImages(*map*) (Representation)

Representation for mappings that delegate work for preimages to a mapping created with GroupHomomorphismByImages (40.1.1).

40.10.5 IsPermGroupGeneralMapping

▷ IsPermGroupGeneralMapping(*map*) (Representation)
▷ IsPermGroupGeneralMappingByImages(*map*) (Representation)
▷ IsPermGroupHomomorphism(*map*) (Representation)
▷ IsPermGroupHomomorphismByImages(*map*) (Representation)

are the representations for mappings that map from a perm group

40.10.6 IsToPermGroupGeneralMappingByImages

▷ IsToPermGroupGeneralMappingByImages(*map*) (Representation)
▷ IsToPermGroupHomomorphismByImages(*map*) (Representation)

is the representation for mappings that map to a perm group

40.10.7 IsGroupGeneralMappingByPcgs

▷ IsGroupGeneralMappingByPcgs(*map*) (Representation)

is the representations for mappings that map a pcgs to images and thus may use exponents to decompose generators.

40.10.8 IsPcGroupGeneralMappingByImages

▷ IsPcGroupGeneralMappingByImages(*map*) (Representation)
▷ IsPcGroupHomomorphismByImages(*map*) (Representation)

is the representation for mappings from a pc group

40.10.9 IsToPcGroupGeneralMappingByImages

▷ IsToPcGroupGeneralMappingByImages(*map*) (Representation)
▷ IsToPcGroupHomomorphismByImages(*map*) (Representation)

is the representation for mappings to a pc group

40.10.10 IsFromFpGroupGeneralMappingByImages

▷ IsFromFpGroupGeneralMappingByImages(*map*) (Representation)
▷ IsFromFpGroupHomomorphismByImages(*map*) (Representation)

is the representation of mappings from an fp group.

40.10.11 IsFromFpGroupStdGensGeneralMappingByImages

▷ IsFromFpGroupStdGensGeneralMappingByImages(*map*) (Representation)
▷ IsFromFpGroupStdGensHomomorphismByImages(*map*) (Representation)

is the representation of total mappings from an fp group that give images of the standard generators.

Chapter 41

Group Actions

A *group action* is a triple (G, Ω, μ), where G is a group, Ω a set and $\mu: \Omega \times G \to \Omega$ a function that is compatible with the group arithmetic. We call Ω the *domain* of the action.

In GAP, Ω can be a duplicate-free collection (an object that permits access to its elements via the $\Omega[n]$ operation, for example a list), it does not need to be sorted (see `IsSet` (21.17.4)).

The acting function μ is a binary GAP function that returns the image $\mu(x, g)$ for a point $x \in \Omega$ and a group element $g \in G$.

In GAP, groups always act from the right, that is $\mu(\mu(x, g), h) = \mu(x, gh)$.

GAP does not test whether the acting function μ satisfies the conditions for a group operation but silently assumes that is does. (If it does not, results are unpredictable.)

The first section of this chapter, 41.1, describes the various ways how operations for group actions can be called.

Functions for several commonly used action are already built into GAP. These are listed in section 41.2.

The sections 41.7 and 41.8 describe homomorphisms and mappings associated to group actions as well as the permutation group image of an action.

The other sections then describe operations to compute orbits, stabilizers, as well as properties of actions.

Finally section 41.12 describes the concept of "external sets" which represent the concept of a *G-set* and underly the actions mechanism.

41.1 About Group Actions

The syntax which is used by the operations for group actions is quite flexible. For example we can call the operation `OrbitsDomain` (41.4.3) for the orbits of the group `G` on the domain `Omega` in the following ways:

`OrbitsDomain`$(G, \Omega[, \mu])$

The acting function μ is optional. If it is not given, the built-in action `OnPoints` (41.2.1) (which defines an action via the caret operator `^`) is used as a default.

`OrbitsDomain`$(G, \Omega, gens, acts[, \mu])$

This second version of `OrbitsDomain` (41.4.3) permits one to implement an action induced by a homomorphism: If the group H acts on Ω via μ and $\varphi: G \to H$ is a homomorphism, G acts on Ω via the induced action $\mu'(x, g) = \mu(x, g^{\varphi})$.

Here *gens* must be a set of generators of G and *acts* the images of *gens* under φ. μ is the acting function for H. Again, the function μ is optional and `OnPoints` (41.2.1) is used as a default.

The advantage of this notation is that GAP does not need to construct this homomorphism φ and the range group `H` as GAP objects. (If a small group G acts via complicated objects *acts* this otherwise could lead to performance problems.)

GAP does not test whether the mapping *gens* $\mapsto$ *acts* actually induces a homomorphism and the results are unpredictable if this is not the case.

`OrbitsDomain`(*extset*)

A third variant is to call the operation with an external set, which then provides G, Ω and μ. You will find more about external sets in Section 41.12.

For operations like `Stabilizer` (41.5.2) of course the domain must be replaced by an element of the domain of the action.

41.2 Basic Actions

GAP already provides acting functions for the more common actions of a group. For built-in operations such as `Stabilizer` (41.5.2) special methods are available for many of these actions.

If one needs an action for which no acting function is provided by the library it can be implemented via a GAP function that conforms to the syntax

```
actfun( omega, g )
```

where `omega` is an element of the action domain, `g` is an element of the acting group, and the return value is the image of `omega` under `g`.

For example one could define the following function that acts on pairs of polynomials via `OnIndeterminates` (41.2.13):

Example

```
OnIndeterminatesPairs:= function( polypair, g )
  return [ OnIndeterminates( polypair[1], g ),
           OnIndeterminates( polypair[2], g ) ];
end;
```

Note that this function *must* implement a group action from the *right*. This is not verified by GAP and results are unpredictable otherwise.

41.2.1 OnPoints

▷ `OnPoints(`*pnt, g*`)` (function)

returns *pnt* `^` *g*. This is for example the action of a permutation group on points, or the action of a group on its elements via conjugation. The action of a matrix group on vectors from the right is described by both `OnPoints` and `OnRight` (41.2.2).

41.2.2 OnRight

▷ `OnRight(`*pnt, g*`)` (function)

returns *pnt * g*. This is for example the action of a group on its elements via right multiplication, or the action of a group on the cosets of a subgroup. The action of a matrix group on vectors from the right is described by both OnPoints (41.2.1) and OnRight.

41.2.3 OnLeftInverse

▷ OnLeftInverse(*pnt, g*) (function)

returns g^{-1} * *pnt*. Forming the inverse is necessary to make this a proper action, as in GAP groups always act from the right.

OnLeftInverse is used for example in the representation of a right coset as an external set (see 41.12), that is, a right coset Ug is an external set for the group U acting on it via OnLeftInverse.)

41.2.4 OnSets

▷ OnSets(*set, g*) (function)

Let *set* be a proper set (see 21.19). OnSets returns the proper set formed by the images of all points x of *set* via the action function OnPoints (41.2.1), applied to x and g.

OnSets is for example used to compute the action of a permutation group on blocks.

(OnTuples (41.2.5) is an action on lists that preserves the ordering of entries.)

41.2.5 OnTuples

▷ OnTuples(*tup, g*) (function)

Let *tup* be a list. OnTuples returns the list formed by the images of all points x of *tup* via the action function OnPoints (41.2.1), applied to x and g.

(OnSets (41.2.4) is an action on lists that additionally sorts the entries of the result.)

41.2.6 OnPairs

▷ OnPairs(*tup, g*) (function)

is a special case of OnTuples (41.2.5) for lists *tup* of length 2.

41.2.7 OnSetsSets

▷ OnSetsSets(*set, g*) (function)

implements the action on sets of sets. For the special case that the sets are pairwise disjoint, it is possible to use OnSetsDisjointSets (41.2.8). *set* must be a sorted list whose entries are again sorted lists, otherwise an error is triggered (see 41.3).

41.2.8 OnSetsDisjointSets

▷ `OnSetsDisjointSets(`*set*`, `*g*`)` (function)

implements the action on sets of pairwise disjoint sets (see also `OnSetsSets` (41.2.7)). *set* must be a sorted list whose entries are again sorted lists, otherwise an error is triggered (see 41.3).

41.2.9 OnSetsTuples

▷ `OnSetsTuples(`*set*`, `*g*`)` (function)

implements the action on sets of tuples. *set* must be a sorted list, otherwise an error is triggered (see 41.3).

41.2.10 OnTuplesSets

▷ `OnTuplesSets(`*set*`, `*g*`)` (function)

implements the action on tuples of sets. *set* must be a list whose entries are again sorted lists, otherwise an error is triggered (see 41.3).

41.2.11 OnTuplesTuples

▷ `OnTuplesTuples(`*set*`, `*g*`)` (function)

implements the action on tuples of tuples.

Example

```
gap> g:=Group((1,2,3),(2,3,4));;
gap> Orbit(g,1,OnPoints);
[ 1, 2, 3, 4 ]
gap> Orbit(g,(),OnRight);
[ (), (1,2,3), (2,3,4), (1,3,2), (1,3)(2,4), (1,2)(3,4), (2,4,3),
  (1,4,2), (1,4,3), (1,3,4), (1,2,4), (1,4)(2,3) ]
gap> Orbit(g,[1,2],OnPairs);
[ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ], [ 3, 1 ], [ 3, 4 ], [ 2, 1 ],
  [ 1, 4 ], [ 4, 1 ], [ 4, 2 ], [ 3, 2 ], [ 2, 4 ], [ 4, 3 ] ]
gap> Orbit(g,[1,2],OnSets);
[ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ], [ 3, 4 ], [ 1, 4 ], [ 2, 4 ] ]
gap> Orbit(g,[[1,2],[3,4]],OnSetsSets);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 4 ], [ 2, 3 ] ],
  [ [ 1, 3 ], [ 2, 4 ] ] ]
gap> Orbit(g,[[1,2],[3,4]],OnTuplesSets);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 2, 3 ], [ 1, 4 ] ],
  [ [ 1, 3 ], [ 2, 4 ] ], [ [ 3, 4 ], [ 1, 2 ] ],
  [ [ 1, 4 ], [ 2, 3 ] ], [ [ 2, 4 ], [ 1, 3 ] ] ]
gap> Orbit(g,[[1,2],[3,4]],OnSetsTuples);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 4 ], [ 2, 3 ] ],
  [ [ 1, 3 ], [ 4, 2 ] ], [ [ 2, 4 ], [ 3, 1 ] ],
  [ [ 2, 1 ], [ 4, 3 ] ], [ [ 3, 2 ], [ 4, 1 ] ] ]
gap> Orbit(g,[[1,2],[3,4]],OnTuplesTuples);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 2, 3 ], [ 1, 4 ] ],
```

```
    [ [ 1, 3 ], [ 4, 2 ] ], [ [ 3, 1 ], [ 2, 4 ] ],
    [ [ 3, 4 ], [ 1, 2 ] ], [ [ 2, 1 ], [ 4, 3 ] ],
    [ [ 1, 4 ], [ 2, 3 ] ], [ [ 4, 1 ], [ 3, 2 ] ],
    [ [ 4, 2 ], [ 1, 3 ] ], [ [ 3, 2 ], [ 4, 1 ] ],
    [ [ 2, 4 ], [ 3, 1 ] ], [ [ 4, 3 ], [ 2, 1 ] ] ]
```

41.2.12 OnLines

▷ `OnLines(`*vec*`, `*g*`)` (function)

Let *vec* be a *normed* row vector, that is, its first nonzero entry is normed to the identity of the relevant field, see `NormedRowVector` (23.2.1). The function `OnLines` returns the row vector obtained from first multiplying *vec* from the right with *g* (via `OnRight` (41.2.2)) and then normalizing the resulting row vector by scalar multiplication from the left.

This action corresponds to the projective action of a matrix group on one-dimensional subspaces.

If *vec* is a zero vector or is not normed then an error is triggered (see 41.3).

Example

```
gap> gl:=GL(2,5);;v:=[1,0]*Z(5)^0;
[ Z(5)^0, 0*Z(5) ]
gap> h:=Action(gl,Orbit(gl,v,OnLines),OnLines);
Group([ (2,3,5,6), (1,2,4)(3,6,5) ])
```

41.2.13 OnIndeterminates (as a permutation action)

▷ `OnIndeterminates(`*poly*`, `*perm*`)` (function)

A permutation *perm* acts on the multivariate polynomial *poly* by permuting the indeterminates as it permutes points.

Example

```
gap> x:=Indeterminate(Rationals,1);; y:=Indeterminate(Rationals,2);;
gap> OnIndeterminates(x^7*y+x*y^4,(1,17)(2,28));
x_17^7*x_28+x_17*x_28^4
gap> Stabilizer(Group((1,2,3,4),(1,2)),x*y,OnIndeterminates);
Group([ (1,2), (3,4) ])
```

41.2.14 Permuted (as a permutation action)

▷ `Permuted(`*list*`, `*perm*`)` (function)

The following example demonstrates `Permuted` (21.20.18) being used to implement a permutation action on a domain:

Example

```
gap> g:=Group((1,2,3),(1,2));;
gap> dom:=[ "a", "b", "c" ];;
gap> Orbit(g,dom,Permuted);
[ [ "a", "b", "c" ], [ "c", "a", "b" ], [ "b", "a", "c" ],
  [ "b", "c", "a" ], [ "a", "c", "b" ], [ "c", "b", "a" ] ]
```

41.2.15 OnSubspacesByCanonicalBasis

▷ `OnSubspacesByCanonicalBasis(bas, mat)` (function)
▷ `OnSubspacesByCanonicalBasisConcatenations(basvec, mat)` (function)

implements the operation of a matrix group on subspaces of a vector space. `bas` must be a list of (linearly independent) vectors which forms a basis of the subspace in Hermite normal form. `mat` is an element of the acting matrix group. The function returns a mutable matrix which gives the basis of the image of the subspace in Hermite normal form. (In other words: it triangulizes the product of `bas` with `mat`.)

`bas` must be given in Hermite normal form, otherwise an error is triggered (see 41.3).

41.3 Action on canonical representatives

A variety of action functions assumes that the objects on which it acts are given in a particular form, for example canonical representatives. Affected actions are for example `OnSetsSets` (41.2.7), `OnSetsDisjointSets` (41.2.8), `OnSetsTuples` (41.2.9), `OnTuplesSets` (41.2.10), `OnLines` (41.2.12) and `OnSubspacesByCanonicalBasis` (41.2.15).

If orbit seeds or domain elements are not given in the required form GAP will issue an error message:

Example

```
gap> Orbit(SymmetricGroup(5),[[2,4],[1,3]],OnSetsSets);
Error, Action not well-defined. See the manual section
‘‘Action on canonical representatives’’.
```

In this case the affected domain elements have to be brought in canonical form, as documented for the respective action function. For interactive use this is most easily done by acting with the identity element of the group.

(A similar error could arise if a user-defined action function is used which actually does not implement an action from the right.)

41.4 Orbits

If a group G acts on a set Ω, the set of all images of $x \in \Omega$ under elements of G is called the *orbit* of x. The set of orbits of G is a partition of Ω.

41.4.1 Orbit

▷ `Orbit(G[, Omega], pnt[, gens, acts][, act])` (operation)

The orbit of the point `pnt` is the list of all images of `pnt` under the action of the group `G` w.r.t. the action function `act` or `OnPoints` (41.2.1) if no action function is given.

(Note that the arrangement of points in this list is not defined by the operation.)

The orbit of `pnt` will always contain one element that is *equal* to `pnt`, however for performance reasons this element is not necessarily *identical* to `pnt`, in particular if `pnt` is mutable.

Example

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> Orbit(g,1);
[ 1, 3, 2, 4 ]
gap> Orbit(g,[1,2],OnSets);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ], [ 2, 4 ] ]
```

(See Section 41.2 for information about specific actions.)

41.4.2 Orbits (operation)

▷ Orbits(*G, seeds[, gens, acts][, act]*) (operation)
▷ Orbits(*xset*) (attribute)

returns a duplicate-free list of the orbits of the elements in *seeds* under the action *act* of *G* or under OnPoints (41.2.1) if no action function is given.

(Note that the arrangement of orbits or of points within one orbit is not defined by the operation.)

41.4.3 OrbitsDomain

▷ OrbitsDomain(*G, Omega[, gens, acts][, act]*) (operation)
▷ OrbitsDomain(*xset*) (attribute)

returns a list of the orbits of *G* on the domain *Omega* (given as lists) under the action *act* or under OnPoints (41.2.1) if no action function is given.

This operation is often faster than Orbits (41.4.2). The domain *Omega* must be closed under the action of *G*, otherwise an error can occur.

(Note that the arrangement of orbits or of points within one orbit is not defined by the operation.)

Example

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> Orbits(g,[1..5]);
[ [ 1, 3, 2, 4 ], [ 5 ] ]
gap> OrbitsDomain(g,Arrangements([1..4],3),OnTuples);
[ [ [ 1, 2, 3 ], [ 3, 1, 2 ], [ 1, 4, 2 ], [ 2, 3, 1 ], [ 2, 1, 4 ],
      [ 3, 4, 1 ], [ 1, 3, 4 ], [ 4, 2, 1 ], [ 4, 1, 3 ],
      [ 2, 4, 3 ], [ 3, 2, 4 ], [ 4, 3, 2 ] ],
  [ [ 1, 2, 4 ], [ 3, 1, 4 ], [ 1, 4, 3 ], [ 2, 3, 4 ], [ 2, 1, 3 ],
      [ 3, 4, 2 ], [ 1, 3, 2 ], [ 4, 2, 3 ], [ 4, 1, 2 ],
      [ 2, 4, 1 ], [ 3, 2, 1 ], [ 4, 3, 1 ] ] ]
gap> OrbitsDomain(g,GF(2)^2,[(1,2,3),(1,4)(2,3)],
> [[[Z(2)^0,Z(2)^0],[Z(2)^0,0*Z(2)]],[[Z(2)^0,0*Z(2)],[0*Z(2),Z(2)^0]]]);
[ [ <an immutable GF2 vector of length 2> ],
  [ <an immutable GF2 vector of length 2>,
      <an immutable GF2 vector of length 2>,
      <an immutable GF2 vector of length 2> ] ]
```

(See Section 41.2 for information about specific actions.)

41.4.4 OrbitLength

▷ `OrbitLength(G, Omega, pnt[, gens, acts][, act])` (operation)

computes the length of the orbit of *pnt* under the action function `act` or `OnPoints` (41.2.1) if no action function is given.

41.4.5 OrbitLengths

▷ `OrbitLengths(G, seeds[, gens, acts][, act])` (operation)
▷ `OrbitLengths(xset)` (attribute)

computes the lengths of all the orbits of the elements in *seeds* under the action `act` of *G*.

41.4.6 OrbitLengthsDomain

▷ `OrbitLengthsDomain(G, Omega[, gens, acts][, act])` (operation)
▷ `OrbitLengthsDomain(xset)` (attribute)

computes the lengths of all the orbits of *G* on *Omega*.

This operation is often faster than `OrbitLengths` (41.4.5). The domain *Omega* must be closed under the action of *G*, otherwise an error can occur.

Example

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> OrbitLength(g,[1,2,3,4],OnTuples);
12
gap> OrbitLengths(g,Arrangements([1..4],4),OnTuples);
[ 12, 12 ]
```

41.5 Stabilizers

The *stabilizer* of a point x under the action of a group G is the set of all those elements in G which fix x.

41.5.1 OrbitStabilizer

▷ `OrbitStabilizer(G[, Omega], pnt[, gens, acts], act)` (operation)

computes the orbit and the stabilizer of *pnt* simultaneously in a single orbit-stabilizer algorithm. The stabilizer will have *G* as its parent.

41.5.2 Stabilizer

▷ `Stabilizer(G[, Omega], pnt[, gens, acts][, act])` (function)

computes the stabilizer in *G* of the point *pnt*, that is the subgroup of those elements of *G* that fix *pnt*. The stabilizer will have *G* as its parent.

Example

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> Stabilizer(g,4);
Group([ (1,3,2) ])
```

The stabilizer of a set or tuple of points can be computed by specifying an action of sets or tuples of points.

Example

```
gap> Stabilizer(g,[1,2],OnSets);
Group([ (1,2)(3,4) ])
gap> Stabilizer(g,[1,2],OnTuples);
Group(())
gap> OrbitStabilizer(g,[1,2],OnSets);
rec(
  orbit := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ],
      [ 2, 4 ] ], stabilizer := Group([ (1,2)(3,4) ]) )
```

(See Section 41.2 for information about specific actions.)

The standard methods for all these actions are an orbit-stabilizer algorithm. For permutation groups backtrack algorithms are used. For solvable groups an orbit-stabilizer algorithm for solvable groups, which uses the fact that the orbits of a normal subgroup form a block system (see [LNS84]) is used.

41.5.3 OrbitStabilizerAlgorithm

▷ `OrbitStabilizerAlgorithm(`*G, Omega, blist, gens, acts, pntact*`)` (function)

This operation should not be called by a user. It is documented however for purposes to extend or maintain the group actions package (the word "package" here refers to the GAP functionality for group actions, not to a GAP package).

`OrbitStabilizerAlgorithm` performs an orbit stabilizer algorithm for the group *G* acting with the generators *gens* via the generator images *gens* and the group action *act* on the element *pnt*. (For technical reasons *pnt* and *act* are put in one record with components `pnt` and `act` respectively.)

The *pntact* record may carry a component *stabsub*. If given, this must be a subgroup stabilizing *all* points in the domain and can be used to abbreviate stabilizer calculations.

The *pntact* component also may contain the boolean entry `onlystab` set to `true`. In this case the `orbit` component may be omitted from the result.

The argument *Omega* (which may be replaced by `false` to be ignored) is the set within which the orbit is computed (once the orbit is the full domain, the orbit calculation may stop). If *blist* is given it must be a bit list corresponding to *Omega* in which elements which have been found already will be "ticked off" with `true`. (In particular, the entries for the orbit of *pnt* still must be all set to `false`). Again the remaining action domain (the bits set initially to `false`) can be used to stop if the orbit cannot grow any longer. Another use of the bit list is if *Omega* is an enumerator which can determine `PositionCanonical` (21.16.3) values very quickly. In this situation it can be worth to search images not in the orbit found so far, but via their position in *Omega* and use a the bit list to keep track whether the element is in the orbit found so far.

41.6 Elements with Prescribed Images

41.6.1 RepresentativeAction

▷ RepresentativeAction(*G[, Omega], d, e[, gens, acts][, act]*) (operation)

computes an element of *G* that maps *d* to *e* under the given action and returns fail if no such element exists.

Example

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> RepresentativeAction(g,1,3);
(1,3)(2,4)
gap> RepresentativeAction(g,1,3,OnPoints);
(1,3)(2,4)
gap> RepresentativeAction(g,(1,2,3),(2,4,3));
(1,2,4)
gap> RepresentativeAction(g,(1,2,3),(2,3,4));
fail
gap> RepresentativeAction(g,Group((1,2,3)),Group((2,3,4)));
(1,2,4)
gap>  RepresentativeAction(g,[1,2,3],[1,2,4],OnSets);
(2,4,3)
gap>  RepresentativeAction(g,[1,2,3],[1,2,4],OnTuples);
fail
```

(See Section 41.2 for information about specific actions.)

Again the standard method for RepresentativeAction is an orbit-stabilizer algorithm, for permutation groups and standard actions a backtrack algorithm is used.

41.7 The Permutation Image of an Action

When a group G acts on a domain Ω, an enumeration of *Omega* yields a homomorphism from G into the symmetric group on $\{1,\ldots,|\Omega|\}$. In GAP, the enumeration of Ω is provided by the Enumerator (30.3.2) value of Ω which of course is Ω itself if it is a list.

For an action homomorphism, the operation UnderlyingExternalSet (41.12.16) will return the external set on Ω which affords the action.

41.7.1 ActionHomomorphism

▷ ActionHomomorphism(*G, Omega[, gens, acts][, act][, "surjective"]*) (operation)
▷ ActionHomomorphism(*xset[, "surjective"]*) (attribute)
▷ ActionHomomorphism(*action*) (attribute)

computes a homomorphism from *G* into the symmetric group on |*Omega*| points that gives the permutation action of *G* on *Omega*.

By default the homomorphism returned by ActionHomomorphism is not necessarily surjective (its Range (32.3.7) value is the full symmetric group) to avoid unnecessary computation of the image. If the optional string argument "surjective" is given, a surjective homomorphism is created.

The third version (which is supported only for GAP3 compatibility) returns the action homomorphism that belongs to the image obtained via Action (41.7.2).

(See Section 41.2 for information about specific actions.)

Example

```
gap> g:=Group((1,2,3),(1,2));;
gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples);
<action homomorphism>
gap> Image(hom);
Group(
[ (1,9,13)(2,10,14)(3,7,15)(4,8,16)(5,12,17)(6,11,18)(19,22,23)(20,21,
    24), (1,7)(2,8)(3,9)(4,10)(5,11)(6,12)(13,15)(14,16)(17,18)(19,
    21)(20,22)(23,24) ])
gap> Size(Range(hom));Size(Image(hom));
62044840173323943936000 0
6
gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples,
> "surjective");;
gap> Size(Range(hom));
6
```

When acting on a domain, the operation PositionCanonical (21.16.3) is used to determine the position of elements in the domain. This can be used to act on a domain given by a list of representatives for which PositionCanonical (21.16.3) is implemented, for example the return value of RightTransversal (39.8.1).

41.7.2 Action (for a group, an action domain, etc.)

▷ Action(*G, Omega[, gens, acts][, act]*) (operation)

▷ Action(*xset*) (attribute)

returns the image group of ActionHomomorphism (41.7.1) called with the same parameters.

Note that (for compatibility reasons to be able to get the action homomorphism) this image group internally stores the action homomorphism. If *G* or *Omega* are extremely big, this can cause memory problems. In this case compute only generator images and form the image group yourself.

(See Section 41.2 for information about specific actions.)

The following code shows for example how to create the regular action of a group.

Example

```
gap> g:=Group((1,2,3),(1,2));;
gap> Action(g,AsList(g),OnRight);
Group([ (1,4,5)(2,3,6), (1,3)(2,4)(5,6) ])
```

41.7.3 SparseActionHomomorphism

▷ SparseActionHomomorphism(*G, start[, gens, acts][, act]*) (operation)

▷ SortedSparseActionHomomorphism(*G, start[, gens, acts][, act]*) (operation)

SparseActionHomomorphism computes the action homomorphism (see ActionHomomorphism (41.7.1)) with arguments *G*, *D*, and the optional arguments given, where *D* is the union of the *G*-orbits of all points in *start*. In the Orbit (41.4.1) calls that are used to create *D*, again the optional arguments given are entered.)

If *G* acts on a very large domain not surjectively this may yield a permutation image of substantially smaller degree than by action on the whole domain.

The operation SparseActionHomomorphism will only use \= (31.11.1) comparisons of points in the orbit. Therefore it can be used even if no good \< (31.11.1) comparison method for these points is available. However the image group will depend on the generators *gens* of *G*.

The operation SortedSparseActionHomomorphism in contrast will sort the orbit and thus produce an image group which does not depend on these generators.

Example

```
gap> h:=Group(Z(3)*[[[1,1],[0,1]]]);
Group([ [ [ Z(3), Z(3) ], [ 0*Z(3), Z(3) ] ] ])
gap> hom:=ActionHomomorphism(h,GF(3)^2,OnRight);;
gap> Image(hom);
Group([ (2,3)(4,9,6,7,5,8) ])
gap> hom:=SparseActionHomomorphism(h,[Z(3)*[1,0]],OnRight);;
gap> Image(hom);
Group([ (1,2,3,4,5,6) ])
```

41.8 Action of a group on itself

Of particular importance is the action of a group on its elements or cosets of a subgroup. These actions can be obtained by using ActionHomomorphism (41.7.1) for a suitable domain (for example a list of subgroups). For the following (frequently used) types of actions however special (often particularly efficient) functions are provided. A special case is the regular action on all elements.

41.8.1 FactorCosetAction

▷ FactorCosetAction(*G, U[, N]*) (operation)

This command computes the action of the group *G* on the right cosets of the subgroup *U*. If a normal subgroup *N* of *G* is given, it is stored as kernel of this action.

Example

```
gap> g:=Group((1,2,3,4,5),(1,2));;u:=SylowSubgroup(g,2);;Index(g,u);
15
gap> FactorCosetAction(g,u);
<action epimorphism>
gap> Range(last);
Group([ (1,9,13,10,4)(2,8,14,11,5)(3,7,15,12,6),
  (1,7)(2,8)(3,9)(5,6)(10,11)(14,15) ])
```

41.8.2 RegularActionHomomorphism

▷ RegularActionHomomorphism(*G*) (attribute)

returns an isomorphism from *G* onto the regular permutation representation of *G*.

41.8.3 AbelianSubfactorAction

▷ AbelianSubfactorAction(*G, M, N*) (operation)

Let *G* be a group and $M \geq N$ be subgroups of a common parent that are normal under *G*, such that the subfactor M/N is elementary abelian. The operation AbelianSubfactorAction returns a list [*phi* , *alpha* , *bas*] where *bas* is a list of elements of *M* which are representatives for a basis of M/N, *alpha* is a map from *M* into a n-dimensional row space over $GF(p)$ where $[M : N] = p^n$ that is the natural homomorphism of *M* by *N* with the quotient represented as an additive group. Finally *phi* is a homomorphism from *G* into $GL_n(p)$ that represents the action of *G* on the factor M/N.

Note: If only matrices for the action are needed, LinearActionLayer (45.14.3) might be faster.

Example

```
gap> g:=Group((1,8,10,7,3,5)(2,4,12,9,11,6),
>             (1,9,5,6,3,10)(2,11,12,8,4,7));;
gap> c:=ChiefSeries(g);;List(c,Size);
[ 96, 48, 16, 4, 1 ]
gap> HasElementaryAbelianFactorGroup(c[3],c[4]);
true
gap> SetName(c[3],"my_group");;
gap> a:=AbelianSubfactorAction(g,c[3],c[4]);
[ [ (1,8,10,7,3,5)(2,4,12,9,11,6), (1,9,5,6,3,10)(2,11,12,8,4,7) ] ->
    [ <an immutable 2x2 matrix over GF2>,
      <an immutable 2x2 matrix over GF2> ],
  MappingByFunction( my_group, ( GF(2)^
    2 ), function( e ) ... end, function( r ) ... end ),
  Pcgs([ (2,9,3,8)(4,11,5,10), (1,6,12,7)(4,10,5,11) ]) ]
gap> mat:=Image(a[1],g);
Group([ <an immutable 2x2 matrix over GF2>,
  <an immutable 2x2 matrix over GF2> ])
gap> Size(mat);
3
gap> e:=PreImagesRepresentative(a[2],[Z(2),0*Z(2)]);
(2,9,3,8)(4,11,5,10)
gap> e in c[3];e in c[4];
true
false
```

41.9 Permutations Induced by Elements and Cycles

If only the permutation image of a single element is needed, it might not be worth to create the action homomorphism, the following operations yield the permutation image and cycles of a single element.

41.9.1 Permutation

▷ Permutation(*g, Omega[, gens, acts][, act]*) (function)

▷ Permutation(*g, xset*) (function)

computes the permutation that corresponds to the action of *g* on the permutation domain *Omega* (a list of objects that are permuted). If an external set *xset* is given, the permutation domain is the

HomeEnumerator (41.12.5) value of this external set (see Section 41.12). Note that the points of the returned permutation refer to the positions in *Omega*, even if *Omega* itself consists of integers.

If *g* does not leave the domain invariant, or does not map the domain injectively then fail is returned.

41.9.2 PermutationCycle

▷ PermutationCycle(*g, Omega, pnt[, act]*) (operation)

computes the permutation that represents the cycle of *pnt* under the action of the element *g*.

Example

```
gap> Permutation([[Z(3),-Z(3)],[Z(3),0*Z(3)]],AsList(GF(3)^2));
(2,7,6)(3,4,8)
gap> Permutation((1,2,3)(4,5)(6,7),[4..7]);
(1,2)(3,4)
gap> PermutationCycle((1,2,3)(4,5)(6,7),[4..7],4);
(1,2)
```

41.9.3 Cycle

▷ Cycle(*g, Omega, pnt[, act]*) (operation)

returns a list of the points in the cycle of *pnt* under the action of the element *g*.

41.9.4 CycleLength

▷ CycleLength(*g, Omega, pnt[, act]*) (operation)

returns the length of the cycle of *pnt* under the action of the element *g*.

41.9.5 Cycles

▷ Cycles(*g, Omega[, act]*) (operation)

returns a list of the cycles (as lists of points) of the action of the element *g*.

41.9.6 CycleLengths

▷ CycleLengths(*g, Omega[, act]*) (operation)

returns the lengths of all the cycles under the action of the element *g* on *Omega*.

Example

```
gap> Cycle((1,2,3)(4,5)(6,7),[4..7],4);
[ 4, 5 ]
gap> CycleLength((1,2,3)(4,5)(6,7),[4..7],4);
2
gap> Cycles((1,2,3)(4,5)(6,7),[4..7]);
[ [ 4, 5 ], [ 6, 7 ] ]
gap> CycleLengths((1,2,3)(4,5)(6,7),[4..7]);
[ 2, 2 ]
```

41.9.7 CycleIndex

▷ `CycleIndex(g, Omega[, act])` (function)
▷ `CycleIndex(G, Omega[, act])` (function)

The *cycle index* of a permutation g acting on `Omega` is defined as

$$z(g) = s_1^{c_1} s_2^{c_2} \cdots s_n^{c_n}$$

where c_k is the number of k-cycles in the cycle decomposition of g and the s_i are indeterminates.

The *cycle index* of a group `G` is defined as

$$Z(G) = \left(\sum_{g \in G} z(g) \right) / |G|.$$

The indeterminates used by `CycleIndex` are the indeterminates 1 to n over the rationals (see `Indeterminate` (66.1.1)).

Example

```
gap> g:=TransitiveGroup(6,8);
S_4(6c) = 1/2[2^3]S(3)
gap> CycleIndex(g);
1/24*x_1^6+1/8*x_1^2*x_2^2+1/4*x_1^2*x_4+1/4*x_2^3+1/3*x_3^2
```

41.10 Tests for Actions

41.10.1 IsTransitive

▷ `IsTransitive(G, Omega[, gens, acts][, act])` (operation)
▷ `IsTransitive(G)` (property)
▷ `IsTransitive(xset)` (property)

returns `true` if the action implied by the arguments is transitive, or `false` otherwise.

We say that a group `G` acts *transitively* on a domain D if and only if for every pair of points $d, e \in D$ there is an element g in `G` such that $d^g = e$.

For permutation groups, the syntax `IsTransitive(G)` is also permitted and tests whether the group is transitive on the points moved by it, that is the group $\langle (2,3,4), (2,3) \rangle$ is transitive (on 3 points).

41.10.2 Transitivity

▷ `Transitivity(G, Omega[, gens, acts][, act])` (operation)
▷ `Transitivity(xset)` (attribute)

returns the degree k (a non-negative integer) of transitivity of the action implied by the arguments, i.e. the largest integer k such that the action is k-transitive. If the action is not transitive `0` is returned.

An action is *k-transitive* if every k-tuple of points can be mapped simultaneously to every other k-tuple.

Example

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> IsTransitive(g,[1..5]);
false
gap> Transitivity(g,[1..4]);
2
```

41.10.3 RankAction

▷ RankAction(*G, Omega[, gens, acts][, act]*) (operation)
▷ RankAction(*xset*) (attribute)

returns the rank of a transitive action, i.e. the number of orbits of the point stabilizer.

Example

```
gap> RankAction(g,Combinations([1..4],2),OnSets);
4
```

41.10.4 IsSemiRegular

▷ IsSemiRegular(*G, Omega[, gens, acts][, act]*) (operation)
▷ IsSemiRegular(*xset*) (property)

returns `true` if the action implied by the arguments is semiregular, or `false` otherwise.
An action is *semiregular* is the stabilizer of each point is the identity.

41.10.5 IsRegular

▷ IsRegular(*G, Omega[, gens, acts][, act]*) (operation)
▷ IsRegular(*xset*) (property)

returns `true` if the action implied by the arguments is regular, or `false` otherwise.

An action is *regular* if it is both semiregular (see `IsSemiRegular` (41.10.4)) and transitive (see `IsTransitive` (41.10.1)). In this case every point *pnt* of *Omega* defines a one-to-one correspondence between *G* and *Omega*.

Example

```
gap> IsSemiRegular(g,Arrangements([1..4],3),OnTuples);
true
gap> IsRegular(g,Arrangements([1..4],3),OnTuples);
false
```

41.10.6 Earns

▷ Earns(*G, Omega[, gens, acts][, act]*) (operation)
▷ Earns(*xset*) (attribute)

returns a list of the elementary abelian regular (when acting on *Omega*) normal subgroups of *G*.
At the moment only methods for a primitive group *G* are implemented.

41.10.7 IsPrimitive

▷ `IsPrimitive(`*G, Omega[, gens, acts][, act]*`)` (operation)
▷ `IsPrimitive(`*xset*`)` (property)

returns `true` if the action implied by the arguments is primitive, or `false` otherwise.

An action is *primitive* if it is transitive and the action admits no nontrivial block systems. See 41.11.

Example

```
gap> IsPrimitive(g,Orbit(g,(1,2)(3,4)));
true
```

41.11 Block Systems

A *block system* (system of imprimitivity) for the action of a group G on an action domain Ω is a partition of Ω which –as a partition– remains invariant under the action of G.

41.11.1 Blocks

▷ `Blocks(`*G, Omega[, seed][, gens, acts][, act]*`)` (operation)
▷ `Blocks(`*xset[, seed]*`)` (attribute)

computes a block system for the action. If *seed* is not given and the action is imprimitive, a minimal nontrivial block system will be found. If *seed* is given, a block system in which *seed* is the subset of one block is computed. The action must be transitive.

Example

```
gap> g:=TransitiveGroup(8,3);
E(8)=2[x]2[x]2
gap> Blocks(g,[1..8]);
[ [ 1, 8 ], [ 2, 3 ], [ 4, 5 ], [ 6, 7 ] ]
gap> Blocks(g,[1..8],[1,4]);
[ [ 1, 4 ], [ 2, 7 ], [ 3, 6 ], [ 5, 8 ] ]
```

(See Section 41.2 for information about specific actions.)

41.11.2 MaximalBlocks

▷ `MaximalBlocks(`*G, Omega[, seed][, gens, acts][, act]*`)` (operation)
▷ `MaximalBlocks(`*xset[, seed]*`)` (attribute)

returns a block system that is maximal (i.e., blocks are maximal with respect to inclusion) for the action of *G* on *Omega*. If *seed* is given, a block system is computed in which *seed* is a subset of one block.

Example

```
gap> MaximalBlocks(g,[1..8]);
[ [ 1, 2, 3, 8 ], [ 4, 5, 6, 7 ] ]
```

41.11.3 RepresentativesMinimalBlocks

▷ RepresentativesMinimalBlocks(*G, Omega[, gens, acts][, act]*) (operation)
▷ RepresentativesMinimalBlocks(*xset*) (attribute)

computes a list of block representatives for all minimal (i.e blocks are minimal with respect to inclusion) nontrivial block systems for the action.

Example

```
gap> RepresentativesMinimalBlocks(g,[1..8]);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 1, 7 ],
  [ 1, 8 ] ]
```

41.11.4 AllBlocks

▷ AllBlocks(*G*) (attribute)

computes a list of representatives of all block systems for a permutation group *G* acting transitively on the points moved by the group.

Example

```
gap> AllBlocks(g);
[ [ 1, 8 ], [ 1, 2, 3, 8 ], [ 1, 4, 5, 8 ], [ 1, 6, 7, 8 ], [ 1, 3 ],
  [ 1, 3, 5, 7 ], [ 1, 3, 4, 6 ], [ 1, 5 ], [ 1, 2, 5, 6 ], [ 1, 2 ],
  [ 1, 2, 4, 7 ], [ 1, 4 ], [ 1, 7 ], [ 1, 6 ] ]
```

The stabilizer of a block can be computed via the action OnSets (41.2.4):

Example

```
gap> Stabilizer(g,[1,8],OnSets);
Group([ (1,8)(2,3)(4,5)(6,7) ])
```

If bs is a partition of the action domain, given as a set of sets, the stabilizer under the action OnSetsDisjointSets (41.2.8) returns the largest subgroup which preserves bs as a block system.

Example

```
gap> g:=Group((1,2,3,4,5,6,7,8),(1,2));;
gap> bs:=[[1,2,3,4],[5,6,7,8]];;
gap> Stabilizer(g,bs,OnSetsDisjointSets);
Group([ (6,7), (5,6), (5,8), (2,3), (3,4)(5,7), (1,4),
  (1,5,4,8)(2,6,3,7) ])
```

41.12 External Sets

When considering group actions, sometimes the concept of a *G-set* is used. This is a set Ω endowed with an action of G. The elements of the G-set are the same as those of Ω, however concepts like equality and equivalence of G-sets do not only consider the underlying domain Ω but the group action as well.

This concept is implemented in GAP via *external sets*.

The constituents of an external set are stored in the attributes ActingDomain (41.12.3), FunctionAction (41.12.4) and HomeEnumerator (41.12.5).

Most operations for actions are applicable as an attribute for an external set.

The most prominent external subsets are orbits, see `ExternalOrbit` (41.12.9).

Many subsets of a group, such as conjugacy classes or cosets (see `ConjugacyClass` (39.10.1) and `RightCoset` (39.7.1)) are implemented as external orbits.

External sets also are implicitly underlying action homomorphisms, see `UnderlyingExternalSet` (41.12.16) and `SurjectiveActionHomomorphismAttr` (41.12.17).

41.12.1 IsExternalSet

▷ `IsExternalSet(`*obj*`)` (Category)

An *external set* specifies a group action $\mu : \Omega \times G \mapsto \Omega$ of a group G on a domain Ω. The external set knows the group, the domain and the actual acting function. Mathematically, an external set is the set Ω, which is endowed with the action of a group G via the group action μ. For this reason GAP treats an external set as a domain whose elements are the elements of Ω. An external set is always a union of orbits. Currently the domain Ω must always be finite. If Ω is not a list, an enumerator for Ω is automatically chosen, see `Enumerator` (30.3.2).

41.12.2 ExternalSet

▷ `ExternalSet(`*G, Omega[, gens, acts][, act]*`)` (operation)

creates the external set for the action *act* of *G* on *Omega*. *Omega* can be either a proper set, or a domain which is represented as described in 12.4 and 30, or (to use less memory but with a slower performance) an enumerator (see `Enumerator` (30.3.2)) of this domain.

Example

```
gap> g:=Group((1,2,3),(2,3,4));;
gap> e:=ExternalSet(g,[1..4]);
<xset:[ 1, 2, 3, 4 ]>
gap> e:=ExternalSet(g,g,OnRight);
<xset:[ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2),
  (1,3,4), (1,3)(2,4), (1,4,2), (1,4,3), (1,4)(2,3) ]>
gap> Orbits(e);
[ [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3), (2,4,3), (1,4,2),
      (1,2,3), (1,3,4), (2,3,4), (1,3,2), (1,4,3), (1,2,4) ] ]
```

41.12.3 ActingDomain

▷ `ActingDomain(`*xset*`)` (attribute)

This attribute returns the group with which the external set *xset* was defined.

41.12.4 FunctionAction

▷ `FunctionAction(`*xset*`)` (attribute)

is the acting function with which the external set *xset* was defined.

41.12.5 HomeEnumerator

▷ `HomeEnumerator(xset)` (attribute)

returns an enumerator of the action domain with which the external set *xset* was defined. For external subsets, this is in general different from the `Enumerator` (30.3.2) value of *xset*, which enumerates only the subset.

Example

```
gap> ActingDomain(e);
Group([ (1,2,3), (2,3,4) ])
gap> FunctionAction(e)=OnRight;
true
gap> HomeEnumerator(e);
[ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2),
  (1,3,4), (1,3)(2,4), (1,4,2), (1,4,3), (1,4)(2,3) ]
```

41.12.6 IsExternalSubset

▷ `IsExternalSubset(obj)` (Representation)

An external subset is the restriction of an external set to a subset of the domain (which must be invariant under the action). It is again an external set.

41.12.7 ExternalSubset

▷ `ExternalSubset(G, xset, start[, gens, acts], act)` (operation)

constructs the external subset of *xset* on the union of orbits of the points in *start*.

41.12.8 IsExternalOrbit

▷ `IsExternalOrbit(obj)` (Representation)

An external orbit is an external subset consisting of one orbit.

41.12.9 ExternalOrbit

▷ `ExternalOrbit(G, Omega, pnt[, gens, acts], act)` (operation)

constructs the external subset on the orbit of *pnt*. The `Representative` (30.4.7) value of this external set is *pnt*.

Example

```
gap> e:=ExternalOrbit(g,g,(1,2,3));
(1,2,3)^G
```

41.12.10 StabilizerOfExternalSet

▷ StabilizerOfExternalSet(*xset*) (attribute)

computes the stabilizer of the Representative (30.4.7) value of the external set *xset*. The stabilizer will have the acting group of *xset* as its parent.

Example

```
gap> Representative(e);
(1,2,3)
gap> StabilizerOfExternalSet(e);
Group([ (1,2,3) ])
```

41.12.11 ExternalOrbits

▷ ExternalOrbits(*G, Omega[, gens, acts][, act]*) (operation)
▷ ExternalOrbits(*xset*) (attribute)

computes a list of external orbits that give the orbits of *G*.

Example

```
gap> ExternalOrbits(g,AsList(g));
[ ()^G, (2,3,4)^G, (2,4,3)^G, (1,2)(3,4)^G ]
```

41.12.12 ExternalOrbitsStabilizers

▷ ExternalOrbitsStabilizers(*G, Omega[, gens, acts][, act]*) (operation)
▷ ExternalOrbitsStabilizers(*xset*) (attribute)

In addition to ExternalOrbits (41.12.11), this operation also computes the stabilizers of the representatives of the external orbits at the same time. (This can be quicker than computing the ExternalOrbits (41.12.11) value first and the stabilizers afterwards.)

Example

```
gap> e:=ExternalOrbitsStabilizers(g,AsList(g));
[ ()^G, (2,3,4)^G, (2,4,3)^G, (1,2)(3,4)^G ]
gap> HasStabilizerOfExternalSet(e[3]);
true
gap> StabilizerOfExternalSet(e[3]);
Group([ (2,4,3) ])
```

41.12.13 CanonicalRepresentativeOfExternalSet

▷ CanonicalRepresentativeOfExternalSet(*xset*) (attribute)

The canonical representative of an external set *xset* may only depend on the defining attributes *G*, *Omega*, *act* of *xset* and (in the case of external subsets) Enumerator(*xset*). It must *not* depend, e.g., on the representative of an external orbit. GAP does not know methods for arbitrary external sets to compute a canonical representative, see CanonicalRepresentativeDeterminatorOfExternalSet (41.12.14).

41.12.14 CanonicalRepresentativeDeterminatorOfExternalSet

▷ CanonicalRepresentativeDeterminatorOfExternalSet(*xset*) (attribute)

returns a function that takes as its arguments the acting group and a point. This function returns a list of length 1 or 3, the first entry being the canonical representative and the other entries (if bound) being the stabilizer of the canonical representative and a conjugating element, respectively. An external set is only guaranteed to be able to compute a canonical representative if it has a CanonicalRepresentativeDeterminatorOfExternalSet.

41.12.15 ActorOfExternalSet

▷ ActorOfExternalSet(*xset*) (attribute)

returns an element mapping Representative(*xset*) to CanonicalRepresentativeOfExternalSet(*xset*) under the given action.

Example

```
gap> u:=Subgroup(g,[(1,2,3)]);;
gap> e:=RightCoset(u,(1,2)(3,4));;
gap> CanonicalRepresentativeOfExternalSet(e);
(2,4,3)
gap> ActorOfExternalSet(e);
(1,3,2)
gap> FunctionAction(e)((1,2)(3,4),last);
(2,4,3)
```

41.12.16 UnderlyingExternalSet

▷ UnderlyingExternalSet(*acthom*) (attribute)

The underlying set of an action homomorphism *acthom* is the external set on which it was defined.

Example

```
gap> g:=Group((1,2,3),(1,2));;
gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples);;
gap> s:=UnderlyingExternalSet(hom);
<xset:[[ 1, 2, 3 ],[ 1, 2, 4 ],[ 1, 3, 2 ],[ 1, 3, 4 ],[ 1, 4, 2 ],
[ 1, 4, 3 ],[ 2, 1, 3 ],[ 2, 1, 4 ],[ 2, 3, 1 ],[ 2, 3, 4 ],
[ 2, 4, 1 ],[ 2, 4, 3 ],[ 3, 1, 2 ],[ 3, 1, 4 ],[ 3, 2, 1 ], ...]>
gap> Print(s,"\n");
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 2 ], [ 1, 3, 4 ], [ 1, 4, 2 ],
  [ 1, 4, 3 ], [ 2, 1, 3 ], [ 2, 1, 4 ], [ 2, 3, 1 ], [ 2, 3, 4 ],
  [ 2, 4, 1 ], [ 2, 4, 3 ], [ 3, 1, 2 ], [ 3, 1, 4 ], [ 3, 2, 1 ],
  [ 3, 2, 4 ], [ 3, 4, 1 ], [ 3, 4, 2 ], [ 4, 1, 2 ], [ 4, 1, 3 ],
  [ 4, 2, 1 ], [ 4, 2, 3 ], [ 4, 3, 1 ], [ 4, 3, 2 ] ]
```

41.12.17 SurjectiveActionHomomorphismAttr

▷ SurjectiveActionHomomorphismAttr(*xset*) (attribute)

returns an action homomorphism for the external set *xset* which is surjective. (As the `Image` (32.4.6) value of this homomorphism has to be computed to obtain the range, this may take substantially longer than `ActionHomomorphism` (41.7.1).)

Chapter 42

Permutations

GAP offers a data type *permutation* to describe the elements of permutation groups.

The points on which permutations in GAP act are the positive integers up to a certain architecture dependent limit, and the image of a point i under a permutation p is written i^p, which is expressed as $i\hat{}p$ in GAP. (This action is also implemented by the function `OnPoints` (41.2.1).) If $i\hat{}p$ is different from i, we say that i is *moved* by p, otherwise it is *fixed*. Permutations in GAP are entered and displayed in cycle notation, such as `(1,2,3)(4,5)`.

The preimage of the point i under the permutation p can be computed as i/p, without constructing the inverse of p.

For arithmetic operations for permutations and their precedence, see 31.12.

In the names of the GAP functions that deal with permutations, the word "Permutation" is usually abbreviated to "Perm", to save typing. For example, the category test function for permutations is `IsPerm` (42.1.1).

42.1 IsPerm (Filter)

Internally, GAP stores a permutation as a list of the d images of the integers $1,\ldots,d$, where the "internal degree" d is the largest integer moved by the permutation or bigger. When a permutation is read in in cycle notation, d is always set to the largest moved integer, but a bigger d can result from a multiplication of two permutations, because the product is not shortened if it fixes d. The images are stored all as 16-bit integers or all as 32-bit integers, depending on whether $d \leq 65536$ or not. For example, if $m \geq 65536$, the permutation $(1,2,\ldots,m)$ has internal degree $d = m$ and takes $4m$ bytes of memory for storage. But — since the internal degree is not reduced — this means that the identity permutation `()` calculated as $(1,2,\ldots,m)*(1,2,\ldots,m)^{-1}$ also takes $4m$ bytes of storage. It can take even more because the internal list has sometimes room for more than d images.

The operation `RestrictedPerm` (42.5.4) reduces the storage degree of its result and therefore can be used to save memory if intermediate calculations in large degree result in a small degree result.

Permutations do not belong to a specific group. That means that one can work with permutations without defining a permutation group that contains them.

Example

```
gap> (1,2,3);
(1,2,3)
gap> (1,2,3) * (2,3,4);
(1,3)(2,4)
gap> 17^(2,5,17,9,8);
```

```
9
gap> OnPoints(17,(2,5,17,9,8));
9
```

The operation `Permuted` (21.20.18) can be used to permute the entries of a list according to a permutation.

42.1.1 IsPerm

▷ `IsPerm(obj)` (Category)

Each *permutation* in GAP lies in the category `IsPerm`. Basic operations for permutations are `LargestMovedPoint` (42.3.2), multiplication of two permutations via `*`, and exponentiation `^` with first argument a positive integer i and second argument a permutation π, the result being the image i^π of the point i under π.

42.1.2 IsPermCollection

▷ `IsPermCollection(obj)` (Category)
▷ `IsPermCollColl(obj)` (Category)

are the categories for collections of permutations and collections of collections of permutations, respectively.

42.1.3 PermutationsFamily

▷ `PermutationsFamily` (global variable)

is the family of all permutations.

42.2 Comparison of Permutations

42.2.1 \= (for permutations)

▷ `\=(p1, p2)` (method)
▷ `\<(p1, p2)` (method)

Two permutations are equal if they move the same points and all these points have the same images under both permutations.

The permutation *p1* is smaller than *p2* if $p1 \neq p2$ and $i^{p1} < i^{p2}$, where i is the smallest point with $i^{p1} \neq i^{p2}$. Therefore the identity permutation is the smallest permutation, see also Section 31.11.

Permutations can be compared with certain other GAP objects, see 4.12 for the details.

Example

```
gap> (1,2,3) = (2,3,1);
true
gap> (1,2,3) * (2,3,4) = (1,3)(2,4);
true
gap> (1,2,3) < (1,3,2);      # 1^(1,2,3) = 2 < 3 = 1^(1,3,2)
```

```
true
gap> (1,3,2,4) < (1,3,4,2);  # 2^(1,3,2,4) = 4 > 1 = 2^(1,3,4,2)
false
```

42.2.2 DistancePerms

▷ `DistancePerms(`*perm1, perm2*`)` (operation)

returns the number of points for which *perm1* and *perm2* have different images. This should always produce the same result as `NrMovePoints(`*perm1/perm2*`)` but some methods may be much faster than this form, since no new permutation needs to be created.

42.2.3 SmallestGeneratorPerm

▷ `SmallestGeneratorPerm(`*perm*`)` (function)

is the smallest permutation that generates the same cyclic group as the permutation *perm*. This is very efficient, even when *perm* has large order.

Example

```
gap> SmallestGeneratorPerm( (1,4,3,2) );
(1,2,3,4)
```

42.3 Moved Points of Permutations

42.3.1 SmallestMovedPoint (for a permutation)

▷ `SmallestMovedPoint(`*perm*`)` (attribute)
▷ `SmallestMovedPoint(`*C*`)` (attribute)

is the smallest positive integer that is moved by *perm* if such an integer exists, and `infinity` (18.2.1) if *perm* is the identity. For *C* a collection or list of permutations, the smallest value of `SmallestMovedPoint` for the elements of *C* is returned (and `infinity` (18.2.1) if *C* is empty).

42.3.2 LargestMovedPoint (for a permutation)

▷ `LargestMovedPoint(`*perm*`)` (attribute)
▷ `LargestMovedPoint(`*C*`)` (attribute)

For a permutation *perm*, this attribute contains the largest positive integer which is moved by *perm* if such an integer exists, and `0` if *perm* is the identity. For *C* a collection or list of permutations, the largest value of `LargestMovedPoint` for the elements of *C* is returned (and `0` if *C* is empty).

42.3.3 MovedPoints (for a permutation)

▷ `MovedPoints(`*perm*`)` (attribute)
▷ `MovedPoints(`*C*`)` (attribute)

is the proper set of the positive integers moved by at least one permutation in the collection *C*, respectively by the permutation *perm*.

42.3.4 NrMovedPoints (for a permutation)

▷ NrMovedPoints(*perm*) (attribute)
▷ NrMovedPoints(*C*) (attribute)

is the number of positive integers that are moved by *perm*, respectively by at least one element in the collection *C*. (The actual moved points are returned by MovedPoints (42.3.3).)

Example

```
gap> SmallestMovedPointPerm((4,5,6)(7,2,8));
2
gap> LargestMovedPointPerm((4,5,6)(7,2,8));
8
gap> NrMovedPointsPerm((4,5,6)(7,2,8));
6
gap> MovedPoints([(2,3,4),(7,6,3),(5,47)]);
[ 2, 3, 4, 5, 6, 7, 47 ]
gap> NrMovedPoints([(2,3,4),(7,6,3),(5,47)]);
7
gap> SmallestMovedPoint([(2,3,4),(7,6,3),(5,47)]);
2
gap> LargestMovedPoint([(2,3,4),(7,6,3),(5,47)]);
47
gap> LargestMovedPoint([()]);
0
```

42.4 Sign and Cycle Structure

42.4.1 SignPerm

▷ SignPerm(*perm*) (attribute)

The *sign* of a permutation *perm* is defined as $(-1)^k$ where k is the number of cycles of *perm* of even length.

The sign is a homomorphism from the symmetric group onto the multiplicative group $\{+1,-1\}$, the kernel of which is the alternating group.

42.4.2 CycleStructurePerm

▷ CycleStructurePerm(*perm*) (attribute)

is the cycle structure (i.e. the numbers of cycles of different lengths) of the permutation *perm*. This is encoded in a list l in the following form: The i-th entry of l contains the number of cycles of *perm* of length $i+1$. If *perm* contains no cycles of length $i+1$ it is not bound. Cycles of length 1 are ignored.

Example

```
gap> SignPerm((1,2,3)(4,5));
-1
gap> CycleStructurePerm((1,2,3)(4,5,9,7,8));
[ , 1,, 1 ]
```

42.5 Creating Permutations

42.5.1 ListPerm

▷ ListPerm(*perm[, length]*) (function)

is a list l that contains the images of the positive integers under the permutation *perm*. That means that l[i] $= i$^*perm*, where i lies between 1 and the largest point moved by *perm* (see LargestMovedPoint (42.3.2)).

An optional second argument specifies the length of the desired list.

42.5.2 PermList

▷ PermList(*list*) (function)

is the permutation π that moves points as described by the list *list*. That means that $i^\pi =$ *list*[i] if i lies between 1 and the length of *list*, and $i^\pi = i$ if i is larger than the length of the list *list*. PermList will return fail if *list* does not define a permutation, i.e., if *list* is not dense, or if *list* contains a positive integer twice, or if *list* contains an integer not in the range [1 .. Length(*list*)]. If *list* contains non-integer entries an error is raised.

42.5.3 MappingPermListList

▷ MappingPermListList(*src, dst*) (function)

Let *src* and *dst* be lists of positive integers of the same length, such that neither may contain an element twice. MappingPermListList returns a permutation π such that *src*[i]^π = *dst*[i]. The permutation π fixes all points larger than the maximum of the entries in *src* and *dst*. If there are several such permutations, it is not specified which of them MappingPermListList returns.

42.5.4 RestrictedPerm

▷ RestrictedPerm(*perm, list*) (operation)
▷ RestrictedPermNC(*perm, list*) (operation)

RestrictedPerm returns the new permutation that acts on the points in the list *list* in the same way as the permutation *perm*, and that fixes those points that are not in *list*. The resulting permutation is stored internally of degree given by the maximal entry of *list*. *list* must be a list of positive integers such that for each i in *list* the image i^*perm* is also in *list*, i.e., *list* must be the union of cycles of *perm*.

RestrictedPermNC does not check whether *list* is a union of cycles.

Example

```
gap> ListPerm((3,4,5));
[ 1, 2, 4, 5, 3 ]
gap> PermList([1,2,4,5,3]);
(3,4,5)
gap> MappingPermListList([2,5,1,6],[7,12,8,2]);
(1,8,5,12,11,10,9,6,2,7,4,3)
gap> RestrictedPerm((1,2)(3,4),[3..5]);
(3,4)
```

42.5.5 AsPermutation

▷ `AsPermutation(`*f*`)` (operation)

Returns: A permutation or `fail`.

Partial permutations and transformations which define permutations (mathematically) can be converted into GAP permutations using `AsPermutation`; see Chapters 53 and 54 for more details about transformations and partial permutations.

for partial permutations

If the partial permutation *f* is a permutation of its image, then `AsPermutation` returns this permutation. If *f* does not permute its image, then `fail` is returned.

for transformations

A transformation is a permutation if and only if its rank equals its degree. If a transformation in GAP is a permutation, then `AsPermutation` returns this permutation. If *f* is not a permutation, then `fail` is returned.

The function `Permutation` (41.9.1) can also be used to convert partial permutations and transformations into permutations where appropriate.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ],
> [ 2, 7, 9, 4, 1, 10, 5, 6, 3, 8 ] );
(1,2,7,5)(3,9)(4)(6,10,8)
gap> AsPermutation(f);
(1,2,7,5)(3,9)(6,10,8)
gap> f:= PartialPerm( [ 1, 2, 3, 4, 5, 7, 8 ], [ 5, 3, 8, 1, 9, 4, 10 ] );
[2,3,8,10][7,4,1,5,9]
gap> AsPermutation(f);
fail
gap> f:=Transformation( [ 5, 8, 3, 5, 8, 6, 2, 2, 7, 8 ] );;
gap> AsPermutation(f);
fail
gap> f:=Transformation( [ 1, 3, 6, 6, 2, 10, 2, 3, 10, 5 ] );;
gap> AsPermutation(f);
fail
gap> f:=Transformation( [ 2, 7, 9, 4, 1, 10, 5, 6, 3, 8 ] );
Transformation( [ 2, 7, 9, 4, 1, 10, 5, 6, 3, 8 ] )
gap> AsPermutation(f);
(1,2,7,5)(3,9)(6,10,8)
```

Chapter 43

Permutation Groups

43.1 IsPermGroup (Filter)

43.1.1 IsPermGroup

▷ `IsPermGroup(`*obj*`)` (Category)

A permutation group is a group of permutations on a finite set Ω of positive integers. GAP does *not* require the user to specify the operation domain Ω when a permutation group is defined.

Example

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
```

Permutation groups are groups and therefore all operations for groups (see Chapter 39) can be applied to them. In many cases special methods are installed for permutation groups that make computations more effective.

43.2 The Natural Action

The functions `MovedPoints` (42.3.3), `NrMovedPoints` (42.3.4), `LargestMovedPoint` (42.3.2), and `SmallestMovedPoint` (42.3.1) are defined for arbitrary collections of permutations (see 42.3), in particular they can be applied to permutation groups.

Example

```
gap> g:= Group( (2,3,5,6), (2,3) );;
gap> MovedPoints( g );  NrMovedPoints( g );
[ 2, 3, 5, 6 ]
4
gap> LargestMovedPoint( g );  SmallestMovedPoint( g );
6
2
```

The action of a permutation group on the positive integers is a group action (via the acting function `OnPoints` (41.2.1)). Therefore all action functions can be applied (see the Chapter 41), for example `Orbit` (41.4.1), `Stabilizer` (41.5.2), `Blocks` (41.11.1), `IsTransitive` (41.10.1), `IsPrimitive` (41.10.7).

If one has a list of group generators and is interested in the moved points (see above) or orbits, it may be useful to avoid the explicit construction of the group for efficiency reasons. For the special case of the action of permutations on positive integers via `^`, the functions `OrbitPerms` (43.2.1) and `OrbitsPerms` (43.2.2) are provided for this purpose.

Similarly, several functions concerning the natural action of permutation groups address stabilizer chains (see 43.6) rather than permutation groups themselves, for example `BaseStabChain` (43.10.1).

43.2.1 OrbitPerms

▷ `OrbitPerms(`*perms, pnt*`)` (function)

returns the orbit of the positive integer *pnt* under the group generated by the permutations in the list *perms*.

43.2.2 OrbitsPerms

▷ `OrbitsPerms(`*perms, D*`)` (function)

returns the list of orbits of the positive integers in the list *D* under the group generated by the permutations in the list *perms*.

Example

```
gap> OrbitPerms( [ (1,2,3)(4,5), (3,6) ], 1 );
[ 1, 2, 3, 6 ]
gap> OrbitsPerms( [ (1,2,3)(4,5), (3,6) ], [ 1 .. 6 ] );
[ [ 1, 2, 3, 6 ], [ 4, 5 ] ]
```

43.3 Computing a Permutation Representation

43.3.1 IsomorphismPermGroup

▷ `IsomorphismPermGroup(`*G*`)` (attribute)

returns an isomorphism from the group *G* onto a permutation group which is isomorphic to *G*. The method will select a suitable permutation representation.

Example

```
gap> g:=SmallGroup(24,12);
<pc group of size 24 with 4 generators>
gap> iso:=IsomorphismPermGroup(g);
<action isomorphism>
gap> Image(iso,g.3*g.4);
(1,12)(2,16)(3,19)(4,5)(6,22)(7,8)(9,23)(10,11)(13,24)(14,15)(17,
18)(20,21)
```

In many cases the permutation representation constructed by `IsomorphismPermGroup` is regular.

43.3.2 SmallerDegreePermutationRepresentation

▷ `SmallerDegreePermutationRepresentation(`*G*`)` (function)

Let *G* be a permutation group that acts transitively on its moved points. SmallerDegreePermutationRepresentation tries to find a faithful permutation representation of smaller degree. The result is a group homomorphism onto a permutation group, in the worst case this is the identity mapping on *G*.

If the cheap option is given, the function only tries to reduce to orbits or actions on blocks, otherwise also actions on cosets of random subgroups are tried.

Note that the result is not guaranteed to be a faithful permutation representation of smallest degree, or of smallest degree among the transitive permutation representations of *G*. Using GAP interactively, one might be able to choose subgroups of small index for which the cores intersect trivially; in this case, the actions on the cosets of these subgroups give rise to an intransitive permutation representation the degree of which may be smaller than the original degree.

The methods used might involve the use of random elements and the permutation representation (or even the degree of the representation) is not guaranteed to be the same for different calls of SmallerDegreePermutationRepresentation.

If the option cheap is given less work is spent on trying to get a small degree representation, if the value of this option is set to the string "skip" the identity mapping is returned. (This is useful if a function called internally might try a degree reduction.)

Example

```
gap> image:= Image( iso );;  NrMovedPoints( image );
24
gap> small:= SmallerDegreePermutationRepresentation( image );;
gap> Image( small );
Group([ (2,3), (2,3,4), (1,2)(3,4), (1,3)(2,4) ])
```

Example

```
gap> p:=Group((1,2,3,4,5,6),(1,2));;p:=Action(p,AsList(p),OnRight);;
gap> Length(MovedPoints(p));
720
gap> q:=SmallerDegreePermutationRepresentation(p);;
gap> NrMovedPoints(Image(q));
6
```

43.4 Symmetric and Alternating Groups

The commands SymmetricGroup (50.1.10) and AlternatingGroup (50.1.9) (see Section 50.1) construct symmetric and alternating permutation groups. GAP can also detect whether a given permutation group is a symmetric or alternating group on the set of its moved points; if so then the group is called a *natural* symmetric or alternating group, respectively.

The functions IsSymmetricGroup (43.4.2) and IsAlternatingGroup (43.4.3) can be used to check whether a given group (not necessarily a permutation group) is isomorphic to a symmetric or alternating group.

43.4.1 IsNaturalSymmetricGroup

▷ IsNaturalSymmetricGroup(*group*) (property)

▷ IsNaturalAlternatingGroup(*group*) (property)

A group is a natural symmetric or alternating group if it is a permutation group acting as symmetric or alternating group, respectively, on its moved points.

For groups that are known to be natural symmetric or natural alternating groups, very efficient methods for computing membership, conjugacy classes, Sylow subgroups etc. are used.

Example

```
gap> g:=Group((1,5,7,8,99),(1,99,13,72));;
gap> IsNaturalSymmetricGroup(g);
true
gap> g;
Sym( [ 1, 5, 7, 8, 13, 72, 99 ] )
gap> IsNaturalSymmetricGroup( Group( (1,2)(4,5), (1,2,3)(4,5,6) ) );
false
```

43.4.2 IsSymmetricGroup

▷ `IsSymmetricGroup(`*group*`)` (property)

is `true` if the group *group* is isomorphic to a symmetric group.

43.4.3 IsAlternatingGroup

▷ `IsAlternatingGroup(`*group*`)` (property)

is `true` if the group *group* is isomorphic to a alternating group.

43.4.4 SymmetricParentGroup

▷ `SymmetricParentGroup(`*grp*`)` (attribute)

For a permutation group *grp* this function returns the symmetric group that moves the same points as *grp* does.

Example

```
gap> SymmetricParentGroup( Group( (1,2), (4,5), (7,8,9) ) );
Sym( [ 1, 2, 4, 5, 7, 8, 9 ] )
```

43.5 Primitive Groups

43.5.1 ONanScottType

▷ `ONanScottType(`*G*`)` (attribute)

returns the type of a primitive permutation group *G*, according to the O'Nan-Scott classification. The labelling of the different types is not consistent in the literature, we use the following identifications. The two-letter code given is the name of the type as used by Praeger.

1 Affine. (HA)

2 Almost simple. (AS)

3a Diagonal, Socle consists of two normal subgroups. (HS)

3b Diagonal, Socle is minimal normal. (SD)

4a Product action with the first factor primitive of type 3a. (HC)

4b Product action with the first factor primitive of type 3b. (CD)

4c Product action with the first factor primitive of type 2. (PA)

5 Twisted wreath product (TW)

See [EH01] for correspondence to other labellings used in the literature. As it can contain letters, the type is returned as a string.

If `G` is not a permutation group or does not act primitively on the points moved by it, the result is undefined.

43.5.2 SocleTypePrimitiveGroup

▷ `SocleTypePrimitiveGroup(G)` (attribute)

returns the socle type of the primitive permutation group `G`. The socle of a primitive group is the direct product of isomorphic simple groups, therefore the type is indicated by a record with components `series`, `parameter` (both as described under `IsomorphismTypeInfoFiniteSimpleGroup` (39.15.12)), and `width` for the number of direct factors.

If `G` does not have a faithful primitive action, the result is undefined.

Example

```
gap> g:=AlternatingGroup(5);;
gap> h:=DirectProduct(g,g);;
gap> p:=List([1,2],i->Projection(h,i));;
gap> ac:=Action(h,AsList(g),
> function(g,h) return Image(p[1],h)^-1*g*Image(p[2],h);end);;
gap> Size(ac);NrMovedPoints(ac);IsPrimitive(ac,[1..60]);
3600
60
true
gap> ONanScottType(ac);
"3a"
gap> SocleTypePrimitiveGroup(ac);
rec(
  name := "A(5) ~ A(1,4) = L(2,4) ~ B(1,4) = O(3,4) ~ C(1,4) = S(2,4) \
~ 2A(1,4) = U(2,4) ~ A(1,5) = L(2,5) ~ B(1,5) = O(3,5) ~ C(1,5) = S(2,\
5) ~ 2A(1,5) = U(2,5)", parameter := 5, series := "A", width := 2 )
```

43.6 Stabilizer Chains

Many of the algorithms for permutation groups use a *stabilizer chain* of the group. The concepts of stabilizer chains, *bases*, and *strong generating sets* were introduced by Charles Sims in [Sim70]. A further discussion of base change is given in section 87.1.

Let $B = [b_1, \ldots, b_n]$ be a list of points, $G^{(1)} = G$ and $G^{(i+1)} = Stab_{G^{(i)}}(b_i)$, such that $G^{(n+1)} = \{()\}$. Then the list $[b_1, \ldots, b_n]$ is called a *base* of G, the points b_i are called *base points*. A set S of generators

for G satisfying the condition $\langle S \cap G^{(i)} \rangle = G^{(i)}$ for each $1 \leq i \leq n$, is called a *strong generating set* (SGS) of G. (More precisely we ought to say that it is a SGS of G *relative* to B). The chain of subgroups $G^{(i)}$ of G itself is called the *stabilizer chain* of G relative to B.

Since $[b_1, \ldots, b_n]$, where n is the degree of G and b_i are the moved points of G, certainly is a base for G there exists a base for each permutation group. The number of points in a base is called the *length* of the base. A base B is called *reduced* if there exists no i such that $G^{(i)} = G^{(i+1)}$. (This however does not imply that no subset of B could also serve as a base.) Note that different reduced bases for one permutation group G may have different lengths. For example, the irreducible degree 416 permutation representation of the Chevalley Group $G_2(4)$ possesses reduced bases of lengths 5 and 7.

Let $R^{(i)}$ be a right transversal of $G^{(i+1)}$ in $G^{(i)}$, i.e. a set of right coset representatives of the cosets of $G^{(i+1)}$ in $G^{(i)}$. Then each element g of G has a unique representation as a product of the form $g = r_n \ldots r_1$ with $r_i \in R^{(i)}$. The cosets of $G^{(i+1)}$ in $G^{(i)}$ are in bijective correspondence with the points in $O^{(i)} := b_i^{G^{(i)}}$. So we could represent a transversal as a list T such that $T[p]$ is a representative of the coset corresponding to the point $p \in O^{(i)}$, i.e., an element of $G^{(i)}$ that takes b_i to p. (Note that such a list has holes in all positions corresponding to points not contained in $O^{(i)}$.)

This approach however will store many different permutations as coset representatives which can be a problem if the degree n gets bigger. Our goal therefore is to store as few different permutations as possible such that we can still reconstruct each representative in $R^{(i)}$, and from them the elements in G. A *factorized inverse transversal* T is a list where $T[p]$ is a generator of $G^{(i)}$ such that $p^{T[p]}$ is a point that lies earlier in $O^{(i)}$ than p (note that we consider $O^{(i)}$ as a list, not as a set). If we assume inductively that we know an element $r \in G^{(i)}$ that takes b_i to $p^{T[p]}$, then $rT[p]^{-1}$ is an element in $G^{(i)}$ that takes b_i to p. GAP uses such factorized inverse transversals.

Another name for a factorized inverse transversal is a *Schreier tree*. The vertices of the tree are the points in $O^{(i)}$, and the root of the tree is b_i. The edges are defined as the ordered pairs $(p, p^{T[p]})$, for $p \in O^{(i)} \setminus \{b_i\}$. The edge $(p, p^{T[p]})$ is labelled with the generator $T[p]$, and the product of edge labels along the unique path from p to b_i is the inverse of the transversal element carrying b_i to p.

Before we describe the construction of stabilizer chains in 43.8, we explain in 43.7 the idea of using non-deterministic algorithms; this is necessary for understanding the options available for the construction of stabilizer chains. After that, in 43.9 it is explained how a stabilizer chain is stored in GAP, 43.10 lists operations for stabilizer chains, and 43.11 lists low level routines for manipulating stabilizer chains.

43.7 Randomized Methods for Permutation Groups

For most computations with permutation groups, it is crucial to construct stabilizer chains efficiently. Sims's original construction in [Sim70] is deterministic, and is called the Schreier-Sims algorithm, because it is based on Schreier's Lemma ([HJ59, p. 96]): given $K = \langle S \rangle$ and a transversal T for K mod L, one can obtain $|S||T|$ generators for L. This lemma is applied recursively, with consecutive point stabilizers $G^{(i)}$ and $G^{(i+1)}$ playing the role of K and L.

In permutation groups of large degree, the number of Schreier generators to be processed becomes too large, and the deterministic Schreier-Sims algorithm becomes impractical. Therefore, GAP uses randomized algorithms. The method selection process, which is quite different from Version 3, works the following way.

If a group acts on not more than a hundred points, Sims's original deterministic algorithm is applied. In groups of degree greater than hundred, a heuristic algorithm based on ideas in [BCFS91]

constructs a stabilizer chain. This construction is complemented by a verify-routine that either proves the correctness of the stabilizer chain or causes the extension of the chain to a correct one. The user can influence the verification process by setting the value of the record component `random` (cf. 43.8).

If the `random` value equals 1000 then a slight extension of an unpublished method of Sims is used. The outcome of this verification process is always correct. The user also can prescribe any integer x, $1 \leq x \leq 999$ as the value of `random`. In this case, a randomized verification process from [BCFS91] is applied, and the result of the stabilizer chain construction is guaranteed to be correct with probability at least $x/1000$. The practical performance of the algorithm is much better than the theoretical guarantee.

If the stabilizer chain is not correct then the elements in the product of transversals $R^{(m)}R^{(m-1)}\cdots R^{(1)}$ constitute a proper subset of the group G in question. This means that a membership test with this stabilizer chain returns `false` for all elements that are not in G, but it may also return `false` for some elements of G; in other words, the result `true` of a membership test is always correct, whereas the result `false` may be incorrect.

The construction and verification phases are separated because there are situations where the verification step can be omitted; if one happens to know the order of the group in advance then the randomized construction of the stabilizer chain stops as soon as the product of the lengths of the basic orbits of the chain equals the group order, and the chain will be correct (see the `size` option of the `StabChain` (43.8.1) command).

Although the worst case running time is roughly quadratic for Sims's verification and roughly linear for the randomized one, in most examples the running time of the stabilizer chain construction with `random` value 1000 (i.e., guaranteed correct output) is about the same as the running time of randomized verification with guarantee of at least 90 percent correctness. Therefore, we suggest to use the default value `random` $= 1000$. Possible uses of `random` values less than 1000 are when one has to run through a large collection of subgroups, and a low value of random is used to choose quickly a candidate for more thorough examination; another use is when the user suspects that the quadratic bottleneck of the guaranteed correct verification is hit.

We will give two examples to illustrate these ideas.

Example

```
gap> h:= SL(4,7);;
gap> o:= Orbit( h, [1,0,0,0]*Z(7)^0, OnLines );;
gap> op:= Action( h, o, OnLines );;
gap> NrMovedPoints( op );
400
```

We created a permutation group on 400 points. First we compute a guaranteed correct stabilizer chain (see `StabChain` (43.8.1)).

Example

```
gap> h:= Group( GeneratorsOfGroup( op ) );;
gap> StabChain( h );;  time;
1120
gap> Size( h );
2317591180800
```

Now randomized verification will be used. We require that the result is guaranteed correct with probability 90 percent. This means that if we would do this calculation many times over, GAP would *guarantee* that in least 90 percent of all calculations the result is correct. In fact the results are much

better than the guarantee, but we cannot promise that this will really happen. (For the meaning of the `random` component in the second argument of `StabChain` (43.8.1).)

First the group is created anew.

Example

```
gap> h:= Group( GeneratorsOfGroup( op ) );;
gap> StabChain( h, rec( random:= 900 ) );;  time;
1410
gap> Size( h );
2317591180800
```

The result is still correct, and the running time is actually somewhat slower. If you give the algorithm additional information so that it can check its results, things become faster and the result is guaranteed to be correct.

Example

```
gap> h:=Group( GeneratorsOfGroup( op ) );;
gap> SetSize( h, 2317591180800 );
gap> StabChain( h );;  time;
170
```

The second example gives a typical group when the verification with `random` value 1000 is slow. The problem is that the group has a stabilizer subgroup $G^{(i)}$ such that the fundamental orbit $O^{(i)}$ is split into a lot of orbits when we stabilize b_i and one additional point of $O^{(i)}$.

Example

```
gap> p1:=PermList(Concatenation([401],[1..400]));;
gap> p2:=PermList(List([1..400],i->(i*20 mod 401)));;
gap> d:=DirectProduct(Group(p1,p2),SymmetricGroup(5));;
gap> h:=Group(GeneratorsOfGroup(d));;
gap> StabChain(h);;time;Size(h);
1030
192480
gap> h:=Group(GeneratorsOfGroup(d));;
gap> StabChain(h,rec(random:=900));;time;Size(h);
570
192480
```

When stabilizer chains of a group G are created with `random` value less than 1000, this is noted in the group G, by setting of the record component `random` in the value of the attribute `StabChainOptions` (43.8.2) for G. As errors induced by the random methods might propagate, any group or homomorphism created from G inherits a `random` component in its `StabChainOptions` (43.8.2) value from the corresponding component for G.

A lot of algorithms dealing with permutation groups use randomized methods; however, if the initial stabilizer chain construction for a group is correct, these further methods will provide guaranteed correct output.

43.8 Construction of Stabilizer Chains

43.8.1 StabChain (for a group (and a record))

▷ `StabChain(`*`G[, options]`*`)` (function)

▷ `StabChain(`*`G, base`*`)` (function)

▷ StabChainOp(*G*, *options*) (operation)
▷ StabChainMutable(*G*) (attribute)
▷ StabChainMutable(*permhomom*) (attribute)
▷ StabChainImmutable(*G*) (attribute)

These commands compute a stabilizer chain for the permutation group *G*; additionally, StabChainMutable is also an attribute for the group homomorphism *permhomom* whose source is a permutation group.

(The mathematical background of stabilizer chains is sketched in 43.6, more information about the objects representing stabilizer chains in GAP can be found in 43.9.)

StabChainOp is an operation with two arguments *G* and *options*, the latter being a record which controls some aspects of the computation of a stabilizer chain (see below); StabChainOp returns a *mutable* stabilizer chain. StabChainMutable is a *mutable* attribute for groups or homomorphisms, its default method for groups is to call StabChainOp with empty options record. StabChainImmutable is an attribute with *immutable* values; its default method dispatches to StabChainMutable.

StabChain is a function with first argument a permutation group *G*, and optionally a record *options* as second argument. If the value of StabChainImmutable for *G* is already known and if this stabilizer chain matches the requirements of *options*, StabChain simply returns this stored stabilizer chain. Otherwise StabChain calls StabChainOp and returns an immutable copy of the result; additionally, this chain is stored as StabChainImmutable value for *G*. If no *options* argument is given, its components default to the global variable DefaultStabChainOptions (43.8.3). If *base* is a list of positive integers, the version StabChain(*G*, *base*) defaults to StabChain(*G*, rec(base:= *base*)).

If given, *options* is a record whose components specify properties of the desired stabilizer chain or which may help the algorithm. Default values for all of them can be given in the global variable DefaultStabChainOptions (43.8.3). The following options are supported.

base **(default an empty list)**
: A list of points, through which the resulting stabilizer chain shall run. For the base B of the resulting stabilizer chain *S* this means the following. If the reduced component of *options* is true then those points of base with nontrivial basic orbits form the initial segment of B, if the reduced component is false then base itself is the initial segment of B. Repeated occurrences of points in base are ignored. If a stabilizer chain for *G* is already known then the stabilizer chain is computed via a base change.

knownBase **(no default value)**
: A list of points which is known to be a base for the group. Such a known base makes it easier to test whether a permutation given as a word in terms of a set of generators is the identity, since it suffices to map the known base with each factor consecutively, rather than multiplying the whole permutations (which would mean to map every point). This speeds up the Schreier-Sims algorithm which is used when a new stabilizer chain is constructed; it will not affect a base change, however. The component knownBase bears no relation to the base component, you may specify a known base knownBase and a desired base base independently.

reduced **(default** true**)**
: If this is true the resulting stabilizer chain *S* is reduced, i.e., the case $G^{(i)} = G^{(i+1)}$ does not occur. Setting reduced to false makes sense only if the component base (see above) is also set; in this case all points of base will occur in the base B of *S*, even if they have trivial basic

orbits. Note that if base is just an initial segment of B, the basic orbits of the points in $B \setminus$ base are always nontrivial.

tryPcgs **(default** true**)**
: If this is true and either the degree is at most 100 or the group is known to be solvable, GAP will first try to construct a pcgs (see Chapter 45) for *G* which will succeed and implicitly construct a stabilizer chain if *G* is solvable. If *G* turns out non-solvable, one of the other methods will be used. This solvability check is comparatively fast, even if it fails, and it can save a lot of time if *G* is solvable.

random **(default** 1000**)**
: If the value is less than 1000, the resulting chain is correct with probability at least random/1000. The random option is explained in more detail in 43.7.

size **(default** Size(*G*) **if this is known, i.e., if** HasSize(*G*) **is** true**)**
: If this component is present, its value is assumed to be the order of the group *G*. This information can be used to prove that a non-deterministically constructed stabilizer chain is correct. In this case, GAP does a non-deterministic construction until the size is correct.

limit **(default** Size(Parent(*G*)) **or** StabChainOptions(Parent(*G*)).limit **if it is present)**
: If this component is present, it must be greater than or equal to the order of *G*. The stabilizer chain construction stops if size limit is reached.

43.8.2 StabChainOptions

▷ StabChainOptions(*G*) (attribute)

is a record that stores the options with which the stabilizer chain stored in StabChainImmutable (43.8.1) has been computed (see StabChain (43.8.1) for the options that are supported).

43.8.3 DefaultStabChainOptions

▷ DefaultStabChainOptions (global variable)

are the options for StabChain (43.8.1) which are set as default.

43.8.4 StabChainBaseStrongGenerators

▷ StabChainBaseStrongGenerators(*base, sgs, one*) (function)

Let *base* be a base for a permutation group G, and let *sgs* be a strong generating set for G with respect to *base*; *one* must be the appropriate identity element of G (see One (31.10.2), in most cases this will be ()). This function constructs a stabilizer chain without the need to find Schreier generators; so this is much faster than the other algorithms.

43.8.5 MinimalStabChain

▷ MinimalStabChain(*G*) (attribute)

returns the reduced stabilizer chain corresponding to the base $[1,2,3,4,\ldots]$.

43.9 Stabilizer Chain Records

If a permutation group has a stabilizer chain, this is stored as a recursive structure. This structure is itself a record `S` and it has

(1) components that provide information about one level $G^{(i)}$ of the stabilizer chain (which we call the "current stabilizer") and

(2) a component `stabilizer` that holds another such record, namely the stabilizer chain of the next stabilizer $G^{(i+1)}$.

This gives a recursive structure where the "outermost" record representing the "topmost" stabilizer is bound to the group record component `stabChain` and has the components explained below. Note: Since the structure is recursive, *never print a stabilizer chain!* (Unless you want to exercise the scrolling capabilities of your terminal.)

`identity`
: the identity element of the current stabilizer.

`labels`
: a list of permutations which contains labels for the Schreier tree of the current stabilizer, i.e., it contains elements for the factorized inverse transversal. The first entry in this list is always the `identity`. Note that GAP tries to arrange things so that the `labels` components are identical (i.e., the same GAP object) in every stabilizer of the chain; thus the `labels` of a stabilizer do not necessarily all lie in the this stabilizer (but see `genlabels` below).

`genlabels`
: a list of integers indexing some of the permutations in the `labels` component. The `labels` addressed in this way form a generating set for the current stabilizer. If the `genlabels` component is empty, the rest of the stabilizer chain represents the trivial subgroup, and can be ignored, e.g., when calculating the size.

`generators`
: a list of generators for the current stabilizer. Usually, it is `labels{ genlabels }`.

`orbit`
: the vertices of the Schreier tree, which form the basic orbit $b_i^{G^{(i)}}$, ordered in such a way that the base point b_i is in the first position in the orbit.

`transversal`
: The factorized inverse transversal found during the orbit algorithm. The element g stored at `transversal`$[i]$ will map i to another point j that in the Schreier tree is closer to the base point. By iterated application (`transversal`$[j]$ and so on) eventually the base point is reached and an element that maps i to the base point found as product.

`translabels`
: An index list such that `transversal`$[j] =$ `labels`$[$ `translabels`$[j]]$. This list takes up comparatively little memory and is used to speed up base changes.

`stabilizer`
: If the current stabilizer is not yet the trivial group, the stabilizer chain continues with the

stabilizer of the current base point, which is again represented as a record with components `labels`, `identity`, `genlabels`, `generators`, `orbit`, `translabels`, `transversal` (and perhaps `stabilizer`). This record is bound to the `stabilizer` component of the current stabilizer. The last member of a stabilizer chain is recognized by the fact that it has no `stabilizer` component bound.

It is possible that different stabilizer chains share the same record as one of their iterated `stabilizer` components.

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> StabChain(g);
<stabilizer chain record, Base [ 1, 2, 3 ], Orbit length 4, Size: 24>
gap> BaseOfGroup(g);
[ 1, 2, 3 ]
gap> StabChainOptions(g);
rec( random := 1000 )
gap> DefaultStabChainOptions;
rec( random := 1000, reduced := true, tryPcgs := true )
```

43.10 Operations for Stabilizer Chains

43.10.1 BaseStabChain

▷ `BaseStabChain(`*S*`)` (function)

returns the base belonging to the stabilizer chain *S*.

43.10.2 BaseOfGroup

▷ `BaseOfGroup(`*G*`)` (attribute)

returns a base of the permutation group *G*. There is *no* guarantee that a stabilizer chain stored in *G* corresponds to this base!

43.10.3 SizeStabChain

▷ `SizeStabChain(`*S*`)` (function)

returns the product of the orbit lengths in the stabilizer chain *S*, that is, the order of the group described by *S*.

43.10.4 StrongGeneratorsStabChain

▷ `StrongGeneratorsStabChain(`*S*`)` (function)

returns a strong generating set corresponding to the stabilizer chain *S*.

43.10.5 GroupStabChain

▷ `GroupStabChain([G, ]S)` (function)

constructs a permutation group with stabilizer chain `S`, i.e., a group with generators `Generators( S )` to which `S` is assigned as component `stabChain`. If the optional argument `G` is given, the result will have the parent `G`.

43.10.6 OrbitStabChain

▷ `OrbitStabChain(S, pnt)` (function)

returns the orbit of `pnt` under the group described by the stabilizer chain `S`.

43.10.7 IndicesStabChain

▷ `IndicesStabChain(S)` (function)

returns a list of the indices of the stabilizers in the stabilizer chain `S`.

43.10.8 ListStabChain

▷ `ListStabChain(S)` (function)

returns a list that contains at position i the stabilizer of the first $i-1$ base points in the stabilizer chain `S`.

43.10.9 ElementsStabChain

▷ `ElementsStabChain(S)` (function)

returns a list of all elements of the group described by the stabilizer chain `S`.

43.10.10 InverseRepresentative

▷ `InverseRepresentative(S, pnt)` (function)

calculates the transversal element which maps `pnt` back to the base point of `S`. It just runs back through the Schreier tree from `pnt` to the root and multiplies the labels along the way.

43.10.11 SiftedPermutation

▷ `SiftedPermutation(S, g)` (function)

sifts the permutation `g` through the stabilizer chain `S` and returns the result after the last step.

The element `g` is sifted as follows: `g` is replaced by `g * InverseRepresentative( S, S.orbit[1]^g )`, then `S` is replaced by `S.stabilizer` and this process is repeated until `S` is trivial or `S.orbit[1]^g` is not in the basic orbit `S.orbit`. The remainder `g` is returned, it is the identity permutation if and only if the original `g` is in the group G described by the original `S`.

43.10.12 MinimalElementCosetStabChain

▷ `MinimalElementCosetStabChain(S, g)` (function)

Let G be the group described by the stabilizer chain `S`. This function returns a permutation h such that $Gg = Gh$ (that is, $g/h \in G$) and with the additional property that the list of images under h of the base belonging to `S` is minimal w.r.t. lexicographical ordering.

43.10.13 LargestElementStabChain

▷ `LargestElementStabChain(S, id)` (function)

Let G be the group described by the stabilizer chain `S`. This function returns the element $h \in G$ with the property that the list of images under h of the base belonging to `S` is maximal w.r.t. lexicographical ordering. The second argument must be an identity element (used to start the recursion).

43.10.14 ApproximateSuborbitsStabilizerPermGroup

▷ `ApproximateSuborbitsStabilizerPermGroup(G, pnt)` (function)

returns an approximation of the orbits of `Stabilizer( G, pnt )` on all points of the orbit `Orbit( G, pnt )`, without computing the full point stabilizer; As not all Schreier generators are used, the result may represent the orbits of only a subgroup of the point stabilizer.

43.11 Low Level Routines to Modify and Create Stabilizer Chains

These operations modify a stabilizer chain or obtain new chains with specific properties. They are rather technical and should only be used if such low-level routines are deliberately required. (For all functions in this section the parameter `S` is a stabilizer chain.)

43.11.1 CopyStabChain

▷ `CopyStabChain(S)` (function)

This function returns a copy of the stabilizer chain `S` that has no mutable object (list or record) in common with `S`. The `labels` components of the result are possibly shared by several levels, but superfluous labels are removed. (An entry in `labels` is superfluous if it does not occur among the `genlabels` or `translabels` on any of the levels which share that `labels` component.)

This is useful for stabiliser sub-chains that have been obtained as the (iterated) `stabilizer` component of a bigger chain.

43.11.2 CopyOptionsDefaults

▷ `CopyOptionsDefaults(G, options)` (function)

sets components in a stabilizer chain options record *options* according to what is known about the group `G`. This can be used to obtain a new stabilizer chain for `G` quickly.

43.11.3 ChangeStabChain

▷ ChangeStabChain(*S, base[, reduced]*) (function)

changes or reduces a stabilizer chain *S* to be adapted to the base *base*. The optional argument *reduced* is interpreted as follows.

reduced = false:
: change the stabilizer chain, do not reduce it,

reduced = true:
: change the stabilizer chain, reduce it.

43.11.4 ExtendStabChain

▷ ExtendStabChain(*S, base*) (function)

extends the stabilizer chain *S* so that it corresponds to base *base*. The original base of *S* must be a subset of *base*.

43.11.5 ReduceStabChain

▷ ReduceStabChain(*S*) (function)

changes the stabilizer chain *S* to a reduced stabilizer chain by eliminating trivial steps.

43.11.6 RemoveStabChain

▷ RemoveStabChain(*S*) (function)

S must be a stabilizer record in a stabilizer chain. This chain then is cut off at *S* by changing the entries in *S*. This can be used to remove trailing trivial steps.

43.11.7 EmptyStabChain

▷ EmptyStabChain(*labels, id[, pnt]*) (function)

constructs a stabilizer chain for the trivial group with identity value equal to*id* and labels = $\{id\} \cup$ *labels* (but of course with genlabels and generators values an empty list). If the optional third argument *pnt* is present, the only stabilizer of the chain is initialized with the one-point basic orbit [*pnt*] and with translabels and transversal components.

43.11.8 InsertTrivialStabilizer

▷ InsertTrivialStabilizer(*S, pnt*) (function)

InsertTrivialStabilizer initializes the current stabilizer with *pnt* as EmptyStabChain (43.11.7) did, but assigns the original *S* to the new *S*.stabilizer component, such that a new level with trivial basic orbit (but identical labels and ShallowCopyed genlabels and generators)

is inserted. This function should be used only if *pnt* really is fixed by the generators of *S*, because then new generators can be added and the orbit and transversal at the same time extended with `AddGeneratorsExtendSchreierTree` (43.11.10).

43.11.9 IsFixedStabilizer

▷ `IsFixedStabilizer(`*S, pnt*`)` (function)

returns `true` if *pnt* is fixed by all generators of *S* and `false` otherwise.

43.11.10 AddGeneratorsExtendSchreierTree

▷ `AddGeneratorsExtendSchreierTree(`*S, new*`)` (function)

adds the elements in *new* to the list of generators of *S* and at the same time extends the orbit and transversal. This is the only legal way to extend a Schreier tree (because this involves careful handling of the tree components).

43.12 Backtrack

A main use for stabilizer chains is in backtrack algorithms for permutation groups. GAP implements a partition-backtrack algorithm as described in [Leo91] and refined in [The97].

43.12.1 SubgroupProperty

▷ `SubgroupProperty(`*G, Pr[, L]*`)` (function)

Pr must be a one-argument function that returns `true` or `false` for elements of the group *G*, and the subset of elements of *G* that fulfill *Pr* must be a subgroup. (*If the latter is not true the result of this operation is unpredictable!*) This command computes this subgroup. The optional argument *L* must be a subgroup of the set of all elements in *G* fulfilling *Pr* and can be given if known in order to speed up the calculation.

43.12.2 ElementProperty

▷ `ElementProperty(`*G, Pr[, L[, R]]*`)` (function)

`ElementProperty` returns an element π of the permutation group *G* such that the one-argument function *Pr* returns `true` for π. It returns `fail` if no such element exists in *G*. The optional arguments *L* and *R* are subgroups of *G* such that the property *Pr* has the same value for all elements in the cosets *L g* and *g R*, respectively, with $g \in G$.

A typical example of using the optional subgroups *L* and *R* is the conjugacy test for elements a and b for which one can set $L := C_G(a)$ and $R := C_G(b)$.

Example

```
gap> propfun:= el -> (1,2,3)^el in [ (1,2,3), (1,3,2) ];;
gap> SubgroupProperty( g, propfun, Subgroup( g, [ (1,2,3) ] ) );
Group([ (1,2,3), (2,3) ])
```

```
gap> ElementProperty( g, el -> Order( el ) = 2 );
(2,4)
```

Chapter 42 describes special operations to construct permutations in the symmetric group without using backtrack constructions.

Backtrack routines are also called by the methods for permutation groups that compute centralizers, normalizers, intersections, conjugating elements as well as stabilizers for the operations of a permutation group via `OnPoints` (41.2.1), `OnSets` (41.2.4), `OnTuples` (41.2.5) and `OnSetsSets` (41.2.7). Some of these methods use more specific refinements than `SubgroupProperty` (43.12.1) or `ElementProperty`. For the definition of refinements, and how one can define refinements, see Section 87.2.

43.12.3 TwoClosure

▷ `TwoClosure(`*G*`)` (attribute)

The *2-closure* of a transitive permutation group *G* on *n* points is the largest subgroup of the symmetric group S_n which has the same orbits on sets of ordered pairs of points as the group *G* has. It also can be interpreted as the stabilizer of the orbital graphs of *G*.

Example

```
gap> TwoClosure(Group((1,2,3),(2,3,4)));
Sym( [ 1 .. 4 ] )
```

43.12.4 InfoBckt

▷ `InfoBckt` (info class)

is the info class for the partition backtrack routines.

43.13 Working with large degree permutation groups

Permutation groups of large degree (usually at least a few 10000) can pose a challenge to the heuristics used in the algorithms for permutation groups. This section lists a few useful tricks that may speed up calculations with such large groups enormously.

The first aspect concerns solvable groups: A lot of calculations (including an initial stabilizer chain computation thanks to the algorithm from [Sim90]) are faster if a permutation group is known to be solvable. On the other hand, proving nonsolvability can be expensive for higher degrees. Therefore GAP will automatically test a permutation group for solvability, only if the degree is not exceeding 100. (See also the `tryPcgs` component of `StabChainOptions` (43.8.2).) It is therefore beneficial to tell a group of larger degree, which is known to be solvable, that it is, using `SetIsSolvableGroup(`*G*`,true)`.

The second aspect concerns memory usage. A permutation on more than 65536 points requires 4 bytes per point for storing. So permutations on 256000 points require roughly 1MB of storage per permutation. Just storing the permutations required for a stabilizer chain might already go beyond the available memory, in particular if the base is not very short. In such a situation it can be useful, to replace the permutations by straight line program elements (see 37.9).

The following code gives an example of usage: We create a group of degree 231000. Using straight line program elements, one can compute a stabilizer chain in about 200 MB of memory.

Example

```
gap> Read("largeperms"); # read generators from file
gap> gens:=StraightLineProgGens(permutationlist);;
gap> g:=Group(gens);
<permutation group with 5 generators>
gap> # use random algorithm (faster, but result is monte carlo)
gap> StabChainOptions(g).random:=1;;
gap> Size(g); # enforce computation of a stabilizer chain
352969829814506607555723283375823418805608027364917220787701179633600 0
```

Without straight line program elements, the same calculation runs into memory problems after a while even with 512MB of workspace:

Example

```
gap> h:=Group(permutationlist);
<permutation group with 5 generators>
gap> StabChainOptions(h).random:=1;;
gap> Size(h);
exceeded the permitted memory (‘-o’ command line option) at
mlimit := 1; called from
SCRMakeStabStrong( S.stabilizer, [ g ], param, orbits, where, basesize,
 base, correct, missing, false ); called from
 SCRMakeStabStrong( S.stabilizer, [ g ], param, orbits, where, basesize,
...
```

The advantage in memory usage however is paid for in runtime: Comparisons of elements become much more expensive. One can avoid some of the related problems by registering a known base with the straight line program elements (see `StraightLineProgGens` (37.9.3)). In this case element comparison will only compare the images of the given base points. If we are planning to do extensive calculations with the group, it can even be worth to recreate it with straight line program elements knowing a previously computed base:

Example

```
gap> # get the base we computed already
gap> bas:=BaseStabChain(StabChainMutable(g));
[ 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55,
...
  2530, 2533, 2554, 2563, 2569 ]
gap> gens:=StraightLineProgGens(permutationlist,bas);;
gap> g:=Group(gens);;
gap> SetSize(g,
> 352969829814506607555723283375823418805608027364917220787701179633600 0);
gap> Random(g);; # enforce computation of a stabilizer chain
```

As we know already base and size, this second stabilizer chain calculation is much faster than the first one and takes less memory.

Chapter 44

Matrix Groups

Matrix groups are groups generated by invertible square matrices.

In the following example we temporarily increase the line length limit from its default value 80 to 83 in order to get a nicer output format.

Example

```
gap> m1 := [ [ Z(3)^0, Z(3)^0,   Z(3) ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ 0*Z(3),   Z(3), 0*Z(3) ] ];;
gap> m2 := [ [   Z(3),   Z(3), Z(3)^0 ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ Z(3)^0, 0*Z(3),   Z(3) ] ];;
gap> m := Group( m1, m2 );
Group(
[
  [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
      [ 0*Z(3), Z(3), 0*Z(3) ] ],
  [ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
      [ Z(3)^0, 0*Z(3), Z(3) ] ] ])
```

44.1 IsMatrixGroup (Filter)

For most operations, GAP only provides methods for finite matrix groups. Many calculations in finite matrix groups are done via so-called "nice monomorphisms" (see Section 40.5) that represent a faithful action on vectors.

44.1.1 IsMatrixGroup

▷ `IsMatrixGroup(`*grp*`)` (Category)

The category of matrix groups.

44.2 Attributes and Properties for Matrix Groups

44.2.1 DimensionOfMatrixGroup

▷ DimensionOfMatrixGroup(*mat-grp*) (attribute)

The dimension of the matrix group.

44.2.2 DefaultFieldOfMatrixGroup

▷ DefaultFieldOfMatrixGroup(*mat-grp*) (attribute)

Is a field containing all the matrix entries. It is not guaranteed to be the smallest field with this property.

44.2.3 FieldOfMatrixGroup

▷ FieldOfMatrixGroup(*matgrp*) (attribute)

The smallest field containing all the matrix entries of all elements of the matrix group *matgrp*. As the calculation of this can be hard, this should only be used if one *really* needs the smallest field, use DefaultFieldOfMatrixGroup (44.2.2) to get (for example) the characteristic.

Example

```
gap> DimensionOfMatrixGroup(m);
3
gap> DefaultFieldOfMatrixGroup(m);
GF(3)
```

44.2.4 TransposedMatrixGroup

▷ TransposedMatrixGroup(*matgrp*) (attribute)

returns the transpose of the matrix group *matgrp*. The transpose of the transpose of *matgrp* is identical to *matgrp*.

Example

```
gap> G := Group( [[0,-1],[1,0]] );
Group([ [ [ 0, -1 ], [ 1, 0 ] ] ])
gap> T := TransposedMatrixGroup( G );
Group([ [ [ 0, 1 ], [ -1, 0 ] ] ])
gap> IsIdenticalObj( G, TransposedMatrixGroup( T ) );
true
```

44.2.5 IsFFEMatrixGroup

▷ IsFFEMatrixGroup(*G*) (Category)

tests whether all matrices in *G* have finite field element entries.

44.3 Actions of Matrix Groups

The basic operations for groups are described in Chapter 41, special actions for *matrix* groups mentioned there are `OnLines` (41.2.12), `OnRight` (41.2.2), and `OnSubspacesByCanonicalBasis` (41.2.15).

For subtleties concerning multiplication from the left or from the right, see 44.7.

44.3.1 ProjectiveActionOnFullSpace

▷ `ProjectiveActionOnFullSpace(`*G, F, n*`)` (function)

Let *G* be a group of *n* by *n* matrices over a field contained in the finite field *F*. `ProjectiveActionOnFullSpace` returns the image of the projective action of *G* on the full row space F^n.

44.3.2 ProjectiveActionHomomorphismMatrixGroup

▷ `ProjectiveActionHomomorphismMatrixGroup(`*G*`)` (function)

returns an action homomorphism for a faithful projective action of *G* on the underlying vector space. (Note: The action is not necessarily on the full space, if a smaller subset can be found on which the action is faithful.)

44.3.3 BlowUpIsomorphism

▷ `BlowUpIsomorphism(`*matgrp, B*`)` (function)

For a matrix group *matgrp* and a basis *B* of a field extension L/K, say, such that the entries of all matrices in *matgrp* lie in L, `BlowUpIsomorphism` returns the isomorphism with source *matgrp* that is defined by mapping the matrix A to `BlownUpMat`(A, B), see `BlownUpMat` (24.13.3).

Example

```
gap> g:= GL(2,4);;
gap> B:= CanonicalBasis( GF(4) );;  BasisVectors( B );
[ Z(2)^0, Z(2^2) ]
gap> iso:= BlowUpIsomorphism( g, B );;
gap> Display( Image( iso, [ [ Z(4), Z(2) ], [ 0*Z(2), Z(4)^2 ] ] ) );
 . 1 1 .
 1 1 . 1
 . . 1 1
 . . 1 .
gap> img:= Image( iso, g );
<matrix group with 2 generators>
gap> Index( GL(4,2), img );
112
```

44.4 GL and SL

(See also section 50.2.)

44.4.1 IsGeneralLinearGroup

▷ IsGeneralLinearGroup(*grp*) (property)
▷ IsGL(*grp*) (property)

The General Linear group is the group of all invertible matrices over a ring. This property tests, whether a group is isomorphic to a General Linear group. (Note that currently only a few trivial methods are available for this operation. We hope to improve this in the future.)

44.4.2 IsNaturalGL

▷ IsNaturalGL(*matgrp*) (property)

This property tests, whether a matrix group is the General Linear group in the right dimension over the (smallest) ring which contains all entries of its elements. (Currently, only a trivial test that computes the order of the group is available.)

44.4.3 IsSpecialLinearGroup

▷ IsSpecialLinearGroup(*grp*) (property)
▷ IsSL(*grp*) (property)

The Special Linear group is the group of all invertible matrices over a ring, whose determinant is equal to 1. This property tests, whether a group is isomorphic to a Special Linear group. (Note that currently only a few trivial methods are available for this operation. We hope to improve this in the future.)

44.4.4 IsNaturalSL

▷ IsNaturalSL(*matgrp*) (property)

This property tests, whether a matrix group is the Special Linear group in the right dimension over the (smallest) ring which contains all entries of its elements. (Currently, only a trivial test that computes the order of the group is available.)

Example

```
gap> IsNaturalGL(m);
false
```

44.4.5 IsSubgroupSL

▷ IsSubgroupSL(*matgrp*) (property)

This property tests, whether a matrix group is a subgroup of the Special Linear group in the right dimension over the (smallest) ring which contains all entries of its elements.

44.5 Invariant Forms

44.5.1 InvariantBilinearForm

▷ `InvariantBilinearForm(`*matgrp*`)` (attribute)

This attribute describes a bilinear form that is invariant under the matrix group *matgrp*. The form is given by a record with the component `matrix` which is a matrix F such that for every generator g of *matgrp* the equation $g \cdot F \cdot g^{tr} = F$ holds.

44.5.2 IsFullSubgroupGLorSLRespectingBilinearForm

▷ `IsFullSubgroupGLorSLRespectingBilinearForm(`*matgrp*`)` (property)

This property tests, whether a matrix group *matgrp* is the full subgroup of GL or SL (the property `IsSubgroupSL` (44.4.5) determines which it is) respecting the form stored as the value of `InvariantBilinearForm` (44.5.1) for *matgrp*.

44.5.3 InvariantSesquilinearForm

▷ `InvariantSesquilinearForm(`*matgrp*`)` (attribute)

This attribute describes a sesquilinear form that is invariant under the matrix group *matgrp* over the field F with q^2 elements, say. The form is given by a record with the component `matrix` which is is a matrix M such that for every generator g of *matgrp* the equation $g \cdot M \cdot (g^{tr})^f = M$ holds, where f is the automorphism of F that raises each element to its q-th power. (f can be obtained as a power of the `FrobeniusAutomorphism` (59.4.1) value of F.)

44.5.4 IsFullSubgroupGLorSLRespectingSesquilinearForm

▷ `IsFullSubgroupGLorSLRespectingSesquilinearForm(`*matgrp*`)` (property)

This property tests, whether a matrix group *matgrp* is the full subgroup of GL or SL (the property `IsSubgroupSL` (44.4.5) determines which it is) respecting the form stored as the value of `InvariantSesquilinearForm` (44.5.3) for *matgrp*.

44.5.5 InvariantQuadraticForm

▷ `InvariantQuadraticForm(`*matgrp*`)` (attribute)

For a matrix group *matgrp*, `InvariantQuadraticForm` returns a record containing at least the component `matrix` whose value is a matrix Q. The quadratic form q on the natural vector space V on which *matgrp* acts is given by $q(v) = vQv^{tr}$, and the invariance under *matgrp* is given by the equation $q(v) = q(vM)$ for all $v \in V$ and M in *matgrp*. (Note that the invariance of q does *not* imply that the matrix Q is invariant under *matgrp*.)

q is defined relative to an invariant symmetric bilinear form f (see `InvariantBilinearForm` (44.5.1)), via the equation $q(\lambda x + \mu y) = \lambda^2 q(x) + \lambda\mu f(x,y) + \mu^2 q(y)$, see [CCN+85, Chapter 3.4]. If f is represented by the matrix F then this implies $F = Q + Q^{tr}$. In characteristic different from 2,

we have $q(x) = f(x,x)/2$, so Q can be chosen as the strictly upper triangular part of F plus half of the diagonal part of F. In characteristic 2, F does not determine Q but still Q can be chosen as an upper (or lower) triangular matrix.

Whenever the `InvariantQuadraticForm` value is set in a matrix group then also the `InvariantBilinearForm` (44.5.1) value can be accessed, and the two values are compatible in the above sense.

44.5.6 IsFullSubgroupGLorSLRespectingQuadraticForm

▷ `IsFullSubgroupGLorSLRespectingQuadraticForm(`*matgrp*`)` (property)

This property tests, whether the matrix group *matgrp* is the full subgroup of GL or SL (the property `IsSubgroupSL` (44.4.5) determines which it is) respecting the `InvariantQuadraticForm` (44.5.5) value of *matgrp*.

Example

```
gap> g:= Sp( 2, 3 );;
gap> m:= InvariantBilinearForm( g ).matrix;
[ [ 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3) ] ]
gap> [ 0, 1 ] * m * [ 1, -1 ];           # evaluate the bilinear form
Z(3)
gap> IsFullSubgroupGLorSLRespectingBilinearForm( g );
true
gap> g:= SU( 2, 4 );;
gap> m:= InvariantSesquilinearForm( g ).matrix;
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ]
gap> [ 0, 1 ] * m * [ 1, 1 ];            # evaluate the bilinear form
Z(2)^0
gap> IsFullSubgroupGLorSLRespectingSesquilinearForm( g );
true
gap> g:= GO( 1, 2, 3 );;
gap> m:= InvariantBilinearForm( g ).matrix;
[ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3) ] ]
gap> [ 0, 1 ] * m * [ 1, 1 ];            # evaluate the bilinear form
Z(3)^0
gap> q:= InvariantQuadraticForm( g ).matrix;
[ [ 0*Z(3), Z(3)^0 ], [ 0*Z(3), 0*Z(3) ] ]
gap> [ 0, 1 ] * q * [ 0, 1 ];            # evaluate the quadratic form
0*Z(3)
gap> IsFullSubgroupGLorSLRespectingQuadraticForm( g );
true
```

44.6 Matrix Groups in Characteristic 0

Most of the functions described in this and the following section have implementations which use functions from the GAP package Carat. If Carat is not installed or not compiled, no suitable methods are available.

44.6.1 IsCyclotomicMatrixGroup

▷ `IsCyclotomicMatrixGroup(G)` (Category)

tests whether all matrices in `G` have cyclotomic entries.

44.6.2 IsRationalMatrixGroup

▷ `IsRationalMatrixGroup(G)` (property)

tests whether all matrices in `G` have rational entries.

44.6.3 IsIntegerMatrixGroup

▷ `IsIntegerMatrixGroup(G)` (property)

tests whether all matrices in `G` have integer entries.

44.6.4 IsNaturalGLnZ

▷ `IsNaturalGLnZ(G)` (property)

tests whether `G` is $GL_n(\mathbb{Z})$ in its natural representation by $n \times n$ integer matrices. (The dimension n will be read off the generating matrices.)

Example

```
gap> IsNaturalGLnZ( GL( 2, Integers ) );
true
```

44.6.5 IsNaturalSLnZ

▷ `IsNaturalSLnZ(G)` (property)

tests whether `G` is $SL_n(\mathbb{Z})$ in its natural representation by $n \times n$ integer matrices. (The dimension n will be read off the generating matrices.)

Example

```
gap> IsNaturalSLnZ( SL( 2, Integers ) );
true
```

44.6.6 InvariantLattice

▷ `InvariantLattice(G)` (attribute)

returns a matrix B, whose rows form a basis of a $\mathbb{Z}$-lattice that is invariant under the rational matrix group `G` acting from the right. It returns `fail` if the group is not unimodular. The columns of the inverse of B span a $\mathbb{Z}$-lattice invariant under `G` acting from the left.

44.6.7 NormalizerInGLnZ

▷ `NormalizerInGLnZ(G)` (attribute)

is an attribute used to store the normalizer of `G` in $GL_n(\mathbb{Z})$, where `G` is an integer matrix group of dimension `n`. This attribute is used by `Normalizer( GL( n, Integers ), G )`.

44.6.8 CentralizerInGLnZ

▷ `CentralizerInGLnZ(G)` (attribute)

is an attribute used to store the centralizer of `G` in $GL_n(\mathbb{Z})$, where `G` is an integer matrix group of dimension `n`. This attribute is used by `Centralizer( GL( n, Integers ), G )`.

44.6.9 ZClassRepsQClass

▷ `ZClassRepsQClass(G)` (attribute)

The conjugacy class in $GL_n(\mathbb{Q})$ of the finite integer matrix group `G` splits into finitely many conjugacy classes in $GL_n(\mathbb{Z})$. `ZClassRepsQClass( G )` returns representative groups for these.

44.6.10 IsBravaisGroup

▷ `IsBravaisGroup(G)` (property)

test whether `G` coincides with its Bravais group (see `BravaisGroup` (44.6.11)).

44.6.11 BravaisGroup

▷ `BravaisGroup(G)` (attribute)

returns the Bravais group of a finite integer matrix group `G`. If C is the cone of positive definite quadratic forms Q invariant under $g \mapsto gQg^{tr}$ for all $g \in$ `G`, then the Bravais group of `G` is the maximal subgroup of $GL_n(\mathbb{Z})$ leaving the forms in that same cone invariant. Alternatively, the Bravais group of `G` can also be defined with respect to the action $g \mapsto g^{tr}Qg$ on positive definite quadratic forms Q. This latter definition is appropriate for groups `G` acting from the right on row vectors, whereas the former definition is appropriate for groups acting from the left on column vectors. Both definitions yield the same Bravais group.

44.6.12 BravaisSubgroups

▷ `BravaisSubgroups(G)` (attribute)

returns the subgroups of the Bravais group of `G`, which are themselves Bravais groups.

44.6.13 BravaisSupergroups

▷ `BravaisSupergroups(G)` (attribute)

returns the subgroups of $GL_n(\mathbb{Z})$ that contain the Bravais group of `G` and are Bravais groups themselves.

44.6.14 NormalizerInGLnZBravaisGroup

▷ `NormalizerInGLnZBravaisGroup(G)` (attribute)

returns the normalizer of the Bravais group of `G` in the appropriate $GL_n(\mathbb{Z})$.

44.7 Acting OnRight and OnLeft

In GAP, matrices by convention act on row vectors from the right, whereas in crystallography the convention is to act on column vectors from the left. The definition of certain algebraic objects important in crystallography implicitly depends on which action is assumed. This holds true in particular for quadratic forms invariant under a matrix group. In a similar way, the representation of affine crystallographic groups, as they are provided by the GAP package CrystGap, depends on which action is assumed. Crystallographers are used to the action from the left, whereas the action from the right is the natural one for GAP. For this reason, a number of functions which are important in crystallography, and whose result depends on which action is assumed, are provided in two versions, one for the usual action from the right, and one for the crystallographic action from the left.

For every such function, this fact is explicitly mentioned. The naming scheme is as follows: If `SomeThing` is such a function, there will be functions `SomeThingOnRight` and `SomeThingOnLeft`, assuming action from the right and from the left, respectively. In addition, there is a generic function `SomeThing`, which returns either the result of `SomeThingOnRight` or `SomeThingOnLeft`, depending on the global variable `CrystGroupDefaultAction` (44.7.1).

44.7.1 CrystGroupDefaultAction

▷ `CrystGroupDefaultAction` (global variable)

can have either of the two values `RightAction` and `LeftAction`. The initial value is `RightAction`. For functions which have variants OnRight and OnLeft, this variable determines which variant is returned by the generic form. The value of `CrystGroupDefaultAction` can be changed with with the function `SetCrystGroupDefaultAction` (44.7.2).

44.7.2 SetCrystGroupDefaultAction

▷ `SetCrystGroupDefaultAction(action)` (function)

allows one to set the value of the global variable `CrystGroupDefaultAction` (44.7.1). Only the arguments `RightAction` and `LeftAction` are allowed. Initially, the value of `CrystGroupDefaultAction` (44.7.1) is `RightAction`.

Chapter 45

Polycyclic Groups

A group `G` is *polycyclic* if there exists a subnormal series $G = C_1 > C_2 > \ldots > C_n > C_{n+1} = \{1\}$ with cyclic factors. Such a series is called *pc series* of `G`.

Every polycyclic group is solvable and every finite solvable group is polycyclic. However, there are infinite solvable groups which are not polycyclic.

In GAP there exists a large number of methods for polycyclic groups which are based upon the polycyclic structure of these groups. These methods are usually very efficient, especially for groups which are given by a pc-presentation (see chapter 46), and can be applied to many types of groups. Hence GAP tries to use them whenever possible, for example, for permutation groups and matrix groups over finite fields that are known to be polycyclic (the only exception is the representation as finitely presented group for which the polycyclic methods cannot be used in general).

At the current state of implementations the GAP library contains methods to compute with finite polycyclic groups, while the GAP package Polycyclic by Bettina Eick and Werner Nickel allows also computations with infinite polycyclic groups which are given by a pc-presentation.

45.1 Polycyclic Generating Systems

Let `G` be a polycyclic group with a pc series as above. A *polycyclic generating sequence* (*pcgs* for short) of `G` is a sequence $P := (g_1, \ldots, g_n)$ of elements of `G` such that $C_i = \langle C_{i+1}, g_i \rangle$ for $1 \leq i \leq n$. Note that each polycyclic group has a pcgs, but except for very small groups, a pcgs is not unique.

For each index i the subsequence of elements $(g_i, \ldots, g_n)$ forms a pcgs of the subgroup C_i. In particular, these *tails* generate the subgroups of the pc series and hence we say that the pc series is *determined* by P.

Let r_i be the index of C_{i+1} in C_i which is either a finite positive number or infinity. Then r_i is the order of $g_i C_{i+1}$ and we call the resulting list of indices the *relative orders* of the pcgs `P`.

Moreover, with respect to a given pcgs $(g_1, \ldots, g_n)$ each element `g` of `G` can be represented in a unique way as a product $g = g_1^{e_1} \cdot g_2^{e_2} \cdots g_n^{e_n}$ with exponents $e_i \in \{0, \ldots, r_i - 1\}$, if r_i is finite, and $e_i \in \mathbb{Z}$ otherwise. Words of this form are called *normal words* or *words in normal form*. Then the integer vector $[e_1, \ldots, e_n]$ is called the *exponent vector* of the element g. Furthermore, the smallest index k such that $e_k \neq 0$ is called the *depth* of `g` and e_k is the *leading exponent* of `g`.

For many applications we have to assume that each of the relative orders r_i is either a prime or infinity. This is equivalent to saying that there are no trivial factors in the pc series and the finite factors of the pc series are maximal refined. Then we obtain that r_i is the order of gC_{i+1} for all elements g in $C_i \setminus C_{i+1}$ and we call r_i the *relative order* of the element g.

45.2 Computing a Pcgs

Suppose a group *G* is given; for example, let *G* be a permutation or matrix group. Then we can ask GAP to compute a pcgs of this group. If *G* is not polycyclic, the result will be `fail`.

Note that these methods can only be applied if *G* is not given as finitely presented group. For finitely presented groups one can try to compute a pcgs via the polycyclic quotient methods, see 47.14.

Note also that a pcgs behaves like a list.

45.2.1 Pcgs

▷ `Pcgs(`*G*`)` (attribute)

returns a pcgs for the group *G*. If *grp* is not polycyclic it returns `fail` *and this result is not stored as attribute value*, in particular in this case the filter `HasPcgs` is *not* set for *G*!

45.2.2 IsPcgs

▷ `IsPcgs(`*obj*`)` (Category)

The category of pcgs.

Example

```
gap> G := Group((1,2,3,4),(1,2));;
gap> p := Pcgs(G);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IsPcgs( p );
true
gap> p[1];
(3,4)
gap> G := Group((1,2,3,4,5),(1,2));;
gap> Pcgs(G);
fail
```

45.2.3 CanEasilyComputePcgs

▷ `CanEasilyComputePcgs(`*grp*`)` (function)

This filter indicates whether it is possible to compute a pcgs for *grp* cheaply. Clearly, *grp* must be polycyclic in this case. However, not for every polycyclic group there is a method to compute a pcgs at low costs. This filter is used in the method selection mainly. Note that this filter may change its value from `false` to `true`.

Example

```
gap> G := Group( (1,2,3,4),(1,2) );
Group([ (1,2,3,4), (1,2) ])
gap> CanEasilyComputePcgs(G);
false
gap> Pcgs(G);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> CanEasilyComputePcgs(G);
true
```

45.3 Defining a Pcgs Yourself

In a number of situations it might be useful to supply a pcgs to a group.

Note that the elementary operations for such a pcgs might be rather inefficient, since GAP has to use generic methods in this case. It might be helpful to supply the relative orders of the self-defined pcgs as well by `SetRelativeOrder`. See also `IsPrimeOrdersPcgs` (45.4.3).

45.3.1 PcgsByPcSequence

▷ `PcgsByPcSequence(`*fam, pcs*`)` (operation)
▷ `PcgsByPcSequenceNC(`*fam, pcs*`)` (operation)

constructs a pcgs for the elements family *fam* from the elements in the list *pcs*. The elements must lie in the family *fam*. `PcgsByPcSequence` and its `NC` variant will always create a new pcgs which is not induced by any other pcgs (cf. `InducedPcgsByPcSequence` (45.7.2)).

Example

```
gap> fam := FamilyObj( (1,2) );; # the family of permutations
gap> p := PcgsByPcSequence( fam, [(1,2),(1,2,3)] );
Pcgs([ (1,2), (1,2,3) ])
gap> RelativeOrders(p);
[ 2, 3 ]
gap> ExponentsOfPcElement( p, (1,3,2) );
[ 0, 2 ]
```

45.4 Elementary Operations for a Pcgs

45.4.1 RelativeOrders

▷ `RelativeOrders(`*pcgs*`)` (attribute)

returns the list of relative orders of the pcgs *pcgs*.

45.4.2 IsFiniteOrdersPcgs

▷ `IsFiniteOrdersPcgs(`*pcgs*`)` (property)

tests whether the relative orders of *pcgs* are all finite.

45.4.3 IsPrimeOrdersPcgs

▷ `IsPrimeOrdersPcgs(`*pcgs*`)` (property)

tests whether the relative orders of *pcgs* are prime numbers. Many algorithms require a pcgs to have this property. The operation `IsomorphismRefinedPcGroup` (46.4.8) can be of help here.

45.4.4 PcSeries

▷ PcSeries(*pcgs*) (attribute)

returns the subnormal series determined by *pcgs*.

45.4.5 GroupOfPcgs

▷ GroupOfPcgs(*pcgs*) (attribute)

The group generated by *pcgs*.

45.4.6 OneOfPcgs

▷ OneOfPcgs(*pcgs*) (attribute)

The identity of the group generated by *pcgs*.

Example

```
gap> G := Group( (1,2,3,4),(1,2) );; p := Pcgs(G);;
gap> RelativeOrders(p);
[ 2, 3, 2, 2 ]
gap> IsFiniteOrdersPcgs(p);
true
gap> IsPrimeOrdersPcgs(p);
true
gap> PcSeries(p);
[ Group([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group([ (1,3)(2,4) ]), Group(())
 ]
```

45.5 Elementary Operations for a Pcgs and an Element

45.5.1 RelativeOrderOfPcElement

▷ RelativeOrderOfPcElement(*pcgs, elm*) (operation)

The relative order of *elm* with respect to the prime order pcgs *pcgs*.

45.5.2 ExponentOfPcElement

▷ ExponentOfPcElement(*pcgs, elm, pos*) (operation)

returns the *pos*-th exponent of *elm* with respect to *pcgs*.

45.5.3 ExponentsOfPcElement

▷ ExponentsOfPcElement(*pcgs, elm[, posran]*) (operation)

returns the exponents of *elm* with respect to *pcgs*. The three argument version returns the exponents in the positions given in *posran*.

45.5.4 DepthOfPcElement

▷ DepthOfPcElement(*pcgs, elm*) (operation)

returns the depth of the element *elm* with respect to *pcgs*.

45.5.5 LeadingExponentOfPcElement

▷ LeadingExponentOfPcElement(*pcgs, elm*) (operation)

returns the leading exponent of *elm* with respect to *pcgs*.

45.5.6 PcElementByExponents

▷ PcElementByExponents(*pcgs, list*) (operation)
▷ PcElementByExponentsNC(*pcgs[, basisind], list*) (operation)

returns the element corresponding to the exponent vector *list* with respect to *pcgs*. The exponents in *list* must be in the range of permissible exponents for *pcgs*. *It is not guaranteed that PcElementByExponents will reduce the exponents modulo the relative orders.* (You should use the operation LinearCombinationPcgs (45.5.7) for this purpose.) The NC version does not check that the lengths of the lists fit together and does not check the exponent range.

The three argument version gives exponents only w.r.t. the generators in *pcgs* indexed by *basisind*.

45.5.7 LinearCombinationPcgs

▷ LinearCombinationPcgs(*pcgs, list[, one]*) (operation)

returns the product $\prod_i pcgs[i]^{list[i]}$. In contrast to PcElementByExponents (45.5.6) this permits negative exponents. *pcgs* might be a list of group elements. In this case, an appropriate identity element *one* must be given. *list* can be empty.

Example

```
gap> G := Group( (1,2,3,4),(1,2) );; P := Pcgs(G);;
gap> g := PcElementByExponents(P, [0,1,1,1]);
(1,2,3)
gap> ExponentsOfPcElement(P, g);
[ 0, 1, 1, 1 ]
```

45.5.8 SiftedPcElement

▷ SiftedPcElement(*pcgs, elm*) (operation)

sifts *elm* through *pcgs*, reducing it if the depth is the same as the depth of one of the generators in *pcgs*. Thus the identity is returned if *elm* lies in the group generated by *pcgs*. *pcgs* must be an induced pcgs (see section 45.7) and *elm* must lie in the span of the parent of *pcgs*.

45.5.9 CanonicalPcElement

▷ `CanonicalPcElement(ipcgs, elm)` (operation)

reduces *elm* at the induces pcgs *ipcgs* such that the exponents of the reduced result *r* are zero at the depths for which there are generators in *ipcgs*. Elements, whose quotient lies in the group generated by *ipcgs* yield the same canonical element.

45.5.10 ReducedPcElement

▷ `ReducedPcElement(pcgs, x, y)` (operation)

reduces the element *x* by dividing off (from the left) a power of *y* such that the leading coefficient of the result with respect to *pcgs* becomes zero. The elements *x* and *y* therefore have to have the same depth.

45.5.11 CleanedTailPcElement

▷ `CleanedTailPcElement(pcgs, elm, dep)` (operation)

returns an element in the span of *pcgs* whose exponents for indices 1 to $dep - 1$ with respect to *pcgs* are the same as those of *elm*, the remaining exponents are undefined. This can be used to obtain more "simple" elements if only representatives in a factor are required, see 45.9.

The difference to `HeadPcElementByNumber` (45.5.12) is that this function is guaranteed to zero out trailing coefficients while `CleanedTailPcElement` will only do this if it can be done cheaply.

45.5.12 HeadPcElementByNumber

▷ `HeadPcElementByNumber(pcgs, elm, dep)` (operation)

returns an element in the span of *pcgs* whose exponents for indices 1 to $dep-1$ with respect to *pcgs* are the same as those of *elm*, the remaining exponents are zero. This can be used to obtain more "simple" elements if only representatives in a factor are required.

45.6 Exponents of Special Products

There are certain products of elements whose exponents are used often within algorithms, and which might be obtained more easily than by computing the product first and to obtain its exponents afterwards. The operations in this section provide a way to obtain such exponent vectors directly.

(The circumstances under which these operations give a speedup depend very much on the pcgs and the representation of elements that is used. So the following operations are not guaranteed to give a speedup in every case, however the default methods are not slower than to compute the exponents of a product and thus these operations should *always* be used if applicable.)

The second class are exponents of products of the generators which make up the pcgs. If the pcgs used is a family pcgs (see `FamilyPcgs` (46.1.1)) then these exponents can be looked up and do not need to be computed.

45.6.1 ExponentsConjugateLayer

▷ `ExponentsConjugateLayer(mpcgs, elm, e)` (operation)

Computes the exponents of `elm^e` with respect to `mpcgs`; `elm` must be in the span of `mpcgs`, `e` a pc element in the span of the parent pcgs of `mpcgs` and `mpcgs` must be the modulo pcgs for an abelian layer. (This is the usual case when acting on a chief factor). In this case if `mpcgs` is induced by the family pcgs (see section 45.7), the exponents can be computed directly by looking up exponents without having to compute in the group and having to collect a potential tail.

45.6.2 ExponentsOfRelativePower

▷ `ExponentsOfRelativePower(pcgs, i)` (operation)

For $p =$ `pcgs[i]` this function returns the exponent vector with respect to `pcgs` of the element p^e where e is the relative order of `p` in `pcgs`. For the family pcgs or pcgs induced by it (see section 45.7), this might be faster than computing the element and computing its exponent vector.

45.6.3 ExponentsOfConjugate

▷ `ExponentsOfConjugate(pcgs, i, j)` (operation)

returns the exponents of `pcgs[i]^pcgs[j]` with respect to `pcgs`. For the family pcgs or pcgs induced by it (see section 45.7), this might be faster than computing the element and computing its exponent vector.

45.6.4 ExponentsOfCommutator

▷ `ExponentsOfCommutator(pcgs, i, j)` (operation)

returns the exponents of the commutator `Comm( pcgs[i],pcgs[j] )` with respect to `pcgs`. For the family pcgs or pcgs induced by it, (see section 45.7), this might be faster than computing the element and computing its exponent vector.

45.7 Subgroups of Polycyclic Groups - Induced Pcgs

Let `U` be a subgroup of `G` and let `P` be a pcgs of `G` as above such that `P` determines the subnormal series $G = C_1 > \ldots > C_{n+1} = \{1\}$. Then the series of subgroups $U \cap C_i$ is a subnormal series of `U` with cyclic or trivial factors. Hence, if we choose an element $u_{i_j} \in (U \cap C_{i_j}) \setminus (U \cap C_{i_j+1})$ whenever this factor is non-trivial, then we obtain a pcgs $Q = (u_{i_1}, \ldots, u_{i_m})$ of U. We say that Q is an *induced pcgs* with respect to `P`. The pcgs `P` is the *parent pcgs* to the induced pcgs `Q`.

Note that the pcgs Q is induced with respect to `P` if and only if the matrix of exponent vectors of the elements u_{i_j} with respect to `P` is in upper triangular form. Thus Q is not unique in general.

In particular, the elements of an induced pcgs do *not necessarily* have leading coefficient 1 relative to the inducing pcgs. The attribute `LeadCoeffsIGS` (45.7.7) holds the leading coefficients in case they have to be renormed in an algorithm.

Each induced pcgs is a pcgs and hence allows all elementary operations for pcgs. On the other hand each pcgs could be transformed into an induced pcgs for the group defined by the pcgs, but note that an arbitrary pcgs is in general not an induced pcgs for technical reasons.

An induced pcgs is "compatible" with its parent, see `ParentPcgs` (45.7.3).

In [LNS84] a "non-commutative Gauss" algorithm is described to compute an induced pcgs of a subgroup *U* from a generating set of *U*. For calling this in GAP, see 45.7.4 to 45.7.8.

To create a subgroup generated by an induced pcgs such that the induced pcgs gets stored automatically, use `SubgroupByPcgs` (45.7.9).

45.7.1 IsInducedPcgs

▷ `IsInducedPcgs(`*pcgs*`)` (Category)

The category of induced pcgs. This a subcategory of pcgs.

45.7.2 InducedPcgsByPcSequence

▷ `InducedPcgsByPcSequence(`*pcgs, pcs*`)` (operation)
▷ `InducedPcgsByPcSequenceNC(`*pcgs, pcs[, depths]*`)` (operation)

If *pcs* is a list of elements that form an induced pcgs with respect to *pcgs* this operation returns an induced pcgs with these elements.

In the third version, the depths of *pcs* with respect to *pcgs* can be given (they are computed anew otherwise).

45.7.3 ParentPcgs

▷ `ParentPcgs(`*pcgs*`)` (attribute)

returns the pcgs by which *pcgs* was induced. If *pcgs* was not induced, it simply returns *pcgs*.

Example

```
gap> G := Group( (1,2,3,4),(1,2) );;
gap> P := Pcgs(G);;
gap> K := InducedPcgsByPcSequence( P, [(1,2,3,4),(1,3)(2,4)] );
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> ParentPcgs( K );
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IsInducedPcgs( K );
true
```

45.7.4 InducedPcgs

▷ `InducedPcgs(`*pcgs, grp*`)` (operation)

computes a pcgs for *grp* which is induced by *pcgs*. If *pcgs* has a parent pcgs, then the result is induced with respect to this parent pcgs.

`InducedPcgs` is a wrapper function only. Therefore, methods for computing computing an induced pcgs should be installed for the operation `InducedPcgsOp`.

45.7.5 InducedPcgsByGenerators

▷ InducedPcgsByGenerators(*pcgs*, *gens*) (operation)
▷ InducedPcgsByGeneratorsNC(*pcgs*, *gens*) (operation)

returns an induced pcgs with respect to *pcgs* for the subgroup generated by *gens*.

45.7.6 InducedPcgsByPcSequenceAndGenerators

▷ InducedPcgsByPcSequenceAndGenerators(*pcgs*, *ind*, *gens*) (operation)

returns an induced pcgs with respect to *pcgs* of the subgroup generated by *ind* and *gens*. Here *ind* must be an induced pcgs with respect to *pcgs* (or a list of group elements that form such an igs) and it will be used as initial sequence for the computation.

Example

```
gap> G := Group( (1,2,3,4),(1,2) );;  P := Pcgs(G);;
gap> I := InducedPcgsByGenerators( P, [(1,2,3,4)] );
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> J := InducedPcgsByPcSequenceAndGenerators( P, I, [(1,2)] );
Pcgs([ (1,2,3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
```

45.7.7 LeadCoeffsIGS

▷ LeadCoeffsIGS(*igs*) (attribute)

This attribute is used to store leading coefficients with respect to the parent pcgs. the *i*-th entry –if bound– is the leading exponent of the element of *igs* that has depth *i* in the parent. (It cannot be assigned to a component in the object created by InducedPcgsByPcSequenceNC (45.7.2) as the permutation group methods call it from within the postprocessing, before this postprocessing however no coefficients may be computed.)

45.7.8 ExtendedPcgs

▷ ExtendedPcgs(*N*, *gens*) (operation)

extends the pcgs *N* (induced w.r.t. *home*) to a new induced pcgs by prepending *gens*. No checks are performed that this really yields an induced pcgs.

45.7.9 SubgroupByPcgs

▷ SubgroupByPcgs(*G*, *pcgs*) (operation)

returns a subgroup of *G* generated by the elements of *pcgs*.

45.8 Subgroups of Polycyclic Groups – Canonical Pcgs

The induced pcgs *Q* of *U* is called *canonical* if the matrix of exponent vectors contains normed vectors only and above each leading entry in the matrix there are 0's only. The canonical pcgs of *U* with respect to *P* is unique and hence such pcgs can be used to compare subgroups.

45.8.1 IsCanonicalPcgs

▷ IsCanonicalPcgs(*pcgs*) (property)

An induced pcgs is canonical if the matrix of the exponent vectors of the elements of *pcgs* with respect to the ParentPcgs (45.7.3) value of *pcgs* is in Hermite normal form (see [LNS84]). While a subgroup can have various induced pcgs with respect to a parent pcgs a canonical pcgs is unique.

45.8.2 CanonicalPcgs

▷ CanonicalPcgs(*pcgs*) (attribute)

returns the canonical pcgs corresponding to the induced pcgs *pcgs*.

Example

```
gap> G := Group((1,2,3,4),(5,6,7));
Group([ (1,2,3,4), (5,6,7) ])
gap> P := Pcgs(G);
Pcgs([ (5,6,7), (1,2,3,4), (1,3)(2,4) ])
gap> I := InducedPcgsByPcSequence(P, [(5,6,7)*(1,3)(2,4),(1,3)(2,4)] );
Pcgs([ (1,3)(2,4)(5,6,7), (1,3)(2,4) ])
gap> CanonicalPcgs(I);
Pcgs([ (5,6,7), (1,3)(2,4) ])
```

45.9 Factor Groups of Polycyclic Groups – Modulo Pcgs

Let *N* be a normal subgroup of *G* such that *G/N* is polycyclic with pcgs $(h_1N,\ldots,h_rN)$. Then we call the sequence of preimages $(h_1,\ldots h_r)$ a *modulo pcgs* of *G/N*. *G* is called the *numerator* of the modulo pcgs and *N* is the *denominator* of the modulo pcgs.

Modulo pcgs are often used to facilitate efficient computations with factor groups, since they allow computations with factor groups without formally defining the factor group at all.

All elementary operations of pcgs, see Sections 45.4 and 45.5, apply to modulo pcgs as well. However, it is in general not possible to compute induced pcgs with respect to a modulo pcgs.

Two more elementary operations for modulo pcgs are NumeratorOfModuloPcgs (45.9.3) and DenominatorOfModuloPcgs (45.9.4).

45.9.1 ModuloPcgs

▷ ModuloPcgs(*G, N*) (operation)

returns a modulo pcgs for the factor G/N which must be solvable, which *N* may be insolvable. ModuloPcgs will return *a* pcgs for the factor, there is no guarantee that it will be "compatible" with any other pcgs. If this is required, the mod operator must be used on induced pcgs, see \mod (45.9.5).

45.9.2 IsModuloPcgs

▷ IsModuloPcgs(*obj*) (Category)

The category of modulo pcgs. Note that each pcgs is a modulo pcgs for the trivial subgroup.

45.9.3 NumeratorOfModuloPcgs

▷ NumeratorOfModuloPcgs(*pcgs*) (attribute)

returns a generating set for the numerator of the modulo pcgs *pcgs*.

45.9.4 DenominatorOfModuloPcgs

▷ DenominatorOfModuloPcgs(*pcgs*) (attribute)

returns a generating set for the denominator of the modulo pcgs *pcgs*.

Example

```
gap> G := Group( (1,2,3,4,5),(1,2) );
Group([ (1,2,3,4,5), (1,2) ])
gap> P := ModuloPcgs(G, DerivedSubgroup(G) );
Pcgs([ (4,5) ])
gap> NumeratorOfModuloPcgs(P);
[ (1,2,3,4,5), (1,2) ]
gap> DenominatorOfModuloPcgs(P);
[ (1,3,2), (2,4,3), (2,3)(4,5) ]
gap> RelativeOrders(P);
[ 2 ]
gap> ExponentsOfPcElement( P, (1,2,3,4,5) );
[ 0 ]
gap> ExponentsOfPcElement( P, (4,5) );
[ 1 ]
```

45.9.5 \mod (for two pcgs)

▷ \mod(*P*, *I*) (method)

Modulo Pcgs can also be built from compatible induced pcgs. Let G be a group with pcgs P and let I be an induced pcgs of a normal subgroup N of G. (Respectively: P and I are both induced with respect to the *same* Pcgs.) Then we can compute a modulo pcgs of G mod N by

P mod *I*

Note that in this case we obtain the advantage that the values of NumeratorOfModuloPcgs (45.9.3) and DenominatorOfModuloPcgs (45.9.4) are just P and I, respectively, and hence are unique.

The resulting modulo pcgs will consist of a subset of P and will be "compatible" with P (or its parent).

Example

```
gap> G := Group((1,2,3,4));;
gap> P := Pcgs(G);
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> I := InducedPcgsByGenerators(P, [(1,3)(2,4)]);
Pcgs([ (1,3)(2,4) ])
gap> M := P mod I;
[ (1,2,3,4) ]
gap> NumeratorOfModuloPcgs(M);
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
```

```
gap> DenominatorOfModuloPcgs(M);
Pcgs([ (1,3)(2,4) ])
```

45.9.6 CorrespondingGeneratorsByModuloPcgs

▷ `CorrespondingGeneratorsByModuloPcgs(`*mpcgs, imgs*`)` (operation)

Let *mpcgs* be a modulo pcgs for a factor of a group G and let U be a subgroup of G generated by *imgs* such that U covers the factor for the modulo pcgs. Then this function computes elements in U corresponding to the generators of the modulo pcgs.

Note that the computation of induced generating sets is not possible for some modulo pcgs.

45.9.7 CanonicalPcgsByGeneratorsWithImages

▷ `CanonicalPcgsByGeneratorsWithImages(`*pcgs, gens, imgs*`)` (operation)

computes a canonical, *pcgs*-induced pcgs for the span of *gens* and simultaneously does the same transformations on *imgs*, preserving thus a correspondence between *gens* and *imgs*. This operation is used to represent homomorphisms from a pc group.

45.10 Factor Groups of Polycyclic Groups in their Own Representation

If substantial calculations are done in a factor it might be worth still to construct the factor group in its own representation (for example by calling `PcGroupWithPcgs` (46.5.1) on a modulo pcgs.

The following functions are intended for working with factor groups obtained by factoring out the tail of a pcgs. They provide a way to map elements or induced pcgs quickly in the factor (respectively to take preimages) without the need to construct a homomorphism.

The setup is always a pcgs *pcgs* of *G* and a pcgs *fpcgs* of a factor group $H = G/N$ which corresponds to a head of *pcgs*.

No tests for validity of the input are performed.

45.10.1 ProjectedPcElement

▷ `ProjectedPcElement(`*pcgs, fpcgs, elm*`)` (function)

returns the image in *H* of an element *elm* of *G*.

45.10.2 ProjectedInducedPcgs

▷ `ProjectedInducedPcgs(`*pcgs, fpcgs, ipcgs*`)` (function)

ipcgs must be an induced pcgs with respect to *pcgs*. This operation returns an induced pcgs with respect to *fpcgs* consisting of the nontrivial images of *ipcgs*.

45.10.3 LiftedPcElement

▷ LiftedPcElement(*pcgs, fpcgs, elm*) (function)

returns a preimage in *G* of an element *elm* of *H*.

45.10.4 LiftedInducedPcgs

▷ LiftedInducedPcgs(*pcgs, fpcgs, ipcgs, ker*) (function)

ipcgs must be an induced pcgs with respect to *fpcgs*. This operation returns an induced pcgs with respect to *pcgs* consisting of the preimages of *ipcgs*, appended by the elements in *ker* (assuming there is a bijection of *pcgs* mod *ker* to *fpcgs*). *ker* might be a simple element list.

45.11 Pcgs and Normal Series

By definition, a pcgs determines a pc series of its underlying group. However, in many applications it will be necessary that this pc series refines a normal series with certain properties; for example, a normal series with abelian factors.

There are functions in GAP to compute a pcgs through a normal series with elementary abelian factors, a central series or the lower p-central series. See also Section 45.13 for a more explicit possibility.

45.11.1 IsPcgsElementaryAbelianSeries

▷ IsPcgsElementaryAbelianSeries(*pcgs*) (property)

returns true if the pcgs *pcgs* refines an elementary abelian series. IndicesEANormalSteps (45.11.3) then gives the indices in the Pcgs, at which the subgroups of this series start.

45.11.2 PcgsElementaryAbelianSeries (for a group)

▷ PcgsElementaryAbelianSeries(*G*) (attribute)
▷ PcgsElementaryAbelianSeries(*list*) (attribute)

computes a pcgs for *G* that refines an elementary abelian series. IndicesEANormalSteps (45.11.3) gives the indices in the pcgs, at which the normal subgroups of this series start. The second variant returns a pcgs that runs through the normal subgroups in the list *list*.

45.11.3 IndicesEANormalSteps

▷ IndicesEANormalSteps(*pcgs*) (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with elementary abelian factors (for example from calling PcgsElementaryAbelianSeries (45.11.2)) Then IndicesEANormalSteps returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroup of *G*. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of

`G`. The list always starts with 1 and ends with $n+1$. (These indices form *one* series with elementary abelian subfactors, not necessarily the most refined one.)

The attribute `EANormalSeriesByPcgs` (45.11.4) returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and `IndicesEANormalSteps` is not guaranteed to work in this situation.

Typically, `IndicesEANormalSteps` is set by `PcgsElementaryAbelianSeries` (45.11.2).

45.11.4 EANormalSeriesByPcgs

▷ `EANormalSeriesByPcgs(pcgs)` (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with elementary abelian factors (for example from calling `PcgsElementaryAbelianSeries` (45.11.2)). This attribute returns the actual series of normal subgroups, corresponding to `IndicesEANormalSteps` (45.11.3).

45.11.5 IsPcgsCentralSeries

▷ `IsPcgsCentralSeries(pcgs)` (property)

returns `true` if the pcgs *pcgs* refines an central elementary abelian series. `IndicesCentralNormalSteps` (45.11.7) then gives the indices in the pcgs, at which the subgroups of this series start.

45.11.6 PcgsCentralSeries

▷ `PcgsCentralSeries(G)` (attribute)

computes a pcgs for `G` that refines a central elementary abelian series. `IndicesCentralNormalSteps` (45.11.7) gives the indices in the pcgs, at which the normal subgroups of this series start.

45.11.7 IndicesCentralNormalSteps

▷ `IndicesCentralNormalSteps(pcgs)` (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with central elementary abelian factors (for example from calling `PcgsCentralSeries` (45.11.6)). Then `IndicesCentralNormalSteps` returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroups of `G`. If i is an element of this list, $(g_i,\ldots,g_n)$ is a normal subgroup of `G`. The list always starts with 1 and ends with $n+1$. (These indices form *one* series with central elementary abelian subfactors, not necessarily the most refined one.)

The attribute `CentralNormalSeriesByPcgs` (45.11.8) returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and `IndicesCentralNormalSteps` is not guaranteed to work in this situation.

Typically, `IndicesCentralNormalSteps` is set by `PcgsCentralSeries` (45.11.6).

45.11.8 CentralNormalSeriesByPcgs

▷ CentralNormalSeriesByPcgs(*pcgs*) (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with central elementary abelian factors (for example from calling PcgsCentralSeries (45.11.6)). This attribute returns the actual series of normal subgroups, corresponding to IndicesCentralNormalSteps (45.11.7).

45.11.9 IsPcgsPCentralSeriesPGroup

▷ IsPcgsPCentralSeriesPGroup(*pcgs*) (property)

returns true if the pcgs *pcgs* refines a p-central elementary abelian series for a p-group. IndicesPCentralNormalStepsPGroup (45.11.11) then gives the indices in the pcgs, at which the subgroups of this series start.

45.11.10 PcgsPCentralSeriesPGroup

▷ PcgsPCentralSeriesPGroup(*G*) (attribute)

computes a pcgs for the p-group *G* that refines a p-central elementary abelian series. IndicesPCentralNormalStepsPGroup (45.11.11) gives the indices in the pcgs, at which the normal subgroups of this series start.

45.11.11 IndicesPCentralNormalStepsPGroup

▷ IndicesPCentralNormalStepsPGroup(*pcgs*) (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with p-central elementary abelian factors (for example from calling PcgsPCentralSeriesPGroup (45.11.10)). Then IndicesPCentralNormalStepsPGroup returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroups of *G*. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of *G*. The list always starts with 1 and ends with $n+1$. (These indices form *one* series with central elementary abelian subfactors, not necessarily the most refined one.)

The attribute PCentralNormalSeriesByPcgsPGroup (45.11.12) returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and IndicesPCentralNormalStepsPGroup is not guaranteed to work in this situation.

Typically, IndicesPCentralNormalStepsPGroup is set by PcgsPCentralSeriesPGroup (45.11.10).

45.11.12 PCentralNormalSeriesByPcgsPGroup

▷ PCentralNormalSeriesByPcgsPGroup(*pcgs*) (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with p-central elementary abelian factors (for example from calling PcgsPCentralSeriesPGroup

(45.11.10)). This attribute returns the actual series of normal subgroups, corresponding to IndicesPCentralNormalStepsPGroup (45.11.11).

45.11.13 IsPcgsChiefSeries

▷ IsPcgsChiefSeries(*pcgs*) (property)

returns true if the pcgs *pcgs* refines a chief series. IndicesChiefNormalSteps (45.11.15) then gives the indices in the pcgs, at which the subgroups of this series start.

45.11.14 PcgsChiefSeries

▷ PcgsChiefSeries(*G*) (attribute)

computes a pcgs for *G* that refines a chief series. IndicesChiefNormalSteps (45.11.15) gives the indices in the pcgs, at which the normal subgroups of this series start.

45.11.15 IndicesChiefNormalSteps

▷ IndicesChiefNormalSteps(*pcgs*) (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a chief series for example from calling PcgsChiefSeries (45.11.14)). Then IndicesChiefNormalSteps returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroups of *G*. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of *G*. The list always starts with 1 and ends with $n+1$. (These indices form *one* series with elementary abelian subfactors, not necessarily the most refined one.)

The attribute ChiefNormalSeriesByPcgs (45.11.16) returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and IndicesChiefNormalSteps is not guaranteed to work in this situation.

Typically, IndicesChiefNormalSteps is set by PcgsChiefSeries (45.11.14).

45.11.16 ChiefNormalSeriesByPcgs

▷ ChiefNormalSeriesByPcgs(*pcgs*) (attribute)

Let *pcgs* be a pcgs obtained as corresponding to a chief series (for example from calling PcgsChiefSeries (45.11.14)). This attribute returns the actual series of normal subgroups, corresponding to IndicesChiefNormalSteps (45.11.15).

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> p:=PcgsElementaryAbelianSeries(g);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IndicesEANormalSteps(p);
[ 1, 2, 3, 5 ]
gap> g:=Group((1,2,3,4),(1,5)(2,6)(3,7)(4,8));;
gap> p:=PcgsCentralSeries(g);
Pcgs([ (1,5)(2,6)(3,7)(4,8), (5,6,7,8), (5,7)(6,8),
  (1,4,3,2)(5,6,7,8), (1,3)(2,4)(5,7)(6,8) ])
gap> IndicesCentralNormalSteps(p);
```

```
[ 1, 2, 4, 5, 6 ]
gap> q:=PcgsPCentralSeriesPGroup(g);
Pcgs([ (1,5)(2,6)(3,7)(4,8), (5,6,7,8), (5,7)(6,8),
  (1,4,3,2)(5,6,7,8), (1,3)(2,4)(5,7)(6,8) ])
gap> IndicesPCentralNormalStepsPGroup(q);
[ 1, 3, 5, 6 ]
```

45.11.17 IndicesNormalSteps

▷ `IndicesNormalSteps(`*pcgs*`)` (attribute)

returns the indices of *all* steps in the pc series, which are normal in the group defined by the pcgs.

(In general, this function yields a slower performance than the more specialized index functions for elementary abelian series etc.)

45.11.18 NormalSeriesByPcgs

▷ `NormalSeriesByPcgs(`*pcgs*`)` (attribute)

returns the subgroups the pc series, which are normal in the group defined by the pcgs.

(In general, this function yields a slower performance than the more specialized index functions for elementary abelian series etc.)

45.12 Sum and Intersection of Pcgs

45.12.1 SumFactorizationFunctionPcgs

▷ `SumFactorizationFunctionPcgs(`*parentpcgs, n, u, kerpcgs*`)` (operation)

computes the sum and intersection of the lists *n* and *u* whose elements form modulo pcgs induced by *parentpcgs* for two subgroups modulo a kernel given by *kerpcgs*. If *kerpcgs* is a tail if the *parent-pcgs* it is sufficient to give the starting depth, this can be more efficient than to construct an explicit pcgs. The factor group modulo *kerpcgs* generated by *n* must be elementary abelian and normal under *u*.

The function returns a record with components

`sum` elements that form a modulo pcgs for the span of both subgroups.

`intersection`
elements that form a modulo pcgs for the intersection of both subgroups.

`factorization`
a function that returns for an element *x* in the span of `sum` a record with components `u` and `n` that give its decomposition.

The record components `sum` and `intersection` are *not* pcgs but only lists of pc elements (to avoid unnecessary creation of induced pcgs).

45.13 Special Pcgs

In short, a special pcgs is a pcgs which has particularly nice properties, for example it always refines an elementary abelian series, for p-groups it even refines a central series. These nice properties permit particularly efficient algorithms.

Let `G` be a finite polycyclic group. A *special pcgs* of `G` is a pcgs which is closely related to a Hall system and the maximal subgroups of `G`. These pcgs have been introduced by C. R. Leedham-Green who also gave an algorithm to compute them. Improvements to this algorithm are due to Bettina Eick. For a more detailed account of their definition the reader is referred to [Eic97]

To introduce the definition of special pcgs we first need to define the *LG-series* and *head complements* of a finite polycyclic group `G`. Let $G = G_1 > G_2 > \ldots G_m > G_{m+1} = \{1\}$ be the lower nilpotent series of G; that is, G_i is the smallest normal subgroup of G_{i-1} with nilpotent factor. To obtain the LG-series of `G` we need to refine this series. Thus consider a factor $F_i := G_i/G_{i+1}$. Since F_i is finite nilpotent, it is a direct product of its Sylow subgroups, say $F_i = P_{i,1} \cdots P_{i,r_i}$. For each Sylow p_j-subgroup $P_{i,j}$ we can consider its lower p_j-central series. To obtain a characteristic central series with elementary abelian factors of F_i we loop over its Sylow subgroups. Each time we consider $P_{i,j}$ in this process we take the next step of its lower p_j-central series into the series of F_i. If there is no next step, then we just skip the consideration of $P_{i,j}$. Note that the second term of the lower p-central series of a p-group is in fact its Frattini subgroup. Thus the Frattini subgroup of F_i is contained in the computed series of this group. We denote the Frattini subgroup of $F_i = G_i/G_{i+1}$ by G_i^*/G_{i+1}.

The factors G_i/G_i^* are called the heads of G, while the (possibly trivial) factors G_i^*/G_{i+1} are the tails of G. A head complement of G is a subgroup U of G such that U/G_i^* is a complement to the head G_i/G_i^* in G/G_i^* for some i.

Now we are able to define a special pcgs of `G`. It is a pcgs of `G` with three additional properties. First, the pc series determined by the pcgs refines the LG-series of `G`. Second, a special pcgs *exhibits* a Hall system of the group `G`; that is, for each set of primes π the elements of the pcgs with relative order in π form a pcgs of a Hall π-subgroup in a Hall system of `G`. Third, a special pcgs exhibits a head complement for each head of `G`.

To record information about the LG-series with the special pcgs we define the *LGWeights* of the special pcgs. These weights are a list which contains a weight w for each elements g of the special pcgs. Such a weight w represents the smallest subgroup of the LG-series containing g.

Since the LG-series is defined in terms of the lower nilpotent series, Sylow subgroups of the factors and lower p-central series of the Sylow subgroup, the weight w is a triple. More precisely, g is contained in the $w[1]$th term U of the lower nilpotent series of `G`, but not in the next smaller one V. Then $w[3]$ is a prime such that gV is contained in the Sylow $w[3]$-subgroup P/V of U/V. Moreover, gV is contained in the $w[2]$th term of the lower p-central series of P/V.

There are two more attributes of a special pcgs containing information about the LG-series: the list *LGLayers* and the list *LGFirst*. The list of layers corresponds to the elements of the special pcgs and denotes the layer of the LG-series in which an element lies. The list LGFirst corresponds to the LG-series and gives the number of the first element in the special pcgs of the corresponding subgroup.

45.13.1 IsSpecialPcgs

▷ `IsSpecialPcgs(obj)` (property)

tests whether `obj` is a special pcgs.

45.13.2 SpecialPcgs

▷ SpecialPcgs(*pcgs*) (attribute)
▷ SpecialPcgs(*G*) (attribute)

computes a special pcgs for the group defined by *pcgs* or for *G*.

45.13.3 LGWeights

▷ LGWeights(*pcgs*) (attribute)

returns the LGWeights of the special pcgs *pcgs*.

45.13.4 LGLayers

▷ LGLayers(*pcgs*) (attribute)

returns the layers of the special pcgs *pcgs*.

45.13.5 LGFirst

▷ LGFirst(*pcgs*) (attribute)

returns the first indices for each layer of the special pcgs *pcgs*.

45.13.6 LGLength

▷ LGLength(*G*) (attribute)

returns the length of the LG-series of the group *G*, if *G* is solvable, and fail otherwise.

Example

```
gap> G := SmallGroup( 96, 220 );
<pc group of size 96 with 6 generators>
gap> spec := SpecialPcgs( G );
Pcgs([ f1, f2, f3, f4, f5, f6 ])
gap> LGWeights(spec);
[ [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 3 ],
  [ 1, 2, 2 ] ]
gap> LGLayers(spec);
[ 1, 1, 1, 1, 2, 3 ]
gap> LGFirst(spec);
[ 1, 5, 6, 7 ]
gap> LGLength( G );
3
gap> p := SpecialPcgs( Pcgs( SmallGroup( 96, 120 ) ) );
Pcgs([ f1, f2, f3, f4, f5, f6 ])
gap> LGWeights(p);
[ [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 2, 2 ], [ 1, 3, 2 ],
  [ 2, 1, 3 ] ]
```

Thus the first group, `SmallGroup(96, 220)`, has a lower nilpotent series of length 1; that is, the group is nilpotent. It is a direct product of its Sylow subgroups. Moreover the Sylow 2-subgroup is generated by the elements `f1, f2, f3, f4, f6`, and the Sylow 3-subgroup is generated by `f5`. The lower 2-central series of the Sylow 2-subgroup has length 2 and the second subgroup in this series is generated by `f6`.

The second group, `SmallGroup(96, 120)`, has a lower nilpotent series of length 2 and hence is not nilpotent. The second subgroup in this series is just the Sylow 3-subgroup and it is generated by `f6`. The subgroup generated by `f1`, ..., `f5` is a Sylow 2-subgroup of the group and also a head complement to the second head of the group. Its lower 2-central series has length 2.

In this example the `FamilyPcgs` (46.1.1) value of the groups used was a special pcgs, but this is not necessarily the case. For performance reasons it can be worth to enforce this, see `IsomorphismSpecialPcGroup` (46.5.3).

45.13.7 IsInducedPcgsWrtSpecialPcgs

▷ `IsInducedPcgsWrtSpecialPcgs(`*pcgs*`)` (property)

tests whether *pcgs* is induced with respect to a special pcgs.

45.13.8 InducedPcgsWrtSpecialPcgs

▷ `InducedPcgsWrtSpecialPcgs(`*G*`)` (attribute)

computes an induced pcgs with respect to the special pcgs of the parent of *G*.

`InducedPcgsWrtSpecialPcgs` will return a pcgs induced by *a* special pcgs (which might differ from the one you had in mind). If you need an induced pcgs compatible with a *given* special pcgs use `InducedPcgs` (45.7.4) for this special pcgs.

45.14 Action on Subfactors Defined by a Pcgs

When working with a polycyclic group, one often needs to compute matrix operations of the group on a factor of the group. For this purpose there are the functions described in 45.14.1 to 45.14.3.

In certain situations, for example within the computation of conjugacy classes of finite soluble groups as described in [MN89], affine actions of groups are required. For this purpose we introduce the functions `AffineAction` (45.14.4) and `AffineActionLayer` (45.14.5).

45.14.1 VectorSpaceByPcgsOfElementaryAbelianGroup

▷ `VectorSpaceByPcgsOfElementaryAbelianGroup(`*mpcgs, fld*`)` (function)

returns the vector space over *fld* corresponding to the modulo pcgs *mpcgs*. Note that *mpcgs* has to define an elementary abelian p-group where p is the characteristic of *fld*.

45.14.2 LinearAction

▷ `LinearAction(`*gens, basisvectors, linear*`)` (operation)
▷ `LinearOperation(`*gens, basisvectors, linear*`)` (operation)

returns a list of matrices, one for each element of *gens*, which corresponds to the matrix action of the elements in *gens* on the basis *basisvectors* via *linear*.

45.14.3 LinearActionLayer

▷ LinearActionLayer(*G, gens, pcgs*) (function)
▷ LinearOperationLayer(*G, gens, pcgs*) (function)

returns a list of matrices, one for each element of *gens*, which corresponds to the matrix action of *G* on the vector space corresponding to the modulo pcgs *pcgs*.

45.14.4 AffineAction

▷ AffineAction(*gens, basisvectors, linear, transl*) (operation)

return a list of matrices, one for each element of *gens*, which corresponds to the affine action of the elements in *gens* on the basis *basisvectors* via *linear* with translation *transl*.

45.14.5 AffineActionLayer

▷ AffineActionLayer(*G, gens, pcgs, transl*) (function)

returns a list of matrices, one for each element of *gens*, which corresponds to the affine action of *G* on the vector space corresponding to the modulo pcgs *pcgs* with translation *transl*.

Example

```
gap> G := SmallGroup( 96, 51 );
<pc group of size 96 with 6 generators>
gap> spec := SpecialPcgs( G );
Pcgs([ f1, f2, f3, f4, f5, f6 ])
gap> LGWeights( spec );
[ [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 3 ], [ 1, 2, 2 ], [ 1, 2, 2 ],
  [ 1, 3, 2 ] ]
gap> mpcgs := InducedPcgsByPcSequence( spec, spec{[4,5,6]} );
Pcgs([ f4, f5, f6 ])
gap> npcgs := InducedPcgsByPcSequence( spec, spec{[6]} );
Pcgs([ f6 ])
gap> modu := mpcgs mod npcgs;
[ f4, f5 ]
gap> mat:=LinearActionLayer( G, spec{[1,2,3]}, modu );
[ <an immutable 2x2 matrix over GF2>,
  <an immutable 2x2 matrix over GF2>,
  <an immutable 2x2 matrix over GF2> ]
gap> Print( mat, "\n" );
[ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] ]
```

45.15 Orbit Stabilizer Methods for Polycyclic Groups

If a pcgs *pcgs* is known for a group *G*, then orbits and stabilizers can be computed by a special method which is particularly efficient. Note that within this function only the elements in *pcgs* and the relative orders of *pcgs* are needed. Hence this function works effectively even if the elementary operations for *pcgs* are slow.

45.15.1 StabilizerPcgs

▷ StabilizerPcgs(*pcgs, pnt[, acts][, act]*) (function)

computes the stabilizer in the group generated by *pcgs* of the point *pnt*. If given, *acts* are elements by which *pcgs* acts, *act* is the acting function. This function returns a pcgs for the stabilizer which is induced by the ParentPcgs of *pcgs*, that is it is compatible with *pcgs*.

45.15.2 Pcgs_OrbitStabilizer

▷ Pcgs_OrbitStabilizer(*pcgs, domain, pnt, oprs, opr*) (function)

runs a solvable group orbit-stabilizer algorithm on *pnt* with *pcgs* acting via the images *oprs* and the operation function *opr*. The domain *domain* can be used to speed up search, if it is not known, false can be given instead. The function returns a record with components orbit, stabpcgs and lengths, the latter indicating the lengths of the orbit whenever it got extended. This can be used to recompute transversal elements. This function should be used only inside algorithms when speed is essential.

45.16 Operations which have Special Methods for Groups with Pcgs

For the following operations there are special methods for groups with pcgs installed:
IsNilpotentGroup (39.15.3), IsSupersolvableGroup (39.15.8), Size (30.4.6), CompositionSeries (39.17.5), ConjugacyClasses (39.10.2), Centralizer (35.4.4), FrattiniSubgroup (39.12.6), PrefrattiniSubgroup (39.12.7), MaximalSubgroups (39.19.7) and related operations, HallSystem (39.13.6) and related operations, MinimalGeneratingSet (39.22.3), Centre (35.4.5), Intersection (30.5.2), AutomorphismGroup (40.7.1), IrreducibleModules (71.15.1).

45.17 Conjugacy Classes in Solvable Groups

There are a variety of algorithms to compute conjugacy classes and centralizers in solvable groups via epimorphic images ([FN79], [MN89], [The93]). Usually these are only invoked as methods, but it is possible to access the algorithm directly.

45.17.1 ClassesSolvableGroup

▷ ClassesSolvableGroup(*G, mode[, opt]*) (function)

computes conjugacy classes and centralizers in solvable groups. *G* is the acting group. *mode* indicates the type of the calculation:

0 Conjugacy classes

4 Conjugacy test for the two elements in *opt*.candidates

In mode 0 the function returns a list of records containing components *representative* and *centralizer*. In mode 4 it returns a conjugating element.

The optional record *opt* may contain the following components that will affect the algorithm's behaviour:

pcgs
: is a pcgs that will be used for the calculation. The attribute EANormalSeriesByPcgs (45.11.4) must return an appropriate series of normal subgroups with elementary abelian factors among them. The algorithm will step down this series. In the case of the calculation of rational classes, it must be a pcgs refining a central series.

candidates
: is a list of elements for which canonical representatives are to be computed or for which a conjugacy test is performed. They must be given in mode 4. In mode 0 a list of classes corresponding to candidates is returned (which may contain duplicates). The representatives chosen are canonical with respect to pcgs. The records returned also contain components operator such that candidate ^ operator = representative.

consider
: is a function consider(fhome, rep, cenp, K, L). Here fhome is a home pcgs for the factor group F in which the calculation currently takes place, rep is an element of the factor and cenp is a pcgs for the centralizer of rep modulo K. In mode 0, when lifting from F/K to F/L (note: for efficiency reasons, F can be different from *G* or L might be not trivial) this function is called before performing the actual lifting and only those representatives for which it returns true are passed to the next level. This permits for example the calculation of only those classes with small centralizers or classes of restricted orders.

45.17.2 CentralizerSizeLimitConsiderFunction

▷ CentralizerSizeLimitConsiderFunction(*sz*) (function)

returns a function (with arguments fhome, rep, cen, K, L) that can be used in ClassesSolvableGroup (45.17.1) as the consider component of the options record. It will restrict the lifting to those classes, for which the size of the centralizer (in the factor) is at most *sz*.

See also SubgroupsSolvableGroup (39.21.3).

Chapter 46

Pc Groups

Pc groups are polycyclic groups that use the polycyclic presentation for element arithmetic. This presentation gives them a "natural" pcgs, the `FamilyPcgs` (46.1.1) with respect to which pcgs operations as described in chapter 45 are particularly efficient.

Let G be a polycyclic group with pcgs $P = (g_1, \ldots, g_n)$ and corresponding relative orders $(r_1, \ldots, r_n)$. Recall that the r_i are positive integers or infinity and let I be the set of indices i with r_i a positive integer. Then G has a finite presentation on the generators $g_1, \ldots, g_n$ with relations of the following form.

$$\begin{aligned} g_i^{r_i} &= g_{i+1}^{a(i,i,i+1)} \cdots g_n^{a(i,i,n)} \\ &\quad \text{for } 1 \leq i \leq n \text{ and } i \in I \\ g_i^{-1} g_j g_i &= g_{i+1}^{a(i,j,i+1)} \cdots g_n^{a(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \end{aligned}$$

For infinite groups we need additionally

$$\begin{aligned} g_i^{-1} g_j^{-1} g_i &= g_{i+1}^{b(i,j,i+1)} \cdots g_n^{b(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \text{ and } j \notin I \\ g_i g_j g_i^{-1} &= g_{i+1}^{c(i,j,i+1)} \cdots g_n^{c(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \text{ and } i \notin I \\ g_i g_j^{-1} g_i^{-1} &= g_{i+1}^{d(i,j,i+1)} \cdots g_n^{d(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \text{ and } i, j, \notin I \end{aligned}$$

Here the right hand sides are assumed to be words in normal form; that is, for $k \in I$ we have for all exponents $0 \leq a(i,j,k), b(i,j,k), c(i,j,k), d(i,j,k) < r_k$.

A finite presentation of this type is called a *power-conjugate presentation* and a *pc group* is a polycyclic group defined by a power-conjugate presentation. Instead of conjugates we could just as well work with commutators and then the presentation would be called a *power-commutator* presentation. Both types of presentation are abbreviated as *pc presentation*. Note that a pc presentation is a rewriting system.

Clearly, whenever a group G with pcgs P is given, then we can write down the corresponding pc presentation. On the other hand, one may just write down a presentation on n abstract generators $g_1, \ldots, g_n$ with relations of the above form and define a group H by this. Then the subgroups $C_i = \langle g_i, \ldots, g_n \rangle$ of H form a subnormal series whose factors are cyclic or trivial. In the case that all factors are non-trivial, we say that the pc presentation of H is *confluent*. Note that GAP 4 can only work correctly with pc groups defined by a confluent pc presentation.

At the current state of implementations the GAP library contains methods to compute with finite polycyclic groups, while the GAP package Polycyclic by Bettina Eick and Werner Nickel allows also computations with infinite polycyclic groups which are given by a pc-presentation.

Algorithms for pc groups use the methods for polycyclic groups described in chapter 45.

46.1 The family pcgs

Clearly, the generators of a power-conjugate presentation of a pc group G form a pcgs of the pc group. This pcgs is called the *family pcgs*.

46.1.1 FamilyPcgs

▷ `FamilyPcgs(`*`grp`*`)` (attribute)

returns a "natural" pcgs of a pc group *grp* (with respect to which pcgs operations as described in Chapter 45 are particularly efficient).

46.1.2 IsFamilyPcgs

▷ `IsFamilyPcgs(`*`pcgs`*`)` (property)

specifies whether the pcgs is a `FamilyPcgs` (46.1.1) of a pc group.

46.1.3 InducedPcgsWrtFamilyPcgs

▷ `InducedPcgsWrtFamilyPcgs(`*`grp`*`)` (attribute)

returns the pcgs which induced with respect to a family pcgs (see `IsParentPcgsFamilyPcgs` (46.1.4) for further details).

46.1.4 IsParentPcgsFamilyPcgs

▷ `IsParentPcgsFamilyPcgs(`*`pcgs`*`)` (property)

This property indicates that the pcgs *pcgs* is induced with respect to a family pcgs.

This property is needed to distinguish between different independent polycyclic generating sequences which a pc group may have, since the elementary operations for a non-family pcgs may not be as efficient as the elementary operations for the family pcgs.

This can have a significant influence on the performance of algorithms for polycyclic groups. Many algorithms require a pcgs that corresponds to an elementary abelian series (see `PcgsElementaryAbelianSeries` (45.11.2)) or even a special pcgs (see 45.13). If the family pcgs has the required properties, it will be used for these purposes, if not GAP has to work with respect to a new pcgs which is *not* the family pcgs and thus takes longer for elementary calculations like `ExponentsOfPcElement` (45.5.3).

Therefore, if the family pcgs chosen for arithmetic is not of importance it might be worth to *change* to another, nicer, pcgs to speed up calculations. This can be achieved, for example, by using the `Range` (32.3.7) value of the isomorphism obtained by `IsomorphismSpecialPcGroup` (46.5.3).

46.2 Elements of pc groups

46.2.1 Comparison of elements of pc groups

▷ `\=(pcword1, pcword2)` (method)
▷ `\<(pcword1, pcword2)` (method)

The elements of a pc group G are always represented as words in normal form with respect to the family pcgs of G. Thus it is straightforward to compare elements of a pc group, since this boils down to a mere comparison of exponent vectors with respect to the family pcgs. In particular, the word problem is efficiently solvable in pc groups.

46.2.2 Arithmetic operations for elements of pc groups

▷ `\*(pcword1, pcword2)` (method)
▷ `Inverse(pcword)` (method)

However, multiplication and inversion of elements in pc groups is not as straightforward as in arbitrary finitely presented groups where a simple concatenation or reversion of the corresponding words is sufficient (but one cannot solve the word problem).

To multiply two elements in a pc group, we first concatenate the corresponding words and then use an algorithm called *collection* to transform the new word into a word in normal form.

Example

```
gap> g := FamilyPcgs( SmallGroup( 24, 12 ) );
Pcgs([ f1, f2, f3, f4 ])
gap> g[4] * g[1];
f1*f3
gap> (g[2] * g[3])^-1;
f2^2*f3*f4
```

46.3 Pc groups versus fp groups

In theory pc groups are finitely presented groups. In practice the arithmetic in pc groups is different from the arithmetic in fp groups. Thus for technical reasons the pc groups in GAP do not form a subcategory of the fp groups and hence the methods for fp groups cannot be applied to pc groups in general.

46.3.1 IsPcGroup

▷ `IsPcGroup(G)` (Category)

tests whether `G` is a pc group.

Example

```
gap> G := SmallGroup( 24, 12 );
<pc group of size 24 with 4 generators>
gap> IsPcGroup( G );
true
gap> IsFpGroup( G );
false
```

46.3.2 IsomorphismFpGroupByPcgs

▷ IsomorphismFpGroupByPcgs(*pcgs, str*) (function)

It is possible to convert a pc group to a fp group in GAP. The function IsomorphismFpGroupByPcgs computes the power-commutator presentation defined by *pcgs*. The string *str* can be used to give a name to the generators of the fp group.

Example

```
gap> p := FamilyPcgs( SmallGroup( 24, 12 ) );
Pcgs([ f1, f2, f3, f4 ])
gap> iso := IsomorphismFpGroupByPcgs( p, "g" );
[ f1, f2, f3, f4 ] -> [ g1, g2, g3, g4 ]
gap> F := Image( iso );
<fp group of size 24 on the generators [ g1, g2, g3, g4 ]>
gap> RelatorsOfFpGroup( F );
[ g1^2, g2^-1*g1^-1*g2*g1*g2^-1, g3^-1*g1^-1*g3*g1*g4^-1*g3^-1,
  g4^-1*g1^-1*g4*g1*g4^-1*g3^-1, g2^3, g3^-1*g2^-1*g3*g2*g4^-1*g3^-1,
  g4^-1*g2^-1*g4*g2*g3^-1, g3^2, g4^-1*g3^-1*g4*g3, g4^2 ]
```

46.4 Constructing Pc Groups

If necessary, you can supply GAP with a pc presentation by hand. (Although this is the most tedious way to input a pc group.) Note that the pc presentation has to be confluent in order to work with the pc group in GAP.

(If you have already a suitable pcgs in another representation, use PcGroupWithPcgs (46.5.1), see below.)

One way is to define a finitely presented group with a pc presentation in GAP and then convert this presentation into a pc group, see PcGroupFpGroup (46.4.1). Note that this does not work for arbitrary presentations of polycyclic groups, see Chapter 47.14 for further information.

Another way is to create and manipulate a collector of a pc group by hand and to use it to define a pc group. GAP provides different collectors for different collecting strategies; at the moment, there are two collectors to choose from: the single collector for finite pc groups (see SingleCollector (46.4.2)) and the combinatorial collector for finite p-groups. See [Sim94] for further information on collecting strategies.

A collector is initialized with an underlying free group and the relative orders of the pc series. Then one adds the right hand sides of the power and the commutator or conjugate relations one by one. Note that omitted relators are assumed to be trivial.

For performance reasons it is beneficial to enforce a "syllable" representation in the free group (see 37.6).

Note that in the end, the collector has to be converted to a group, see GroupByRws (46.4.6).

With these methods a pc group with arbitrary defining pcgs can be constructed. However, for almost all applications within GAP we need to have a pc group whose defining pcgs is a prime order pcgs, see IsomorphismRefinedPcGroup (46.4.8) and RefinedPcGroup (46.4.9).

46.4.1 PcGroupFpGroup

▷ PcGroupFpGroup(*G*) (function)

creates a pc group *P* from an fp group (see Chapter 47) *G* whose presentation is polycyclic. The resulting group *P* has generators corresponding to the generators of *G*. They are printed in the same way as generators of *G*, but they lie in a different family. If the pc presentation of *G* is not confluent, an error message occurs.

Example

```
gap> F := FreeGroup(IsSyllableWordsFamily,"a","b","c","d");;
gap> a := F.1;; b := F.2;; c := F.3;; d := F.4;;
gap> rels := [a^2, b^3, c^2, d^2, Comm(b,a)/b, Comm(c,a)/d, Comm(d,a),
>             Comm(c,b)/(c*d), Comm(d,b)/c, Comm(d,c)];
[ a^2, b^3, c^2, d^2, b^-1*a^-1*b*a*b^-1, c^-1*a^-1*c*a*d^-1,
  d^-1*a^-1*d*a, c^-1*b^-1*c*b*d^-1*c^-1, d^-1*b^-1*d*b*c^-1,
  d^-1*c^-1*d*c ]
gap> G := F / rels;
<fp group on the generators [ a, b, c, d ]>
gap> H := PcGroupFpGroup( G );
<pc group of size 24 with 4 generators>
```

46.4.2 SingleCollector

▷ SingleCollector(*fgrp, relorders*) (function)
▷ CombinatorialCollector(*fgrp, relorders*) (function)

initializes a single collector or a combinatorial collector, where *fgrp* must be a free group and *relorders* must be a list of the relative orders of the pc series.

A combinatorial collector can only be set up for a finite p-group. Here, the relative orders *relorders* must all be equal and a prime.

46.4.3 SetConjugate

▷ SetConjugate(*coll, j, i, w*) (function)

Let $f_1, \ldots, f_n$ be the generators of the underlying free group of the collector *coll*.

For $i < j$, SetConjugate sets the conjugate $f_j^{f_i}$ to equal *w*, which is assumed to be a word in $f_{i+1}, \ldots, f_n$.

46.4.4 SetCommutator

▷ SetCommutator(*coll, j, i, w*) (function)

Let $f_1, \ldots, f_n$ be the generators of the underlying free group of the collector *coll*.

For $i < j$, SetCommutator sets the commutator of f_j and f_i to equal *w*, which is assumed to be a word in $f_{i+1}, \ldots, f_n$.

46.4.5 SetPower

▷ SetPower(*coll, i, w*) (function)

Let $f_1, \ldots, f_n$ be the generators of the underlying free group of the collector *coll*, and let r_i be the corresponding relative orders.

SetPower sets the power $f_i^{r_i}$ to equal w, which is assumed to be a word in $f_{i+1}, \ldots, f_n$.

46.4.6 GroupByRws

▷ GroupByRws(*coll*) (function)
▷ GroupByRwsNC(*coll*) (function)

creates a group from a rewriting system. In the first version it is checked whether the rewriting system is confluent, in the second version this is assumed to be true.

46.4.7 IsConfluent (for pc groups)

▷ IsConfluent(*G*) (function)

checks whether the pc group *G* has been built from a collector with a confluent power-commutator presentation.

Example

```
gap> F := FreeGroup(IsSyllableWordsFamily, 2 );;
gap> coll1 := SingleCollector( F, [2,3] );
<<single collector, 8 Bits>>
gap> SetConjugate( coll1, 2, 1, F.2 );
gap> SetPower( coll1, 1, F.2 );
gap> G1 := GroupByRws( coll1 );
<pc group of size 6 with 2 generators>
gap> IsConfluent(G1);
true
gap> IsAbelian(G1);
true
gap> coll2 := SingleCollector( F, [2,3] );
<<single collector, 8 Bits>>
gap> SetConjugate( coll2, 2, 1, F.2^2 );
gap> G2 := GroupByRws( coll2 );
<pc group of size 6 with 2 generators>
gap> IsAbelian(G2);
false
```

46.4.8 IsomorphismRefinedPcGroup

▷ IsomorphismRefinedPcGroup(*G*) (attribute)

returns an isomorphism from *G* onto an isomorphic pc group whose family pcgs is a prime order pcgs.

46.4.9 RefinedPcGroup

▷ RefinedPcGroup(*G*) (attribute)

returns the range of the IsomorphismRefinedPcGroup (46.4.8) value of *G*.

46.5 Computing Pc Groups

Another possibility to get a pc group in GAP is to convert a polycyclic group given by some other representation to a pc group. For finitely presented groups there are various quotient methods available. For all other types of groups one can use the following functions.

46.5.1 PcGroupWithPcgs

▷ `PcGroupWithPcgs(`*mpcgs*`)` (attribute)

creates a new pc group *G* whose family pcgs is isomorphic to the (modulo) pcgs *mpcgs*.

Example

```
gap> G := Group( (1,2,3), (3,4,1) );;
gap> PcGroupWithPcgs( Pcgs(G) );
<pc group of size 12 with 3 generators>
```

If a pcgs is only given by a list of pc elements, `PcgsByPcSequence` (45.3.1) can be used:

Example

```
gap> G:=Group((1,2,3,4),(1,2));;
gap> p:=PcgsByPcSequence(FamilyObj(One(G)),
> [ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> PcGroupWithPcgs(p);
<pc group of size 24 with 4 generators>
gap> G := SymmetricGroup( 5 );
Sym( [ 1 .. 5 ] )
gap> H := Subgroup( G, [(1,2,3,4,5), (3,4,5)] );
Group([ (1,2,3,4,5), (3,4,5) ])
gap> modu := ModuloPcgs( G, H );
Pcgs([ (4,5) ])
gap> PcGroupWithPcgs(modu);
<pc group of size 2 with 1 generators>
```

46.5.2 IsomorphismPcGroup

▷ `IsomorphismPcGroup(`*G*`)` (attribute)

returns an isomorphism from *G* onto an isomorphic pc group. The series chosen for this pc representation depends on the method chosen. *G* must be a polycyclic group of any kind, for example a solvable permutation group.

Example

```
gap> G := Group( (1,2,3), (3,4,1) );;
gap> iso := IsomorphismPcGroup( G );
Pcgs([ (2,4,3), (1,2)(3,4), (1,3)(2,4) ]) -> [ f1, f2, f3 ]
gap> H := Image( iso );
Group([ f1, f2, f3 ])
```

46.5.3 IsomorphismSpecialPcGroup

▷ IsomorphismSpecialPcGroup(*G*) (attribute)

returns an isomorphism from *G* onto an isomorphic pc group whose family pcgs is a special pcgs. (This can be beneficial to the runtime of calculations.) *G* may be a polycyclic group of any kind, for example a solvable permutation group.

46.6 Saving a Pc Group

As printing a polycyclic group does not display the presentation, one cannot simply print a pc group to a file to save it. For this purpose we need the following function.

46.6.1 GapInputPcGroup

▷ GapInputPcGroup(*grp, string*) (function)

Example

```
gap> G := SmallGroup( 24, 12 );
<pc group of size 24 with 4 generators>
gap> PrintTo( "save", GapInputPcGroup( G, "H" ) );
gap> Read( "save" );
#I A group of order 24 has been defined.
#I It is called H
gap> H = G;
false
gap> IdSmallGroup( H ) = IdSmallGroup( G );
true
gap> RemoveFile( "save" );;
```

46.7 Operations for Pc Groups

All the operations described in Chapters 39 and 45 apply to a pc group. Nearly all methods for pc groups are methods for groups with pcgs as described in Chapter 45. The only method with is special for pc groups is a method to compute intersections of subgroups, since here a pcgs of a parent group is needed and this can only by guaranteed within pc groups.

46.8 2-Cohomology and Extensions

One of the most interesting applications of pc groups is the possibility to compute with extensions of these groups by elementary abelian groups; that is, H is an extension of G by M, if there exists a normal subgroup N in H which is isomorphic to M such that H/N is isomorphic to G.

Pc groups are particularly suited for such applications, since the 2-cohomology can be computed efficiently for such groups and, moreover, extensions of pc groups by elementary abelian groups can be represented as pc groups again.

To define the elementary abelian group M together with an action of G on M we consider M as a MeatAxe module for G over a finite field (section IrreducibleModules (71.15.1) describes

functions that can be used to obtain certain modules). For further information on meataxe modules see Chapter 69. Note that the matrices defining the module must correspond to the pcgs of the group `G`.

There exists an action of the subgroup of *compatible pairs* in $Aut(G) \times Aut(M)$ which acts on the second cohomology group, see `CompatiblePairs` (46.8.8). 2-cocycles which lie in the same orbit under this action define isomorphic extensions of G. However, there may be isomorphic extensions of G corresponding to cocycles in different orbits.

See also the GAP package GrpConst by Hans Ulrich Besche and Bettina Eick that contains methods to construct up to isomorphism the groups of a given order.

Finally we note that for the computation of split extensions it is not necessary that `M` must correspond to an elementary abelian group. Here it is possible to construct split extensions of arbitrary pc groups, see `SplitExtensions` (46.8.10).

46.8.1 TwoCoboundaries

▷ `TwoCoboundaries(G, M)` (operation)

returns the group of 2-coboundaries of a pc group `G` by the `G`-module `M`. The generators of `M` must correspond to the `Pcgs` (45.2.1) value of `G`. The group of coboundaries is given as vector space over the field underlying `M`.

46.8.2 TwoCocycles

▷ `TwoCocycles(G, M)` (operation)

returns the 2-cocycles of a pc group `G` by the `G`-module `M`. The generators of `M` must correspond to the `Pcgs` (45.2.1) value of `G`. The operation returns a list of vectors over the field underlying `M` and the additive group generated by these vectors is the group of 2-cocyles.

46.8.3 TwoCohomology

▷ `TwoCohomology(G, M)` (operation)

returns a record defining the second cohomology group as factor space of the space of cocycles by the space of coboundaries. `G` must be a pc group and the generators of `M` must correspond to the pcgs of `G`.

Example

```
gap> G := SmallGroup( 4, 2 );
<pc group of size 4 with 2 generators>
gap> mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );
[ <a 1x1 matrix over GF2>, <a 1x1 matrix over GF2> ]
gap> M := GModuleByMats( mats, GF(2) );
rec( IsOverFiniteField := true, dimension := 1, field := GF(2),
  generators := [ <an immutable 1x1 matrix over GF2>,
      <an immutable 1x1 matrix over GF2> ], isMTXModule := true )
gap> TwoCoboundaries( G, M );
[  ]
gap> TwoCocycles( G, M );
[ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
```

```
    [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> cc := TwoCohomology( G, M );;
gap> cc.cohom;
<linear mapping by matrix, <vector space of dimension 3 over GF(
2)> -> ( GF(2)^3 )>
```

46.8.4 Extensions

▷ `Extensions(`*G*`, `*M*`)` (operation)

returns all extensions of *G* by the *G*-module *M* up to equivalence as pc groups.

46.8.5 Extension

▷ `Extension(`*G*`, `*M*`, `*c*`)` (operation)
▷ `ExtensionNC(`*G*`, `*M*`, `*c*`)` (operation)

returns the extension of *G* by the *G*-module *M* via the cocycle *c* as pc groups. The `NC` version does not check the resulting group for consistence.

46.8.6 SplitExtension

▷ `SplitExtension(`*G*`, `*M*`)` (function)

returns the split extension of *G* by the *G*-module *M*.

46.8.7 ModuleOfExtension

▷ `ModuleOfExtension(`*E*`)` (function)

returns the module of an extension *E* of *G* by *M*. This is the normal subgroup of *E* which corresponds to *M*.

Example

```
gap> G := SmallGroup( 4, 2 );;
gap> mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );;
gap> M := GModuleByMats( mats, GF(2) );;
gap> co := TwoCocycles( G, M );;
gap> Extension( G, M, co[2] );
<pc group of size 8 with 3 generators>
gap> SplitExtension( G, M );
<pc group of size 8 with 3 generators>
gap> Extensions( G, M );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
```

```
gap> List(last, IdSmallGroup);
[ [ 8, 5 ], [ 8, 2 ], [ 8, 3 ], [ 8, 3 ], [ 8, 2 ], [ 8, 2 ],
  [ 8, 3 ], [ 8, 4 ] ]
```

Note that the extensions returned by `Extensions` (46.8.4) are computed up to equivalence, but not up to isomorphism.

46.8.8 CompatiblePairs

▷ `CompatiblePairs(`*G, M[, D]*`)` (function)

returns the group of compatible pairs of the group *G* with the *G*-module *M* as subgroup of the direct product Aut(*G*) × Aut(*M*). Here Aut(*M*) is considered as subgroup of a general linear group. The optional argument *D* should be a subgroup of Aut(*G*) × Aut(*M*). If it is given, then only the compatible pairs in *D* are computed.

46.8.9 ExtensionRepresentatives

▷ `ExtensionRepresentatives(`*G, M, P*`)` (operation)

returns all extensions of *G* by the *G*-module *M* up to equivalence under action of *P* where *P* has to be a subgroup of the group of compatible pairs of *G* with *M*.

Example

```
gap> G := SmallGroup( 4, 2 );;
gap> mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );;
gap> M := GModuleByMats( mats, GF(2) );;
gap> A := AutomorphismGroup( G );;
gap> B := GL( 1, 2 );;
gap> D := DirectProduct( A, B );
<group of size 6 with 4 generators>
gap> P := CompatiblePairs( G, M, D );
<group of size 6 with 2 generators>
gap> ExtensionRepresentatives( G, M, P );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
gap> Extensions( G, M );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
```

46.8.10 SplitExtensions

▷ SplitExtensions(*G, aut, N*) (function)

returns the split extensions of the pc group *G* by the pc group *N*. *aut* should be a homomorphism from *G* into Aut(*N*).

In the following example we construct the holomorph of Q_8 as split extension of Q_8 by S_4.

Example

```
gap> N := SmallGroup( 8, 4 );
<pc group of size 8 with 3 generators>
gap> IsAbelian( N );
false
gap> A := AutomorphismGroup( N );
<group of size 24 with 4 generators>
gap> iso := IsomorphismPcGroup( A );
CompositionMapping( Pcgs([ (2,6,5,3), (1,3,5)(2,4,6), (2,5)(3,6),
  (1,4)(3,6) ]) -> [ f1, f2, f3, f4 ], <action isomorphism> )
gap> H := Image( iso );
Group([ f1, f2, f3, f4 ])
gap> G := Subgroup( H, Pcgs(H){[1,2]} );
Group([ f1, f2 ])
gap> inv := InverseGeneralMapping( iso );
[ f1*f2, f2^2*f3, f4, f3 ] ->
[ Pcgs([ f1, f2, f3 ]) -> [ f1*f2, f2, f3 ],
  Pcgs([ f1, f2, f3 ]) -> [ f2, f1*f2, f3 ],
  Pcgs([ f1, f2, f3 ]) -> [ f1*f3, f2, f3 ],
  Pcgs([ f1, f2, f3 ]) -> [ f1, f2*f3, f3 ] ]
gap> K := SplitExtension( G, inv, N );
<pc group of size 192 with 7 generators>
```

46.9 Coding a Pc Presentation

If one wants to store a large number of pc groups, then it can be useful to store them in a compressed format, since pc presentations can be space consuming. Here we introduce a method to code and decode pc presentations by integers. To decode a given code the size of the underlying pc group is needed as well. For the full definition and the coding and decoding procedures see [BE99a]. This method is used with the small groups library, see Section 50.7.

46.9.1 CodePcgs

▷ CodePcgs(*pcgs*) (function)

returns the code corresponding to *pcgs*.

Example

```
gap> G := CyclicGroup(512);;
gap> p := Pcgs( G );;
gap> CodePcgs( p );
162895587718739690298008513020159
```

46.9.2 CodePcGroup

▷ CodePcGroup(*G*) (function)

returns the code for a pcgs of *G*.

Example

```
gap> G := DihedralGroup(512);;
gap> CodePcGroup( G );
29402086275773930705603418039499869124317256417 26
```

46.9.3 PcGroupCode

▷ PcGroupCode(*code, size*) (function)

returns a pc group of size *size* corresponding to *code*. The argument *code* must be a valid code for a pcgs, otherwise anything may happen. Valid codes are usually obtained by one of the functions CodePcgs (46.9.1) or CodePcGroup (46.9.2).

Example

```
gap> G := SmallGroup( 24, 12 );;
gap> p := Pcgs( G );;
gap> code := CodePcgs( p );
5790338948
gap> H := PcGroupCode( code, 24 );
<pc group of size 24 with 4 generators>
gap> map := GroupHomomorphismByImages( G, H, p, FamilyPcgs(H) );
Pcgs([ f1, f2, f3, f4 ]) -> Pcgs([ f1, f2, f3, f4 ])
gap> IsBijective(map);
true
```

46.10 Random Isomorphism Testing

The generic isomorphism test for groups may be applied to pc groups as well. However, this test is often quite time consuming. Here we describe another method to test isomorphism by a probabilistic approach.

46.10.1 RandomIsomorphismTest

▷ RandomIsomorphismTest(*coderecs, n*) (function)

The first argument is a list *coderecs* containing records describing groups, and the second argument is a non-negative integer *n*.

The test returns a sublist of *coderecs* where isomorphic copies detected by the probabilistic test have been removed.

The list *coderecs* should contain records with two components, code and order, describing a group via PcGroupCode(code, order) (see PcGroupCode (46.9.3)).

The integer *n* gives a certain amount of control over the probability to detect all isomorphisms. If it is 0, then nothing will be done at all. The larger *n* is, the larger is the probability of finding all isomorphisms. However, due to the underlying method we can not guarantee that the algorithm finds all isomorphisms, no matter how large *n* is.

Chapter 47

Finitely Presented Groups

A *finitely presented group* (in short: FpGroup) is a group generated by a finite set of *abstract generators* subject to a finite set of *relations* that these generators satisfy. Every finite group can be represented as a finitely presented group, though in almost all cases it is computationally much more efficient to work in another representation (even the regular permutation representation).

Finitely presented groups are obtained by factoring a free group by a set of relators. Their elements know about this presentation and compare accordingly.

So to create a finitely presented group you first have to generate a free group (see `FreeGroup` (37.2.1) for details). Then a list of relators is constructed as words in the generators of the free group and is factored out to obtain the finitely presented group. Its generators *are* the images of the free generators. So for example to create the group

$$\langle a,b \mid a^2,b^3,(ab)^5\rangle$$

you can use the following commands:

Example

```
gap> f := FreeGroup( "a", "b" );;
gap> g := f / [ f.1^2, f.2^3, (f.1*f.2)^5 ];
<fp group on the generators [ a, b ]>
```

Note that you cannot call the generators by their names. These names are not variables, but just display figures. So, if you want to access the generators by their names, you first have to introduce the respective variables and to assign the generators to them.

Example

```
gap> Unbind(a);
gap> GeneratorsOfGroup( g );
[ a, b ]
gap> a;
Error, Variable: 'a' must have a value
gap> a := g.1;; b := g.2;; # assign variables
gap> GeneratorsOfGroup( g );
[ a, b ]
gap> a in f;
false
gap> a in g;
true
```

To relieve you of the tedium of typing the above assignments, *when working interactively*, there is the function `AssignGeneratorVariables` (37.2.3).

Note that the generators of the free group are different from the generators of the FpGroup (even though they are displayed by the same names). That means that words in the generators of the free group are not elements of the finitely presented group. Vice versa elements of the FpGroup are not words.

Example

```
gap> a*b = b*a;
false
gap> (b^2*a*b)^2 = a^0;
true
```

Such calculations comparing elements of an FpGroup may run into problems: There exist finitely presented groups for which no algorithm exists (it is known that no such algorithm can exist) that will tell for two arbitrary words in the generators whether the corresponding elements in the FpGroup are equal.

Therefore the methods used by GAP to compute in finitely presented groups may run into warning errors, run out of memory or run forever. If the FpGroup is (by theory) known to be finite the algorithms are guaranteed to terminate (if there is sufficient memory available), but the time needed for the calculation cannot be bounded a priori. See 47.6 and 47.16.

Example

```
gap> (b^2*a*b)^2;
(b^2*a*b)^2
gap> a^0;
<identity ...>
```

A consequence of our convention is that elements of finitely presented groups are not printed in a unique way. See also `SetReducedMultiplication` (47.3.4).

47.1 IsSubgroupFpGroup and IsFpGroup

47.1.1 IsSubgroupFpGroup

▷ `IsSubgroupFpGroup(`*H*`)` (Category)

returns `true` if *H* is a finitely presented group or a subgroup of a finitely presented group.

47.1.2 IsFpGroup

▷ `IsFpGroup(`*G*`)` (function)

is a synonym for `IsSubgroupFpGroup(`*G*`) and IsGroupOfFamily(`*G*`)`.

Free groups are a special case of finitely presented groups, namely finitely presented groups with no relators.

Another special case are groups given by polycyclic presentations. GAP uses a special representation for these groups which is created in a different way. See chapter 46 for details.

Example

```
gap> g:=FreeGroup(2);
<free group on the generators [ f1, f2 ]>
gap> IsFpGroup(g);
true
gap> h:=CyclicGroup(2);
<pc group of size 2 with 1 generators>
gap> IsFpGroup(h);
false
```

47.1.3 InfoFpGroup

▷ `InfoFpGroup` (info class)

The info class for functions dealing with finitely presented groups is `InfoFpGroup`.

47.2 Creating Finitely Presented Groups

47.2.1 \/ (for a free group and a list of elements)

▷ `\/(F, rels)` (method)

creates a finitely presented group given by the presentation $\langle gens \mid rels \rangle$ where *gens* are the free generators of the free group `F`. Note that relations are entered as *relators*, i.e., as words in the generators of the free group. To enter an equation use the quotient operator, i.e., for the relation $a^b = ab$ one has to enter $a^b/(ab)$.

The same result is obtained with the infix operator `/`, i.e., as `F / rels`.

Example

```
gap> f := FreeGroup( 3 );;
gap> f / [ f.1^4, f.2^3, f.3^5, f.1*f.2*f.3 ];
<fp group on the generators [ f1, f2, f3 ]>
```

47.2.2 FactorGroupFpGroupByRels

▷ `FactorGroupFpGroupByRels(G, elts)` (function)

returns the factor group G/N of `G` by the normal closure N of `elts` where `elts` is expected to be a list of elements of `G`.

47.2.3 ParseRelators

▷ `ParseRelators(gens, rels)` (function)

Will translate a list of relations as given in print, e.g. $xy^2 = (xy^3x)^2xy = yzx$ into relators. `gens` must be a list of generators of a free group, each being displayed by a single letter. `rels` is a string that lists a sequence of equalities. These must be written in the letters which are the names of the generators in `gens`. Change of upper/lower case is interpreted to indicate inverses.

Example

```
gap> f:=FreeGroup("x","y","z");;
gap> AssignGeneratorVariables(f);
#I  Assigned the global variables [ x, y, z ]
gap> r:=ParseRelators([x,y,z],
> "x^2 = y^5 = z^3 = (xyxyxy^4)^2 = (xz)^2 = (y^2z)^2 = 1");
[ x^2, y^5, z^3, (x*z)^2, (y^2*z)^2, ((x*y)^3*y^3)^2 ]
gap> g:=f/r;
<fp group on the generators [ x, y, z ]>
```

47.2.4 StringFactorizationWord

▷ `StringFactorizationWord(w)` (function)

returns a string that expresses a given word `w` in compact form written as a string. Inverses are expressed by changing the upper/lower case of the generators, recurring expressions are written as products.

Example

```
gap> StringFactorizationWord(z^-1*x*y*y*y*x*x*y*y*y*x*x*y^-1*x);
"Z(xy3x)2Yx"
```

47.3 Comparison of Elements of Finitely Presented Groups

47.3.1 \= (for two elements in a f.p. group)

▷ `\=(a, b)` (method)

Two elements of a finitely presented group are equal if they are equal in this group. Nevertheless they may be represented as different words in the generators. Because of the fundamental problems mentioned in the introduction to this chapter such a test may take very long and cannot be guaranteed to finish.

The method employed by GAP for such an equality test use the underlying finitely presented group. First (unless this group is known to be infinite) GAP tries to find a faithful permutation representation by a bounded Todd-Coxeter. If this fails, a Knuth-Bendix (see 52.6) is attempted and the words are compared via their normal form.

If only elements in a subgroup are to be tested for equality it thus can be useful to translate the problem in a new finitely presented group by rewriting (see `IsomorphismFpGroup` (47.11.1));

The equality test of elements underlies many "basic" calculations, such as the order of an element, and the same type of problems can arise there. In some cases, working with rewriting systems can still help to solve the problem. The kbmag package provides such functionality, see the package manual for further details.

47.3.2 \< (for two elements in a f.p. group)

▷ `\<(a, b)` (method)

Compared with equality testing, problems get even worse when trying to compute a total ordering on the elements of a finitely presented group. As any ordering that is guaranteed to be reproducible in

different runs of GAP or even with different groups given by syntactically equal presentations would be prohibitively expensive to implement, the ordering of elements is depending on a method chosen by GAP and not guaranteed to stay the same when repeating the construction of an FpGroup. The only guarantee given for the < ordering for such elements is that it will stay the same for one family during its lifetime. The attribute `FpElmComparisonMethod` (47.3.3) is used to obtain a comparison function for a family of FpGroup elements.

47.3.3 FpElmComparisonMethod

▷ `FpElmComparisonMethod(`*fam*`)` (attribute)

If *fam* is the elements family of a finitely presented group this attribute returns a function `smaller(`*left*`, `*right*`)` that will be used to compare elements in *fam*.

47.3.4 SetReducedMultiplication

▷ `SetReducedMultiplication(`*obj*`)` (function)

For an FpGroup *obj*, an element *obj* of it or the family *obj* of its elements, this function will force immediate reduction when multiplying, keeping words short at extra cost per multiplication.

47.4 Preimages in the Free Group

47.4.1 FreeGroupOfFpGroup

▷ `FreeGroupOfFpGroup(`*G*`)` (attribute)

returns the underlying free group for the finitely presented group *G*. This is the group generated by the free generators provided by the `FreeGeneratorsOfFpGroup` (47.4.2) value of *G*.

47.4.2 FreeGeneratorsOfFpGroup

▷ `FreeGeneratorsOfFpGroup(`*G*`)` (attribute)
▷ `FreeGeneratorsOfWholeGroup(`*U*`)` (operation)

`FreeGeneratorsOfFpGroup` returns the underlying free generators corresponding to the generators of the finitely presented group *G* which must be a full FpGroup.

`FreeGeneratorsOfWholeGroup` also works for subgroups of an FpGroup and returns the free generators of the full group that defines the family.

47.4.3 RelatorsOfFpGroup

▷ `RelatorsOfFpGroup(`*G*`)` (attribute)

returns the relators of the finitely presented group *G* as words in the free generators provided by the `FreeGeneratorsOfFpGroup` (47.4.2) value of *G*.

Example

```
gap> f := FreeGroup( "a", "b" );;
gap> g := f / [ f.1^5, f.2^2, f.1^f.2*f.1 ];
<fp group on the generators [ a, b ]>
gap> Size( g );
10
gap> FreeGroupOfFpGroup( g ) = f;
true
gap> FreeGeneratorsOfFpGroup( g );
[ a, b ]
gap> RelatorsOfFpGroup( g );
[ a^5, b^2, b^-1*a*b*a ]
```

Note that these attributes are only available for the *full* finitely presented group. It is possible (for example by using Subgroup (39.3.1)) to construct a subgroup of index 1 which is not identical to the whole group. The latter one can be obtained in this situation via Parent (31.7.1).

Elements of a finitely presented group are not words, but are represented using a word from the free group as representative. The following two commands obtain this representative, respectively create an element in the finitely presented group.

47.4.4 UnderlyingElement (fp group elements)

▷ UnderlyingElement(*elm*) (function)

Let *elm* be an element of a group whose elements are represented as words with further properties. Then UnderlyingElement returns the word from the free group that is used as a representative for *elm*.

Example

```
gap> w := g.1*g.2;
a*b
gap> IsWord( w );
false
gap> ue := UnderlyingElement( w );
a*b
gap> IsWord( ue );
true
```

47.4.5 ElementOfFpGroup

▷ ElementOfFpGroup(*fam, word*) (operation)

If *fam* is the elements family of a finitely presented group and *word* is a word in the free generators underlying this finitely presented group, this operation creates the element with the representative *word* in the free group.

Example

```
gap> ge := ElementOfFpGroup( FamilyObj( g.1 ), f.1*f.2 );
a*b
gap> ge in f;
false
```

```
gap> ge in g;
true
```

47.5 Operations for Finitely Presented Groups

Finitely presented groups are groups and so all operations for groups should be applicable to them (though not necessarily efficient methods are available). Most methods for finitely presented groups rely on coset enumeration. See 47.6 for details.

The command `IsomorphismPermGroup` (43.3.1) can be used to obtain a faithful permutation representation, if such a representation of small degree exists. (Otherwise it might run very long or fail.)

Example

```
gap> f := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> g := f / [ f.1^2, f.2^3, (f.1*f.2)^5 ];
<fp group on the generators [ a, b ]>
gap> h := IsomorphismPermGroup( g );
[ a, b ] -> [ (1,2)(4,5), (2,3,4) ]
gap> u:=Subgroup(g,[g.1*g.2]);;rt:=RightTransversal(g,u);
RightTransversal(<fp group of size 60 on the generators
[ a, b ]>,Group([ a*b ]))
gap> Image(ActionHomomorphism(g,rt,OnRight));
Group([ (1,2)(3,4)(5,7)(6,8)(9,10)(11,12),
  (1,3,2)(4,5,6)(7,8,9)(10,11,12) ])
```

47.5.1 PseudoRandom (for finitely presented groups)

▷ `PseudoRandom(`*F: radius := l*`)` (method)

The default algorithm for `PseudoRandom` (30.7.2) makes little sense for finitely presented or free groups, as it produces words that are extremely long.

By specifying the option `radius`, instead elements are taken as words in the generators of *F* in the ball of radius *l* with equal distribution in the free group.

Example

```
gap> PseudoRandom(g:radius:=20);
a^3*b^2*a^-2*b^-1*a*b^-4*a*b^-1*a*b^-4
```

47.6 Coset Tables and Coset Enumeration

Coset enumeration (see [Neu82] for an explanation) is one of the fundamental tools for the examination of finitely presented groups. This section describes GAP functions that can be used to invoke a coset enumeration.

Note that in addition to the built-in coset enumerator there is the GAP package ACE. Moreover, GAP provides an interactive Todd-Coxeter in the GAP package ITC which is based on the XGAP package.

47.6.1 CosetTable

▷ `CosetTable(G, H)` (operation)

returns the coset table of the finitely presented group `G` on the cosets of the subgroup `H`.

Basically a coset table is the permutation representation of the finitely presented group on the cosets of a subgroup (which need not be faithful if the subgroup has a nontrivial core). Most of the set theoretic and group functions use the regular representation of `G`, i.e., the coset table of `G` over the trivial subgroup.

The coset table is returned as a list of lists. For each generator of `G` and its inverse the table contains a generator list. A generator list is simply a list of integers. If l is the generator list for the generator g and if $l[i] = j$ then generator g takes the coset i to the coset j by multiplication from the right. Thus the permutation representation of `G` on the cosets of `H` is obtained by applying `PermList` (42.5.2) to each generator list.

The coset table is standard (see below).

For finitely presented groups, a coset table is computed by a Todd-Coxeter coset enumeration. Note that you may influence the performance of that enumeration by changing the values of the global variables `CosetTableDefaultLimit` (47.6.7) and `CosetTableDefaultMaxLimit` (47.6.6) described below and that the options described under `CosetTableFromGensAndRels` (47.6.5) are recognized.

Example

```
gap> tab := CosetTable(g, Subgroup(g, [ g.1, g.2*g.1*g.2*g.1*g.2^-1 ]));
[ [ 1, 4, 5, 2, 3 ], [ 1, 4, 5, 2, 3 ], [ 2, 3, 1, 4, 5 ],
  [ 3, 1, 2, 4, 5 ] ]
gap> List( last, PermList );
[ (2,4)(3,5), (2,4)(3,5), (1,2,3), (1,3,2) ]
gap> PrintArray( TransposedMat( tab ) );
[ [  1,  1,  2,  3 ],
  [  4,  4,  3,  1 ],
  [  5,  5,  1,  2 ],
  [  2,  2,  4,  4 ],
  [  3,  3,  5,  5 ] ]
```

The last printout in the preceding example provides the coset table in the form in which it is usually used in hand calculations: The rows correspond to the cosets, the columns correspond to the generators and their inverses in the ordering $g_1, g_1^{-1}, g_2, g_2^{-1}$. (See section 47.7 for a description on the way the numbers are assigned.)

47.6.2 TracedCosetFpGroup

▷ `TracedCosetFpGroup(tab, word, pt)` (function)

Traces the coset number `pt` under the word `word` through the coset table `tab`. (Note: `word` must be in the free group, use `UnderlyingElement` (47.4.4) if in doubt.)

Example

```
gap> TracedCosetFpGroup(tab,UnderlyingElement(g.1),2);
4
```

47.6.3 FactorCosetAction (for fp groups)

▷ `FactorCosetAction(`*G, H*`)` (function)

returns the action of *G* on the cosets of its subgroup *H*.

Example

```
gap> u := Subgroup( g, [ g.1, g.1^g.2 ] );
Group([ a, b^-1*a*b ])
gap> FactorCosetAction( g, u );
[ a, b ] -> [ (2,4)(5,6), (1,2,3)(4,5,6) ]
```

47.6.4 CosetTableBySubgroup

▷ `CosetTableBySubgroup(`*G, H*`)` (operation)

returns a coset table for the action of *G* on the cosets of *H*. The columns of the table correspond to the `GeneratorsOfGroup` (39.2.4) value of *G*.

47.6.5 CosetTableFromGensAndRels

▷ `CosetTableFromGensAndRels(`*fgens, grels, fsgens*`)` (function)

is an internal function which is called by the functions `CosetTable` (47.6.1), `CosetTableInWholeGroup` (47.8.1) and others. It is, in fact, the proper working horse that performs a Todd-Coxeter coset enumeration. *fgens* must be a set of free generators and *grels* a set of relators in these generators. *fsgens* are subgroup generators expressed as words in these generators. The function returns a coset table with respect to *fgens*.

`CosetTableFromGensAndRels` will call `TCENUM.CosetTableFromGensAndRels`. This makes it possible to replace the built-in coset enumerator with another one by assigning `TCENUM` to another record.

The library version which is used by default performs a standard Felsch strategy coset enumeration. You can call this function explicitly as `GAPTCENUM.CosetTableFromGensAndRels` even if other coset enumerators are installed.

The expected parameters are

fgens
: generators of the free group *F*

grels
: relators as words in *F*

fsgens
: subgroup generators as words in *F*.

`CosetTableFromGensAndRels` processes two options (see chapter 8):

`max` The limit of the number of cosets to be defined. If the enumeration does not finish with this number of cosets, an error is raised and the user is asked whether she wants to continue. The default value is the value given in the variable `CosetTableDefaultMaxLimit`. (Due to the algorithm the actual limit used can be a bit higher than the number given.)

`silent`

If set to `true` the algorithm will not raise the error mentioned under option `max` but silently return `fail`. This can be useful if an enumeration is only wanted unless it becomes too big.

47.6.6 CosetTableDefaultMaxLimit

▷ `CosetTableDefaultMaxLimit` (global variable)

is the default limit for the number of cosets allowed in a coset enumeration.

A coset enumeration will not finish if the subgroup does not have finite index, and even if it has it may take many more intermediate cosets than the actual index of the subgroup is. To avoid a coset enumeration "running away" therefore GAP has a "safety stop" built in. This is controlled by the global variable `CosetTableDefaultMaxLimit`.

If this number of cosets is reached, GAP will issue an error message and prompt the user to either continue the calculation or to stop it. The default value is 4096000.

See also the description of the options to `CosetTableFromGensAndRels` (47.6.5).

Example

```
gap> f := FreeGroup( "a", "b" );;
gap> u := Subgroup( f, [ f.2 ] );
Group([ b ])
gap> Index( f, u );
Error, the coset enumeration has defined more than 4096000 cosets
 called from
TCENUM.CosetTableFromGensAndRels( fgens, grels, fsgens ) called from
CosetTableFromGensAndRels( fgens, grels, fsgens ) called from
TryCosetTableInWholeGroup( H ) called from
CosetTableInWholeGroup( H ) called from
IndexInWholeGroup( H ) called from
...
Entering break read-eval-print loop ...
type 'return;' if you want to continue with a new limit of 8192000 cosets,
type 'quit;' if you want to quit the coset enumeration,
type 'maxlimit := 0; return;' in order to continue without a limit
brk> quit;
```

At this point, a `break`-loop (see Section 6.4) has been entered. The line beginning `Error` tells you why this occurred. The next seven lines occur if `OnBreak` (6.4.3) has its default value `Where` (6.4.5). They explain, in this case, how GAP came to be doing a coset enumeration. Then you are given a number of options of how to escape the `break`-loop: you can either continue the calculation with a larger number of permitted cosets, stop the calculation if you don't expect the enumeration to finish (like in the example above), or continue without a limit on the number of cosets. (Choosing the first option will, of course, land you back in a `break`-loop. Try it!)

Setting `CosetTableDefaultMaxLimit` (or the `max` option value, for any function that invokes a coset enumeration) to `infinity` (18.2.1) (or to 0) will force all coset enumerations to continue until they either get a result or exhaust the whole available space. For example, each of the following two inputs

```
gap> CosetTableDefaultMaxLimit := 0;;
gap> Index( f, u );
```

or

```
gap> Index( f, u : max := 0 );
```

have essentially the same effect as choosing the third option (typing: `maxlimit := 0; return;`) at the `brk>` prompt above (instead of `quit;`).

47.6.7 CosetTableDefaultLimit

▷ `CosetTableDefaultLimit` (global variable)

is the default number of cosets with which any coset table is initialized before doing a coset enumeration.

The function performing this coset enumeration will automatically extend the table whenever necessary (as long as the number of cosets does not exceed the value of `CosetTableDefaultMaxLimit` (47.6.6)), but this is an expensive operation. Thus, if you change the value of `CosetTableDefaultLimit`, you should set it to a number of cosets that you expect to be sufficient for your subsequent coset enumerations. On the other hand, if you make it too large, your job will unnecessarily waste a lot of space.

The default value of `CosetTableDefaultLimit` is 1000.

47.6.8 MostFrequentGeneratorFpGroup

▷ `MostFrequentGeneratorFpGroup(`*G*`)` (function)

is an internal function which is used in some applications of coset table methods. It returns the first of those generators of the given finitely presented group *G* which occur most frequently in the relators.

47.6.9 IndicesInvolutaryGenerators

▷ `IndicesInvolutaryGenerators(`*G*`)` (attribute)

returns the indices of those generators of the finitely presented group *G* which are known to be involutions. This knowledge is used by internal functions to improve the performance of coset enumerations.

47.7 Standardization of coset tables

For any two coset numbers i and j with $i < j$ the first occurrence of i in a coset table precedes the first occurrence of j with respect to the usual row-wise ordering of the table entries. Following the notation of Charles Sims' book on computation with finitely presented groups [Sim94] we call such a table a *standard coset table*.

The table entries which contain the first occurrences of the coset numbers $i > 1$ recursively provide for each i a representative of the corresponding coset in form of a unique word w_i in the generators and inverse generators of G. The first coset (which is H itself) can be represented by the empty word

w_1. A coset table is standard if and only if the words $w_1, w_2, \ldots$ are length-plus-lexicographic ordered (as defined in [Sim94]), for short: *lenlex*.

This standardization of coset tables is different from that used in GAP versions 4.2 and earlier. Before that, we ignored the columns that correspond to inverse generators and hence only considered words in the generators of G. We call this older ordering the *semilenlex* standard as it also applies to the case of semigroups where no inverses of the generators are known.

We changed our default from the semilenlex standard to the lenlex standard to be consistent with [Sim94]. However, the semilenlex standardisation remains available and the convention used for all implicit standardisations can be selected by setting the value of the global variable `CosetTableStandard` (47.7.1) to either `"lenlex"` or `"semilenlex"`. Independent of the current value of `CosetTableStandard` (47.7.1) you can standardize (or restandardize) a coset table at any time using `StandardizeTable` (47.7.2).

47.7.1 CosetTableStandard

▷ `CosetTableStandard` (global variable)

specifies the definition of a *standard coset table*. It is used whenever coset tables or augmented coset tables are created. Its value may be `"lenlex"` or `"semilenlex"`. If it is `"lenlex"` coset tables will be standardized using all their columns as defined in Charles Sims' book (this is the new default standard of GAP). If it is `"semilenlex"` they will be standardized using only their generator columns (this was the original GAP standard). The default value of `CosetTableStandard` is `"lenlex"`.

47.7.2 StandardizeTable

▷ `StandardizeTable(`*`table, standard`*`)` (function)

standardizes the given coset table *`table`*. The second argument is optional. It defines the standard to be used, its values may be `"lenlex"` or `"semilenlex"` specifying the new or the old convention, respectively. If no value for the parameter *`standard`* is provided the function will use the global variable `CosetTableStandard` (47.7.1) instead. Note that the function alters the given table, it does not create a copy.

Example

```
gap> StandardizeTable( tab, "semilenlex" );
gap> PrintArray( TransposedMat( tab ) );
[ [  1,  1,  2,  4 ],
  [  3,  3,  4,  1 ],
  [  2,  2,  3,  3 ],
  [  5,  5,  1,  2 ],
  [  4,  4,  5,  5 ] ]
```

47.8 Coset tables for subgroups in the whole group

47.8.1 CosetTableInWholeGroup

▷ `CosetTableInWholeGroup(`*`H`*`)` (attribute)

▷ `TryCosetTableInWholeGroup(`*`H`*`)` (operation)

is equivalent to `CosetTable(G,H)` where `G` is the (unique) finitely presented group such that `H` is a subgroup of `G`. It overrides a `silent` option (see `CosetTableFromGensAndRels` (47.6.5)) with `false`.

The variant `TryCosetTableInWholeGroup` does not override the `silent` option with `false` in case a coset table is only wanted if not too expensive. It will store a result that is not `fail` in the attribute `CosetTableInWholeGroup`.

47.8.2 SubgroupOfWholeGroupByCosetTable

▷ `SubgroupOfWholeGroupByCosetTable(fpfam, tab)` (function)

takes a family *fpfam* of an FpGroup and a coset table *tab* and returns the subgroup of *fpfam*`!.wholeGroup` defined by this coset table. See also `CosetTableBySubgroup` (47.6.4).

47.9 Augmented Coset Tables and Rewriting

47.9.1 AugmentedCosetTableInWholeGroup

▷ `AugmentedCosetTableInWholeGroup(H[, gens])` (operation)

For a subgroup `H` of a finitely presented group, this function returns an augmented coset table. If a generator set *gens* is given, it is guaranteed that *gens* will be a subset of the primary and secondary subgroup generators of this coset table.

It is mutable so we are permitted to add further entries. However existing entries may not be changed. Any entries added however should correspond to the subgroup only and not to an homomorphism.

47.9.2 AugmentedCosetTableMtc

▷ `AugmentedCosetTableMtc(G, H, type, string)` (function)

is an internal function used by the subgroup presentation functions described in 48.2. It applies a Modified Todd-Coxeter coset representative enumeration to construct an augmented coset table (see 48.2) for the given subgroup `H` of `G`. The subgroup generators will be named *string*`1`, *string*`2`, ...

The function accepts the options `max` and `silent` as described for the function `CosetTableFromGensAndRels` (47.6.5).

47.9.3 AugmentedCosetTableRrs

▷ `AugmentedCosetTableRrs(G, table, type, string)` (function)

is an internal function used by the subgroup presentation functions described in 48.2. It applies the Reduced Reidemeister-Schreier method to construct an augmented coset table for the subgroup of `G` which is defined by the given coset table *table*. The new subgroup generators will be named *string*`1`, *string*`2`, ...

47.9.4 RewriteWord

▷ RewriteWord(*aug, word*) (function)

RewriteWord rewrites *word* (which must be a word in the underlying free group with respect to which the augmented coset table *aug* is given) in the subgroup generators given by the augmented coset table *aug*. It returns a Tietze-type word (i.e. a list of integers), referring to the primary and secondary generators of *aug*.

If *word* is not contained in the subgroup, fail is returned.

47.10 Low Index Subgroups

47.10.1 LowIndexSubgroupsFpGroupIterator

▷ LowIndexSubgroupsFpGroupIterator(*G[, H], index[, excluded]*) (operation)
▷ LowIndexSubgroupsFpGroup(*G[, H], index[, excluded]*) (operation)

These functions compute representatives of the conjugacy classes of subgroups of the finitely presented group *G* that contain the subgroup *H* of *G* and that have index less than or equal to *index*.

LowIndexSubgroupsFpGroupIterator returns an iterator (see 30.8) that can be used to run over these subgroups, and LowIndexSubgroupsFpGroup returns the list of these subgroups. If one is interested only in one or a few subgroups up to a given index then preferably the iterator should be used.

If the optional argument *excluded* has been specified, then it is expected to be a list of words in the free generators of the underlying free group of *G*, and LowIndexSubgroupsFpGroup returns only those subgroups of index at most *index* that contain *H*, but do not contain any conjugate of any of the group elements defined by these words.

If not given, *H* defaults to the trivial subgroup.

The algorithm used finds the requested subgroups by systematically running through a tree of all potential coset tables of *G* of length at most *index* (where it skips all branches of that tree for which it knows in advance that they cannot provide new classes of such subgroups). The time required to do this depends, of course, on the presentation of *G*, but in general it will grow exponentially with the value of *index*. So you should be careful with the choice of *index*.

Example

```
gap> li:=LowIndexSubgroupsFpGroup( g, TrivialSubgroup( g ), 10 );
[ Group(<fp, no generators known>), Group(<fp, no generators known>),
  Group(<fp, no generators known>), Group(<fp, no generators known>) ]
```

By default, the algorithm computes no generating sets for the subgroups. This can be enforced with GeneratorsOfGroup (39.2.4):

Example

```
gap> GeneratorsOfGroup(li[2]);
[ a, b*a*b^-1 ]
```

If we are interested just in one (proper) subgroup of index at most 10, we can use the function that returns an iterator. The first subgroup found is the group itself, except if a list of excluded elements is entered (see below), so we look at the second subgroup.

Example
```
gap> iter:= LowIndexSubgroupsFpGroupIterator( g, 10 );;
gap> s1:= NextIterator( iter );;  Index( g, s1 );
1
gap> IsDoneIterator( iter );
false
gap> s2:= NextIterator( iter );;  s2 = li[2];
true
```

As an example for an application of the optional parameter *excluded*, we compute all conjugacy classes of torsion free subgroups of index at most 24 in the group $G = \langle x,y,z \mid x^2, y^4, z^3, (xy)^3, (yz)^2, (xz)^3 \rangle$. It is know from theory that each torsion element of this group is conjugate to a power of x, y, z, xy, xz, or yz. (Note that this includes conjugates of y^2.)

Example
```
gap> F := FreeGroup( "x", "y", "z" );;
gap> x := F.1;; y := F.2;; z := F.3;;
gap> G := F / [ x^2, y^4, z^3, (x*y)^3, (y*z)^2, (x*z)^3 ];;
gap> torsion := [ x, y, y^2, z, x*y, x*z, y*z ];;
gap> SetInfoLevel( InfoFpGroup, 2 );
gap> lis := LowIndexSubgroupsFpGroup(G, TrivialSubgroup(G), 24, torsion);;
#I  LowIndexSubgroupsFpGroup called
#I   class 1 of index 24 and length 8
#I   class 2 of index 24 and length 24
#I   class 3 of index 24 and length 24
#I   class 4 of index 24 and length 24
#I   class 5 of index 24 and length 24
#I  LowIndexSubgroupsFpGroup done. Found 5 classes
gap> SetInfoLevel( InfoFpGroup, 0 );
```

If a particular image group is desired, the operation `GQuotients` (40.9.4) (see 47.14) can be useful as well.

47.11 Converting Groups to Finitely Presented Groups

47.11.1 IsomorphismFpGroup

▷ `IsomorphismFpGroup(`*G*`)` (attribute)

returns an isomorphism from the given finite group *G* to a finitely presented group isomorphic to *G*. The function first *chooses a set of generators of* *G* and then computes a presentation in terms of these generators.

Example
```
gap> g := Group( (2,3,4,5), (1,2,5) );;
gap> iso := IsomorphismFpGroup( g );
[ (4,5), (1,2,3,4,5), (1,3,2,4,5) ] -> [ F1, F2, F3 ]
gap> fp := Image( iso );
<fp group on the generators [ F1, F2, F3 ]>
gap> RelatorsOfFpGroup( fp );
[ F1^2, F1^-1*F2*F1*F2^-1*F3*F2^-2, F1^-1*F3*F1*F2*F3^-1*F2*F3*F2^-1,
  F2^5*F3^-5, F2^5*(F3^-1*F2^-1)^2, (F2^-2*F3^2)^2 ]
```

47.11.2 IsomorphismFpGroupByGenerators

▷ IsomorphismFpGroupByGenerators(*G, gens[, string]*) (attribute)
▷ IsomorphismFpGroupByGeneratorsNC(*G, gens, string*) (attribute)

returns an isomorphism from a finite group *G* to a finitely presented group *F* isomorphic to *G*. The generators of *F* correspond to the *generators of G given in the list gens*. If *string* is given it is used to name the generators of the finitely presented group.

The NC version will avoid testing whether the elements in *gens* generate *G*.

Example

```
gap> SetInfoLevel( InfoFpGroup, 1 );
gap> iso := IsomorphismFpGroupByGenerators( g, [ (1,2), (1,2,3,4,5) ] );
#I  the image group has 2 gens and 5 rels of total length 39
[ (1,2), (1,2,3,4,5) ] -> [ F1, F2 ]
gap> fp := Image( iso );
<fp group of size 120 on the generators [ F1, F2 ]>
gap> RelatorsOfFpGroup( fp );
[ F1^2, F2^5, (F2^-1*F1)^4, (F2^-1*F1*F2*F1)^3, (F2^2*F1*F2^-2*F1)^2 ]
```

The main task of the function IsomorphismFpGroupByGenerators is to find a presentation of *G* in the provided generators *gens*. In the case of a permutation group *G* it does this by first constructing a stabilizer chain of *G* and then it works through that chain from the bottom to the top, recursively computing a presentation for each of the involved stabilizers. The method used is essentially an implementation of John Cannon's multi-stage relations-finding algorithm as described in [Neu82] (see also [Can73] for a more graph theoretical description). Moreover, it makes heavy use of Tietze transformations in each stage to avoid an explosion of the total length of the relators.

Note that because of the random methods involved in the construction of the stabilizer chain the resulting presentations of *G* will in general be different for repeated calls with the same arguments.

Example

```
gap> M12 := MathieuGroup( 12 );
Group([ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6),
  (1,12)(2,11)(3,6)(4,8)(5,9)(7,10) ])
gap> gens := GeneratorsOfGroup( M12 );;
gap> iso := IsomorphismFpGroupByGenerators( M12, gens );;
#I  the image group has 3 gens and 23 rels of total length 628
gap> iso := IsomorphismFpGroupByGenerators( M12, gens );;
#I  the image group has 3 gens and 23 rels of total length 569
```

Also in the case of a permutation group *G*, the function IsomorphismFpGroupByGenerators supports the option method that can be used to modify the strategy. The option method may take the following values.

method := "regular"
: This may be specified for groups of small size, up to 10^5 say. It implies that the function first constructs a regular representation *R* of *G* and then a presentation of *R*. In general, this presentation will be much more concise than the default one, but the price is the time needed for the construction of *R*.

method := ["regular", bound]
: This is a refinement of the previous possibility. In this case, bound should be an integer, and if

so the method "regular" as described above is applied to the largest stabilizer in the stabilizer chain of *G* whose size does not exceed the given bound and then the multi-stage algorithm is used to work through the chain from that subgroup to the top.

method := "fast"

This chooses an alternative method which essentially is a kind of multi-stage algorithm for a stabilizer chain of *G* but does not make any attempt do reduce the number of relators as it is done in Cannon's algorithm or to reduce their total length. Hence it is often much faster than the default method, but the total length of the resulting presentation may be huge.

method := "default"

This simply means that the default method shall be used, which is the case if the option method is not given a value.

Example

```
gap> iso := IsomorphismFpGroupByGenerators( M12, gens :
>                                           method := "regular" );;
#I  the image group has 3 gens and 11 rels of total length 92
gap> iso := IsomorphismFpGroupByGenerators( M12, gens :
>                                           method := "fast" );;
#I  the image group has 3 gens and 162 rels of total length 3737
```

Though the option method := "regular" is only checked in the case of a permutation group it also affects the performance and the results of the function IsomorphismFpGroupByGenerators for other groups, e. g. for matrix groups. This happens because, for these groups, the function first calls the function NiceMonomorphism (40.5.2) to get a bijective action homomorphism from *G* to a suitable permutation group, *P* say, and then, recursively, calls itself for the group *P* so that now the option becomes relevant.

Example

```
gap> G := ImfMatrixGroup( 5, 1, 3 );
ImfMatrixGroup(5,1,3)
gap> gens := GeneratorsOfGroup( G );
[ [ [ -1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 0, 1, 0 ],
      [ -1, -1, -1, -1, 2 ], [ -1, 0, 0, 0, 1 ] ],
  [ [ 0, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ],
      [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1 ] ] ]
gap> iso := IsomorphismFpGroupByGenerators( G, gens );;
#I  the image group has 2 gens and 9 rels of total length 94
gap> iso := IsomorphismFpGroupByGenerators( G, gens :
>                                           method := "regular");;
#I  the image group has 2 gens and 6 rels of total length 56
gap> SetInfoLevel( InfoFpGroup, 0 );
gap> iso;
<composed isomorphism:[ [ [ -1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, \
0, 0, 1, 0 ], [ -1, -1, -1, -1, 2 ], [ -1, 0, 0, 0, 1 ] ], [ [ 0, 1, 0\
, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ], [ 1, 0, 0, 0, 0 ], [ 0\
, 0, 0, 0, 1 ] ] ]->[ F1, F2 ]>
gap> ConstituentsCompositionMapping(iso);
[ <action isomorphism>,
 [ (2,3)(4,6,10,17,7,12)(5,8,13,19,26,24)(9,15,23,35,31,43)(14,21,
       32)(16,25,38)(18,27,39,33,45,57)(20,30,36,48,28,40)(22,34,44,
       55,66,74)(29,41,51,61,60,47)(37,49,54,52,46,58)(42,53,65)(50,
```

```
      62,63,72,56,67)(59,69,77,78,64,73)(68,71,75)(76,79),
    (1,2,4,7)(3,5,9,16)(6,11,18,28)(8,14,22,30)(12,13,20,31)(15,24,
      37,50)(17,26,38,45)(19,29,42,27)(21,33,46,35)(23,36,25,39)(32,
      44,56,40)(34,47,59,70)(41,52,64,72)(43,54,65,66)(48,60)(49,61,
      71,55)(51,63,53,57)(58,68,76,62)(67,75,74,77)(73,78,80,79)
  ] -> [ F1, F2 ] ]
```

Since GAP cannot decompose elements of a matrix group into generators, the resulting isomorphism is stored as a composition of a (faithful) permutation action on vectors and a homomorphism from the permutation image to the finitely presented group. In such a situation the constituent mappings can be obtained via `ConstituentsCompositionMapping` (32.2.7) as separate GAP objects.

47.12 New Presentations and Presentations for Subgroups

`IsomorphismFpGroup` (47.11.1) is also used to compute a new finitely presented group that is isomorphic to the given subgroup of a finitely presented group. (This is typically the only method to compute with subgroups of a finitely presented group.)

Example

```
gap> f:=FreeGroup(2);;
gap> g:=f/[f.1^2,f.2^3,(f.1*f.2)^5];
<fp group on the generators [ f1, f2 ]>
gap> u:=Subgroup(g,[g.1*g.2]);
Group([ f1*f2 ])
gap> hom:=IsomorphismFpGroup(u);
[ <[ [ 1, 1 ] ]|f2^-1*f1^-1> ] -> [ F1 ]
gap> new:=Range(hom);
<fp group on the generators [ F1 ]>
gap> List(GeneratorsOfGroup(new),i->PreImagesRepresentative(hom,i));
[ <[ [ 1, 1 ] ]|f2^-1*f1^-1> ]
```

When working with such homomorphisms, some subgroup elements are expressed as extremely long words in the group generators. Therefore the underlying words of subgroup generators stored in the isomorphism (as obtained by `MappingGeneratorsImages` (40.10.2) and displayed when `View` (6.3.3)ing the homomorphism) as well as preimages under the homomorphism are stored in the form of straight line program elements (see 37.9). These will behave like ordinary words and no extra treatment should be necessary.

Example

```
gap> r:=Range(hom).1^10;
F1^10
gap> p:=PreImagesRepresentative(hom,r);
<[ [ 1, 10 ] ]|(f2^-1*f1^-1)^10>
```

If desired, it also is possible to convert these underlying words using `EvalStraightLineProgElm` (37.9.4):

Example

```
gap> r:=EvalStraightLineProgElm(UnderlyingElement(p));
(f2^-1*f1^-1)^10
gap> p:=ElementOfFpGroup(FamilyObj(p),r);
(f2^-1*f1^-1)^10
```

(If you are only interested in a finitely presented group isomorphic to the given subgroup but not in the isomorphism, you may also use the functions `PresentationViaCosetTable` (48.1.5) and `FpGroupPresentation` (48.1.4) (see 48.1).)

Homomorphisms can also be used to obtain an isomorphic finitely presented group with a (hopefully) simpler presentation.

47.12.1 IsomorphismSimplifiedFpGroup

▷ `IsomorphismSimplifiedFpGroup(`*G*`)` (attribute)

applies Tietze transformations to a copy of the presentation of the given finitely presented group *G* in order to reduce it with respect to the number of generators, the number of relators, and the relator lengths.

The operation returns an isomorphism with source *G*, range a group *H* isomorphic to *G*, so that the presentation of *H* has been simplified using Tietze transformations.

Example

```
gap> f:=FreeGroup(3);;
gap> g:=f/[f.1^2,f.2^3,(f.1*f.2)^5,f.1/f.3];
<fp group on the generators [ f1, f2, f3 ]>
gap> hom:=IsomorphismSimplifiedFpGroup(g);
[ f1, f2, f3 ] -> [ f1, f2, f1 ]
gap> Range(hom);
<fp group on the generators [ f1, f2 ]>
gap> RelatorsOfFpGroup(Range(hom));
[ f1^2, f2^3, (f1*f2)^5 ]
gap> RelatorsOfFpGroup(g);
[ f1^2, f2^3, (f1*f2)^5, f1*f3^-1 ]
```

`IsomorphismSimplifiedFpGroup` uses Tietze transformations to simplify the presentation, see 48.1.6.

47.13 Preimages under Homomorphisms from an FpGroup

For some subgroups of a finitely presented group the number of subgroup generators increases with the index of the subgroup. However often these generators are not needed at all for further calculations, but what is needed is the action of the cosets of the subgroup. This gives the image of the subgroup in a finite quotient and this finite quotient can be used to calculate normalizers, closures, intersections and so forth [Hul01].

The same applies for subgroups that are obtained as preimages under homomorphisms.

47.13.1 SubgroupOfWholeGroupByQuotientSubgroup

▷ `SubgroupOfWholeGroupByQuotientSubgroup(`*fpfam, Q, U*`)` (function)

takes a FpGroup family *fpfam*, a finitely generated group *Q* such that the fp generators of *fpfam* can be mapped by an epimorphism *phi* onto the `GeneratorsOfGroup` (39.2.4) value of *Q*, and a subgroup *U* of *Q*. It returns the subgroup of *fpfam*`!.wholeGroup` which is the full preimage of *U* under *phi*.

47.13.2 IsSubgroupOfWholeGroupByQuotientRep

▷ `IsSubgroupOfWholeGroupByQuotientRep(`*G*`)` (Representation)

is the representation for subgroups of an FpGroup, given by a quotient subgroup. The components *G*`!.quot` and *G*`!.sub` hold quotient, respectively subgroup.

47.13.3 AsSubgroupOfWholeGroupByQuotient

▷ `AsSubgroupOfWholeGroupByQuotient(`*U*`)` (attribute)

returns the same subgroup in the representation `AsSubgroupOfWholeGroupByQuotient`.

See also `SubgroupOfWholeGroupByCosetTable` (47.8.2) and `CosetTableBySubgroup` (47.6.4).

This technique is used by GAP for example to represent the derived subgroup, which is obtained from the quotient G/G'.

Example

```
gap> f:=FreeGroup(2);;g:=f/[f.1^6,f.2^6,(f.1*f.2)^6];;
gap> d:=DerivedSubgroup(g);
Group(<fp, no generators known>)
gap> Index(g,d);
36
```

47.13.4 DefiningQuotientHomomorphism

▷ `DefiningQuotientHomomorphism(`*U*`)` (function)

if *U* is a subgroup in quotient representation (`IsSubgroupOfWholeGroupByQuotientRep` (47.13.2)), this function returns the defining homomorphism from the whole group to *U*`!.quot`.

47.14 Quotient Methods

An important class of algorithms for finitely presented groups are the *quotient algorithms* which compute quotient groups of a given finitely presented group. There are algorithms for epimorphisms onto abelian groups, *p*-groups and solvable groups. (The "low index" algorithm –`LowIndexSubgroupsFpGroup` (47.10.1)– can be considered as well as an algorithm that produces permutation group quotients.)

`MaximalAbelianQuotient` (39.18.4), as defined for general groups, returns the largest abelian quotient of the given group.

Example

```
gap> f:=FreeGroup(2);;fp:=f/[f.1^6,f.2^6,(f.1*f.2)^12];
<fp group on the generators [ f1, f2 ]>
gap> hom:=MaximalAbelianQuotient(fp);
[ f1, f2 ] -> [ f1, f3 ]
gap> Size(Image(hom));
36
```

47.14.1 PQuotient

▷ PQuotient(*F, p[, c][, logord][, ctype]*) (function)

computes a factor p-group of a finitely presented group *F* in form of a quotient system. The quotient system can be converted into an epimorphism from *F* onto the p-group computed by the function EpimorphismQuotientSystem (47.14.2).

For a group G define the exponent-p central series of G inductively by $\mathscr{P}_1(G) = G$ and $\mathscr{P}_{i+1}(G) = [\mathscr{P}_i(G), G]\mathscr{P}_{i+1}(G)^p$. The factor groups modulo the terms of the lower exponent-p central series are p-groups. The group G has p-class c if $\mathscr{P}_c(G) \neq \mathscr{P}_{c+1}(G) = 1$.

The algorithm computes successive quotients modulo the terms of the exponent-p central series of *F*. If the parameter *c* is present, then the factor group modulo the $(c+1)$-th term of the exponent-p central series of *F* is returned. If *c* is not present, then the algorithm attempts to compute the largest factor p-group of *F*. In case *F* does not have a largest factor p-group, the algorithm will not terminate.

By default the algorithm computes only with factor groups of order at most p^{256}. If the parameter *logord* is present, it will compute with factor groups of order at most p^{logord}. If this parameter is specified, then the parameter *c* must also be given. The present implementation produces an error message if the order of a p-quotient exceeds p^{256} or p^{logord}, respectively. Note that the order of intermediate p-groups may be larger than the final order of a p-quotient.

The parameter *ctype* determines the type of collector that is used for computations within the factor p-group. *ctype* must either be "single" in which case a simple collector from the left is used or "combinatorial" in which case a combinatorial collector from the left is used.

47.14.2 EpimorphismQuotientSystem

▷ EpimorphismQuotientSystem(*quotsys*) (operation)

For a quotient system *quotsys* obtained from the function PQuotient (47.14.1), this operation returns an epimorphism $F \to P$ where *F* is the finitely presented group of which *quotsys* is a quotient system and *P* is a pc group isomorphic to the quotient of *F* determined by *quotsys*.

Different calls to this operation will create different groups *P*, each with its own family.

Example

```
gap> PQuotient( FreeGroup(2), 5, 10, 1024, "combinatorial" );
<5-quotient system of 5-class 10 with 520 generators>
gap> phi := EpimorphismQuotientSystem( last );
[ f1, f2 ] -> [ a1, a2 ]
gap> Collected( Factors( Size( Image( phi ) ) ) );
[ [ 5, 520 ] ]
```

47.14.3 EpimorphismPGroup

▷ EpimorphismPGroup(*fpgrp, p[, cl]*) (operation)

computes an epimorphism from the finitely presented group *fpgrp* to the largest p-group of p-class *cl* which is a quotient of *fpgrp*. If *cl* is omitted, the largest finite p-group quotient (of p-class up to 1000) is determined.

Example

```
gap> hom:=EpimorphismPGroup(fp,2);
[ f1, f2 ] -> [ a1, a2 ]
gap> Size(Image(hom));
8
gap> hom:=EpimorphismPGroup(fp,3,7);
[ f1, f2 ] -> [ a1, a2 ]
gap> Size(Image(hom));
6561
```

47.14.4 EpimorphismNilpotentQuotient

▷ EpimorphismNilpotentQuotient(*fpgrp[, n]*) (function)

returns an epimorphism on the class *n* finite nilpotent quotient of the finitely presented group *fpgrp*. If *n* is omitted, the largest finite nilpotent quotient (of p-class up to 1000) is taken.

Example

```
gap> hom:=EpimorphismNilpotentQuotient(fp,7);
[ f1, f2 ] -> [ f1*f4, f2*f5 ]
gap> Size(Image(hom));
52488
```

A related operation which is also applicable to finitely presented groups is GQuotients (40.9.4), which computes all epimorphisms from a (finitely presented) group *F* onto a given (finite) group *G*.

Example

```
gap> GQuotients(fp,Group((1,2,3),(1,2)));
[ [ f1, f2 ] -> [ (1,2), (2,3) ], [ f1, f2 ] -> [ (2,3), (1,2,3) ],
  [ f1, f2 ] -> [ (1,2,3), (2,3) ] ]
```

47.14.5 SolvableQuotient (for a f.p. group and a size)

▷ SolvableQuotient(*F, size*) (function)
▷ SolvableQuotient(*F, primes*) (function)
▷ SolvableQuotient(*F, tuples*) (function)
▷ SQ(*F, ...*) (function)

This routine calls the solvable quotient algorithm for a finitely presented group *F*. The quotient to be found can be specified in the following ways: Specifying an integer *size* finds a quotient of size up to *size* (if such large quotients exist). Specifying a list of primes in *primes* finds the largest quotient involving the given primes. Finally *tuples* can be used to prescribe a chief series.

SQ can be used as a synonym for SolvableQuotient.

47.14.6 EpimorphismSolvableQuotient

▷ EpimorphismSolvableQuotient(*F, param*) (function)

computes an epimorphism from the finitely presented group *fpgrp* to the largest solvable quotient given by *param* (specified as in SolvableQuotient (47.14.5)).

Example

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> fp := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>  f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
> f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
> f.2^-1*f.4^-1*f.2*f.4 ];;
gap> hom:=EpimorphismSolvableQuotient(fp,300);Size(Image(hom));
[ a, b, c, d ] -> [ f1*f2, f1*f2, f2*f3, f2 ]
12
gap> hom:=EpimorphismSolvableQuotient(fp,[2,3]);Size(Image(hom));
[ a, b, c, d ] -> [ f1*f2*f4, f1*f2*f6*f8, f2*f3, f2 ]
1152
```

47.14.7 LargerQuotientBySubgroupAbelianization

▷ `LargerQuotientBySubgroupAbelianization(`*hom, U*`)` (function)

Let *hom* a homomorphism from a finitely presented group G to a finite group H and $U \leq H$. This function will – if it exists – return a subgroup $S \leq G$, such that the core of S is properly contained in the kernel of *hom* as well as in V', where V is the pre-image of U under *hom*. Thus S exposes a larger quotient of G. If no such subgroup exists, *fail* is returned.

Example

```
gap> f:=FreeGroup("x","y","z");;
gap> g:=f/ParseRelators(f,"x^3=y^3=z^5=(xyx^2y^2)^2=(xz)^2=(yz^3)^2=1");
<fp group on the generators [ x, y, z ]>
gap> l:=LowIndexSubgroupsFpGroup(g,6);;
gap> List(l,IndexInWholeGroup);
[ 1, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6 ]
gap> q:=DefiningQuotientHomomorphism(l[6]);;p:=Image(q);Size(p);
Group([ (4,5,6), (1,2,3)(4,6,5), (2,4,6,3,5) ])
360
gap> s:=LargerQuotientBySubgroupAbelianization(q,SylowSubgroup(p,3));
Group(<fp, no generators known>)
gap> Size(Image(DefiningQuotientHomomorphism(s)));
193273528320
```

47.15 Abelian Invariants for Subgroups

Using variations of coset enumeration it is possible to compute the abelian invariants of a subgroup of a finitely presented group without computing a complete presentation for the subgroup in the first place. Typically, the operation `AbelianInvariants` (39.16.1) when called for subgroups should automatically take care of this, but in case you want to have further control about the methods used, the following operations might be of use.

47.15.1 AbelianInvariantsSubgroupFpGroup

▷ `AbelianInvariantsSubgroupFpGroup(`*G, H*`)` (function)

`AbelianInvariantsSubgroupFpGroup` is a synonym for `AbelianInvariantsSubgroupFpGroupRrs` (47.15.3).

47.15.2 AbelianInvariantsSubgroupFpGroupMtc

▷ `AbelianInvariantsSubgroupFpGroupMtc(G, H)` (function)

uses the Modified Todd-Coxeter method to compute the abelian invariants of a subgroup `H` of a finitely presented group `G`.

47.15.3 AbelianInvariantsSubgroupFpGroupRrs

▷ `AbelianInvariantsSubgroupFpGroupRrs(G, H)` (function)
▷ `AbelianInvariantsSubgroupFpGroupRrs(G, table)` (function)

uses the Reduced Reidemeister-Schreier method to compute the abelian invariants of a subgroup `H` of a finitely presented group `G`.

Alternatively to the subgroup `H`, its coset table `table` in `G` may be given as second argument.

47.15.4 AbelianInvariantsNormalClosureFpGroup

▷ `AbelianInvariantsNormalClosureFpGroup(G, H)` (function)

`AbelianInvariantsNormalClosureFpGroup` is a synonym for `AbelianInvariantsNormalClosureFpGroupRrs` (47.15.5).

47.15.5 AbelianInvariantsNormalClosureFpGroupRrs

▷ `AbelianInvariantsNormalClosureFpGroupRrs(G, H)` (function)

uses the Reduced Reidemeister-Schreier method to compute the abelian invariants of the normal closure of a subgroup `H` of a finitely presented group `G`. See 48.2 for details on the different strategies.

The following example shows a calculation for the Coxeter group B_1. This calculation and a similar one for B_0 have been used to prove that $B_1'/B_1'' \cong Z_2^9 \times Z^3$ and $B_0'/B_0'' \cong Z_2^{91} \times Z^{27}$ as stated in in [FJNT95, Proposition 5].

Example

```
gap> # Define the Coxeter group E1.
gap> F := FreeGroup( "x1", "x2", "x3", "x4", "x5" );
<free group on the generators [ x1, x2, x3, x4, x5 ]>
gap> x1 := F.1;; x2 := F.2;; x3 := F.3;; x4 := F.4;; x5 := F.5;;
gap> rels := [ x1^2, x2^2, x3^2, x4^2, x5^2,
>  (x1 * x3)^2, (x2 * x4)^2, (x1 * x2)^3, (x2 * x3)^3, (x3 * x4)^3,
>  (x4 * x1)^3, (x1 * x5)^3, (x2 * x5)^2, (x3 * x5)^3, (x4 * x5)^2,
>  (x1 * x2 * x3 * x4 * x3 * x2)^2 ];;
gap> E1 := F / rels;
<fp group on the generators [ x1, x2, x3, x4, x5 ]>
gap> x1 := E1.1;; x2 := E1.2;; x3 := E1.3;; x4 := E1.4;; x5 := E1.5;;
gap> # Get normal subgroup generators for B1.
gap> H := Subgroup( E1, [ x5 * x2^-1, x5 * x4^-1 ] );;
```

```
gap> # Compute the abelian invariants of B1/B1'.
gap> A := AbelianInvariantsNormalClosureFpGroup( E1, H );
[ 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> # Compute a presentation for B1.
gap> P := PresentationNormalClosure( E1, H );
<presentation with 18 gens and 46 rels of total length 132>
gap> SimplifyPresentation( P );
#I  there are 8 generators and 30 relators of total length 148
gap> B1 := FpGroupPresentation( P );
<fp group on the generators [ _x1, _x2, _x3, _x4, _x6, _x7, _x8, _x11
 ]>
gap> # Compute normal subgroup generators for B1'.
gap> gens := GeneratorsOfGroup( B1 );;
gap> numgens := Length( gens );;
gap> comms := [ ];;
gap> for i in [ 1 .. numgens - 1 ] do
>     for j in [i+1 .. numgens ] do
>         Add( comms, Comm( gens[i], gens[j] ) );
>     od;
> od;
gap> # Compute the abelian invariants of B1'/B1".
gap> K := Subgroup( B1, comms );;
gap> A := AbelianInvariantsNormalClosureFpGroup( B1, K );
[ 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

47.16 Testing Finiteness of Finitely Presented Groups

As a consequence of the algorithmic insolvabilities mentioned in the introduction to this chapter, there cannot be a general method that will test whether a given finitely presented group is actually finite.

Therefore testing the finiteness of a finitely presented group can be problematic. What GAP actually does upon a call of `IsFinite` (30.4.2) (or if it is –probably implicitly– asked for a faithful permutation representation) is to test whether it can find (via coset enumeration) a cyclic subgroup of finite index. If it can, it rewrites the presentation to this subgroup. Since the subgroup is cyclic, its size can be checked easily from the resulting presentation, the size of the whole group is the product of the index and the subgroup size. Since however no bound for the index of such a subgroup (if any exist) is known, such a test might continue unsuccessfully until memory is exhausted.

On the other hand, a couple of methods exist, that might prove that a group is infinite. Again, none is guaranteed to work in every case:

The first method is to find (for example via the low index algorithm, see `LowIndexSubgroupsFpGroup` (47.10.1)) a subgroup U such that $[U : U']$ is infinite. If U has finite index, this can be checked by `IsInfiniteAbelianizationGroup` (47.16.1).

Note that this test has been done traditionally by checking the `AbelianInvariants` (39.16.1) (see section 47.15) of U, `IsInfiniteAbelianizationGroup` (47.16.1) does a similar calculation but stops as soon as it is known whether 0 is an invariant without computing the actual values. This can be notably faster.

Another method is based on p-group quotients, see `NewmanInfinityCriterion` (47.16.2).

47.16.1 IsInfiniteAbelianizationGroup

▷ `IsInfiniteAbelianizationGroup(G)` (attribute)

returns true if the commutator factor group G/G' is infinite. This might be done without computing the full structure of the commutator factor group.

47.16.2 NewmanInfinityCriterion

▷ `NewmanInfinityCriterion(G, p)` (function)

Let G be a finitely presented group and p a prime that divides the order of the commutator factor group of G. This function applies an infinity criterion due to M. F. Newman [New90] to G. (See [Joh97, chapter 16] for a more explicit description.) It returns `true` if the criterion succeeds in proving that G is infinite and `fail` otherwise.

Note that the criterion uses the number of generators and relations in the presentation of G. Reduction of the presentation via Tietze transformations (`IsomorphismSimplifiedFpGroup` (47.12.1)) therefore might produce an isomorphic group, for which the criterion will work better.

Example

```
gap> g:=FibonacciGroup(2,9);
<fp group on the generators [ f1, f2, f3, f4, f5, f6, f7, f8, f9 ]>
gap> hom:=EpimorphismNilpotentQuotient(g,2);;
gap> k:=Kernel(hom);;
gap> Index(g,k);
152
gap> AbelianInvariants(k);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
gap> NewmanInfinityCriterion(Kernel(hom),5);
true
```

This proves that the subgroup `k` (and thus the whole group `g`) is infinite. (This is the original example from [New90].)

Chapter 48

Presentations and Tietze Transformations

A finite presentation describes a group, but usually there is a multitude of presentations that describe isomorphic groups. Therefore a presentation in GAP is different from a finitely presented group though there are ways to translate between both.

An important feature of presentations is that they can be modified (see sections 48.5 to 48.8).

If you only want to get new presentations for subgroups of a finitely presented group (and do not want to manipulate presentations yourself), chances are that the operation `IsomorphismFpGroup` (47.11.1) already does what you want (see 47.12).

48.1 Creating Presentations

Most of the functions creating presentations and all functions performing Tietze transformations on them sort the relators by increasing lengths. The function `PresentationFpGroup` (48.1.1) is an exception because it is intended to reflect the relators that were used to define the involved f. p. group. You may use the command `TzSort` (48.1.2) to sort the presentation.

48.1.1 PresentationFpGroup

▷ `PresentationFpGroup(`*`G[, printlevel]`*`)` (function)

creates a presentation, i. e., a Tietze object, for the given finitely presented group *G*. This presentation will be exactly as the presentation of *G* and *no* initial Tietze transformations are applied to it.

The optional *printlevel* parameter can be used to restrict or to extend the amount of output provided by Tietze transformation commands when being applied to the created presentation. The default value 1 is designed for interactive use and implies explicit messages to be displayed by most of these commands. A *printlevel* value of 0 will suppress these messages, whereas a *printlevel* value of 2 will enforce some additional output.

Example

```
gap> f := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> g := f / [ f.1^3, f.2^2, (f.1*f.2)^3 ];
<fp group on the generators [ a, b ]>
gap> p := PresentationFpGroup( g );
<presentation with 2 gens and 3 rels of total length 11>
```

48.1.2 TzSort

▷ `TzSort(P)` (function)

sorts the relators of the given presentation `P` by increasing lengths. There is no particular ordering defined for the relators of equal length. Note that `TzSort` does not return a new object. It changes the given presentation.

48.1.3 GeneratorsOfPresentation

▷ `GeneratorsOfPresentation(P)` (function)

returns a list of free generators that is a shallow copy (see `ShallowCopy` (12.7.1)) of the current generators of the presentation `P`.

48.1.4 FpGroupPresentation

▷ `FpGroupPresentation(P[, nam])` (function)

constructs an f. p. group as defined by the given Tietze presentation `P`.

Example

```
gap> h := FpGroupPresentation( p );
<fp group on the generators [ a, b ]>
gap> h = g;
false
```

48.1.5 PresentationViaCosetTable

▷ `PresentationViaCosetTable(G[, F, words])` (function)

constructs a presentation for a given concrete finite group. It applies the relations finding algorithm which has been described in [Can73] and [Neu82]. It automatically applies Tietze transformations to the presentation found.

If only a group `G` has been specified, the single stage algorithm is applied.

The operation `IsomorphismFpGroup` (47.11.1) in contrast uses a multiple-stage algorithm using a chief series and stabilizer chains. It usually should be used rather than `PresentationViaCosetTable`. (It does not apply Tietze transformations automatically.)

If the two stage algorithm is to be used, `PresentationViaCosetTable` expects a subgroup `H` of `G` to be provided in form of two additional arguments `F` and `words`, where `F` is a free group with the same number of generators as `G`, and `words` is a list of words in the generators of `F` which supply a list of generators of `H` if they are evaluated as words in the corresponding generators of `G`.

Example

```
gap> G := GeneralLinearGroup( 2, 7 );
GL(2,7)
gap> GeneratorsOfGroup( G );
[ [ [ Z(7), 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ],
  [ [ Z(7)^3, Z(7)^0 ], [ Z(7)^3, 0*Z(7) ] ] ]
gap> Size( G );
2016
```

```
gap> P := PresentationViaCosetTable( G );
<presentation with 2 gens and 5 rels of total length 46>
gap> TzPrintRelators( P );
#I  1. f2^3
#I  2. f1^6
#I  3. (f1^-1*f2^-1)^6
#I  4. f1*f2*f1^-1*f2^-1*f1*f2^-1*f1^-1*f2*f1*f2^-1*f1^-1*f2^-1
#I  5. f1^-3*f2*f1*f2*(f1^-1*f2^-1)^2*f1^-2*f2
```

The two stage algorithm saves an essential amount of space by constructing two coset tables of lengths $|H|$ and $|G|/|H|$ instead of just one coset table of length $|G|$. The next example shows an application of this option in the case of a subgroup of size 7920 and index 12 in a permutation group of size 95040.

Example

```
gap> M12 := Group( [ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6),
> (1,12)(2,11)(3,6)(4,8)(5,9)(7,10) ], () );;
gap> F := FreeGroup( "a", "b", "c" );
<free group on the generators [ a, b, c ]>
gap> words := [ F.1, F.2 ];
[ a, b ]
gap> P := PresentationViaCosetTable( M12, F, words );
<presentation with 3 gens and 10 rels of total length 97>
gap> G := FpGroupPresentation( P );
<fp group on the generators [ a, b, c ]>
gap> RelatorsOfFpGroup( G );
[ c^2, b^4, (a*c)^3, (a*b^-2)^3, a^11,
  a^2*b*a^-2*b^-1*(b^-1*a)^2*a*b^-1, (a*(b*a^-1)^2*b^-1)^2,
  a^2*b*a^2*b^-2*a^-1*b*(a^-1*b^-1)^2,
  (a*b)^2*a^2*b^-1*a^-1*b^-1*a*c*b*c, a^2*(a^2*b)^2*a^-2*c*a*b*a^-1*c
 ]
```

Before it is returned, the resulting presentation is being simplified by appropriate calls of the function `SimplifyPresentation` (48.6.2) (see 48.6), but without allowing any eliminations of generators. This restriction guarantees that we get a bijection between the list of generators of `G` and the list of generators in the presentation. Hence, if the generators of `G` are redundant and if you don't care for the bijection, you may get a shorter presentation by calling the function `SimplifyPresentation` (48.6.2), now without this restriction, once more yourself.

Example

```
gap> H := Group(
> [ (2,5,3), (2,7,5), (1,8,4), (1,8,6), (4,8,6), (3,5,7) ], () );;
gap> P := PresentationViaCosetTable( H );
<presentation with 6 gens and 12 rels of total length 42>
gap> SimplifyPresentation( P );
#I  there are 4 generators and 10 relators of total length 36
```

If you apply the function `FpGroupPresentation` (48.1.4) to the resulting presentation you will get a finitely presented group isomorphic to `G`. Note, however, that the function `IsomorphismFpGroup` (47.11.1) is recommended for this purpose.

48.1.6 SimplifiedFpGroup

▷ SimplifiedFpGroup(*G*) (function)

applies Tietze transformations to a copy of the presentation of the given finitely presented group *G* in order to reduce it with respect to the number of generators, the number of relators, and the relator lengths.

SimplifiedFpGroup returns a group isomorphic to the given one with a presentation which has been tried to simplify via Tietze transformations.

If the connection to the original group is important, then the operation IsomorphismSimplifiedFpGroup (47.12.1) should be used instead.

Example

```
gap> F6 := FreeGroup( 6, "G" );;
gap> G := F6 / [ F6.1^2, F6.2^2, F6.4*F6.6^-1, F6.5^2, F6.6^2,
> F6.1*F6.2^-1*F6.3, F6.1*F6.5*F6.3^-1, F6.2*F6.4^-1*F6.3,
> F6.3*F6.4*F6.5^-1, F6.1*F6.6*F6.3^-2, F6.3^4 ];;
gap> H := SimplifiedFpGroup( G );
<fp group on the generators [ G1, G3 ]>
gap> RelatorsOfFpGroup( H );
[ G1^2, (G1*G3^-1)^2, G3^4 ]
```

In fact, the command

Example

```
H := SimplifiedFpGroup( G );
```

is an abbreviation of the command sequence

Example

```
P := PresentationFpGroup( G, 0 );;
SimplifyPresentation( P );
H := FpGroupPresentation( P );
```

which applies a rather simple-minded strategy of Tietze transformations to the intermediate presentation *P*. If, for some concrete group, the resulting presentation is unsatisfying, then you should try a more sophisticated, interactive use of the available Tietze transformation commands (see 48.6).

48.2 Subgroup Presentations

48.2.1 PresentationSubgroup

▷ PresentationSubgroup(*G, H[, string]*) (function)

PresentationSubgroup is a synonym for PresentationSubgroupRrs (48.2.2).

48.2.2 PresentationSubgroupRrs

▷ PresentationSubgroupRrs(*G, H[, string]*) (function)
▷ PresentationSubgroupRrs(*G, table[, string]*) (function)

uses the Reduced Reidemeister-Schreier method to compute a presentation *P*, say, for a subgroup *H* of a finitely presented group *G*. The generators in the resulting presentation will be named *string*1, *string*2, ..., the default string is "_x". You may access the *i*-th of these generators by *P*!.*i*.

Alternatively to the subgroup *H*, its coset table *table* in *G* may be given as second argument.

Example

```
gap> f := FreeGroup( "a", "b" );;
gap> g := f / [ f.1^2, f.2^3, (f.1*f.2)^5 ];
<fp group on the generators [ a, b ]>
gap> g1 := Size( g );
60
gap> u := Subgroup( g, [ g.1, g.1^g.2 ] );
Group([ a, b^-1*a*b ])
gap> p := PresentationSubgroup( g, u, "g" );
<presentation with 3 gens and 4 rels of total length 12>
gap> gens := GeneratorsOfPresentation( p );
[ g1, g2, g3 ]
gap> TzPrintRelators( p );
#I  1. g1^2
#I  2. g2^2
#I  3. g3*g2*g1
#I  4. g3^5
```

Note that you cannot call the generators by their names. These names are not variables, but just display figures. So, if you want to access the generators by their names, you first will have to introduce the respective variables and to assign the generators to them.

Example

```
gap> gens[1] = g1;
false
gap> g1;
60
gap> g1 := gens[1];; g2 := gens[2];; g3 := gens[3];;
gap> g1;
g1
```

The Reduced Reidemeister-Schreier algorithm is a modification of the Reidemeister-Schreier algorithm of George Havas [Hav74]. It was proposed by Joachim Neubüser and first implemented in 1986 by Andrea Lucchini and Volkmar Felsch in the SPAS system [SPA89]. Like the Reidemeister-Schreier algorithm of George Havas, it needs only the presentation of *G* and a coset table of *H* in *G* to construct a presentation of *H*.

Whenever you call the command PresentationSubgroupRrs, it first obtains a coset table of *H* in *G* if not given. Next, a set of generators of *H* is determined by reconstructing the coset table and introducing in that process as many Schreier generators of *H* in *G* as are needed to do a Felsch strategy coset enumeration without any coincidences. (In general, though containing redundant generators, this set will be much smaller than the set of all Schreier generators. That is why we call the method the *Reduced* Reidemeister-Schreier.)

After having constructed this set of *primary subgroup generators*, say, the coset table is extended to an *augmented coset table* which describes the action of the group generators on coset representatives, i.e., on elements instead of cosets. For this purpose, suitable words in the (primary) subgroup generators have to be associated to the coset table entries. In order to keep the lengths of these words

short, additional *secondary subgroup generators* are introduced as abbreviations of subwords. Their number may be large.

Finally, a Reidemeister rewriting process is used to get defining relators for *H* from the relators of *G*. As the resulting presentation of *H* is a presentation on primary *and* secondary generators, in general you will have to simplify it by appropriate Tietze transformations (see 48.6) or by the command DecodeTree (48.10.1) before you can use it. Therefore it is returned in the form of a presentation, *P* say.

Compared with the Modified Todd-Coxeter method described below, the Reduced Reidemeister-Schreier method (as well as Havas' original Reidemeister-Schreier program) has the advantage that it does not require generators of *H* to be given if a coset table of *H* in *G* is known. This provides a possibility to compute a presentation of the normal closure of a given subgroup (see PresentationNormalClosureRrs (48.2.5)).

For certain applications you may be interested in getting not only just a presentation for *H*, but also a relation between the involved generators of *H* and the generators of *G*. The subgroup generators in the presentation are sorted such that the primary generators precede the secondary ones. Moreover, for each secondary subgroup generator there is a relator in the presentation which expresses this generator as a word in preceding ones. Hence, all we need in addition is a list of words in the generators of *G* which express the primary subgroup generators. In fact, such a list is provided in the attribute PrimaryGeneratorWords (48.2.3) of the resulting presentation.

48.2.3 PrimaryGeneratorWords

▷ PrimaryGeneratorWords(*P*) (attribute)

is an attribute of the presentation *P* which holds a list of words in the associated group generators (of the underlying free group) which express the primary subgroup generators of *P*.

Example

```
gap> PrimaryGeneratorWords( p );
[ a, b^-1*a*b ]
```

48.2.4 PresentationSubgroupMtc

▷ PresentationSubgroupMtc(*G, H[, string][, print, level]*) (function)

uses the Modified Todd-Coxeter coset representative enumeration method to compute a presentation *P*, say, for a subgroup *H* of a finitely presented group *G*. The presentation returned is in generators corresponding to the generators of *H*. The generators in the resulting presentation will be named *string*1, *string*2, ..., the default string is "_x". You may access the *i*-th of these generators by *P*!.*i*.

The default print level is 1. If the print level is set to 0, then the printout of the implicitly called function DecodeTree (48.10.1) will be suppressed.

Example

```
gap> p := PresentationSubgroupMtc( g, u );
#I  there are 3 generators and 4 relators of total length 12
#I  there are 2 generators and 3 relators of total length 14
<presentation with 2 gens and 3 rels of total length 14>
```

The so called Modified Todd-Coxeter method was proposed, in slightly different forms, by Nathan S. Mendelsohn and William O. J. Moser in 1966. Moser's method was proved in [BC76]. It has been generalized to cover a broad spectrum of different versions (see the survey [Neu82]).

The *Modified Todd-Coxeter* method performs an enumeration of coset representatives. It proceeds like an ordinary coset enumeration (see 47.6), but as the product of a coset representative by a group generator or its inverse need not be a coset representative itself, the Modified Todd-Coxeter has to store a kind of correction element for each coset table entry. Hence it builds up a so called *augmented coset table* of `H` in `G` consisting of the ordinary coset table and a second table in parallel which contains the associated subgroup elements.

Theoretically, these subgroup elements could be expressed as words in the given generators of `H`, but in general these words tend to become unmanageable because of their enormous lengths. Therefore, a highly redundant list of subgroup generators is built up starting from the given ("primary") generators of `H` and adding additional ("secondary") generators which are defined as abbreviations of suitable words of length two in the preceding generators such that each of the subgroup elements in the augmented coset table can be expressed as a word of length at most one in the resulting (primary *and* secondary) subgroup generators.

Then a rewriting process (which is essentially a kind of Reidemeister rewriting process) is used to get relators for `H` from the defining relators of `G`.

The resulting presentation involves all the primary, but not all the secondary generators of `H`. In fact, it contains only those secondary generators which explicitly occur in the augmented coset table. If we extended this presentation by those secondary generators which are not yet contained in it as additional generators, and by the definitions of all secondary generators as additional relators, we would get a presentation of `H`, but, in general, we would end up with a large number of generators and relators.

On the other hand, if we avoid this extension, the current presentation will not necessarily define `H` although we have used the same rewriting process which in the case of the `PresentationSubgroupRrs` (48.2.2) command computes a defining set of relators for `H` from an augmented coset table and defining relators of `G`. The different behaviour here is caused by the fact that coincidences may have occurred in the Modified Todd-Coxeter coset enumeration.

To overcome this problem without extending the presentation by all secondary generators, the `PresentationSubgroupMtc` command applies the so called *decoding tree* algorithm which provides a more economical approach. The reader is strongly recommended to carefully read section 48.10 where this algorithm is described in more detail. Here we will only mention that this procedure may add a lot of intermediate generators and relators (and even change the isomorphism type) in a process which in fact eliminates all secondary generators from the presentation and hence finally provides a presentation of `H` on the primary, i.e., the originally given, generators of `H`. This is a remarkable advantage of the command `PresentationSubgroupMtc` compared to the command `PresentationSubgroupRrs` (48.2.2). But note that, for some particular subgroup `H`, the Reduced Reidemeister-Schreier method might quite well produce a more concise presentation.

The resulting presentation is returned in the form of a presentation, *P* say.

As the function `PresentationSubgroupRrs` (48.2.2) described above (see there for details), the function `PresentationSubgroupMtc` returns a list of the primary subgroup generators of `H` in the attribute `PrimaryGeneratorWords` (48.2.3) of *P*. In fact, this list is not very exciting here because it is just a shallow copy of the value of `GeneratorsOfPresentation` (48.1.3) of `H`, however it is needed to guarantee a certain consistency between the results of the different functions for computing subgroup presentations.

Though the decoding tree routine already involves a lot of Tietze transformations, we recommend

that you try to further simplify the resulting presentation by appropriate Tietze transformations (see 48.6).

48.2.5 PresentationNormalClosureRrs

▷ `PresentationNormalClosureRrs(`*G, H[, string]*`)` (function)

uses the Reduced Reidemeister-Schreier method to compute a presentation P, say, for the normal closure of a subgroup *H* of a finitely presented group *G*. The generators in the resulting presentation will be named *string*`1`, *string*`2`, ..., the default string is `"_x"`. You may access the i-th of these generators by P`!.`i.

48.2.6 PresentationNormalClosure

▷ `PresentationNormalClosure(`*G, H[, string]*`)` (function)

`PresentationNormalClosure` is a synonym for `PresentationNormalClosureRrs` (48.2.5).

48.3 Relators in a Presentation

In order to speed up the Tietze transformation routines, each relator in a presentation is internally represented by a list of positive or negative generator numbers, i.e., each factor of the proper GAP word is represented by the position number of the corresponding generator with respect to the current list of generators, or by the respective negative number, if the factor is the inverse of a generator. Note that the numbering of the generators in Tietze words is always relative to a generator list and bears no relation to the internal numbering of generators in a family of associative words.

48.3.1 TietzeWordAbstractWord

▷ `TietzeWordAbstractWord(`*word, fgens*`)` (function)

assumes *fgens* to be a list of free group generators and *word* to be an abstract word in these generators. It converts *word* into a Tietze word, i. e., a list of positive or negative generator numbers.

This function simply calls `LetterRepAssocWord` (37.6.8).

48.3.2 AbstractWordTietzeWord

▷ `AbstractWordTietzeWord(`*word, fgens*`)` (function)

assumes *fgens* to be a list of free group generators and *word* to be a Tietze word in these generators, i. e., a list of positive or negative generator numbers. It converts *word* to an abstract word.

This function simply calls `AssocWordByLetterRep` (37.6.9).

Example

```
gap> F := FreeGroup( "a", "b", "c" ,"d");
<free group on the generators [ a, b, c, d ]>
gap> tzword := TietzeWordAbstractWord(
> Comm(F.4,F.2) * (F.3^2 * F.2)^-1, GeneratorsOfGroup( F ){[2,3,4]} );
[ -3, -1, 3, -2, -2 ]
```

```
gap> AbstractWordTietzeWord( tzword, GeneratorsOfGroup( F ){[2,3,4]} );
d^-1*b^-1*d*c^-2
```

48.4 Printing Presentations

Whenever you create a presentation *P*, say, or assign it to a variable, GAP will respond by printing *P*. However, as *P* may contain a lot of generators and many relators of large length, it would be annoying if the standard print facilities displayed all this information in detail. So they restrict the printout to just one line of text containing the number of generators, the number of relators, and the total length of all relators of *P*. As compensation, GAP offers some special print commands which display various details of a presentation. Note that there is also a function `TzPrintOptions` (48.11.2). It is described in Section 48.11.

48.4.1 TzPrintGenerators

▷ `TzPrintGenerators(`*P[, list]*`)` (function)

prints the generators of the given Tietze presentation *P* together with the number of their occurrences in the relators. The optional second argument can be used to specify the numbers of the generators to be printed. Default: all generators are printed.

Example

```
gap> G := Group( [ (1,2,3,4,5), (2,3,5,4), (1,6)(3,4) ], () );
Group([ (1,2,3,4,5), (2,3,5,4), (1,6)(3,4) ])
gap> P := PresentationViaCosetTable( G );
<presentation with 3 gens and 6 rels of total length 28>
gap> TzPrintGenerators( P );
#I  1.  f1   11 occurrences
#I  2.  f2   10 occurrences
#I  3.  f3   7 occurrences   involution
```

48.4.2 TzPrintRelators

▷ `TzPrintRelators(`*P[, list]*`)` (function)

prints the relators of the given Tietze presentation *P*. The optional second argument *list* can be used to specify the numbers of the relators to be printed. Default: all relators are printed.

Example

```
gap> TzPrintRelators( P );
#I  1. f3^2
#I  2. f2^4
#I  3. (f2^-1*f3)^2
#I  4. f1^5
#I  5. f1^2*f2*f1*f2^-1
#I  6. f1^-1*f3*f1*f3*f1^-1*f2^2*f3
```

48.4.3 TzPrintLengths

▷ `TzPrintLengths(`*P*`)` (function)

prints just a list of all relator lengths of the given presentation *P*.

Example

```
gap> TzPrintLengths( P );
[ 2, 4, 4, 5, 5, 8 ]
```

48.4.4 TzPrintStatus

▷ `TzPrintStatus(`*P[, norepeat]*`)` (function)

is an internal function which is used by the Tietze transformation routines to print the number of generators, the number of relators, and the total length of all relators in the given Tietze presentation *P*. If *norepeat* is specified as `true`, the printing is suppressed if none of the three values has changed since the last call.

Example

```
gap> TzPrintStatus( P );
#I  there are 3 generators and 6 relators of total length 28
```

48.4.5 TzPrintPresentation

▷ `TzPrintPresentation(`*P*`)` (function)

prints the generators and the relators of a Tietze presentation. In fact, it is an abbreviation for the successive call of the three commands `TzPrintGenerators` (48.4.1), `TzPrintRelators` (48.4.2), and `TzPrintStatus` (48.4.4), each with the presentation *P* as only argument.

48.4.6 TzPrint

▷ `TzPrint(`*P[, list]*`)` (function)

prints the current generators of the given presentation *P*, and prints the relators of *P* as Tietze words (without converting them back to abstract words as the functions `TzPrintRelators` (48.4.2) and `TzPrintPresentation` (48.4.5) do). The optional second argument can be used to specify the numbers of the relators to be printed. Default: all relators are printed.

Example

```
gap> TzPrint( P );
#I  generators: [ f1, f2, f3 ]
#I  relators:
#I  1.  2  [ 3, 3 ]
#I  2.  4  [ 2, 2, 2, 2 ]
#I  3.  4  [ -2, 3, -2, 3 ]
#I  4.  5  [ 1, 1, 1, 1, 1 ]
#I  5.  5  [ 1, 1, 2, 1, -2 ]
#I  6.  8  [ -1, 3, 1, 3, -1, 2, 2, 3 ]
```

48.4.7 TzPrintPairs

▷ `TzPrintPairs(`*P[, n]*`)` (function)

prints the *n* most often occurring relator subwords of the form *ab*, where *a* and *b* are different generators or inverses of generators, together with the number of their occurrences. The default value of *n* is 10. A value *n* = 0 is interpreted as `infinity` (18.2.1).

The function `TzPrintPairs` is useful in the context of Tietze transformations which introduce new generators by substituting words in the current generators (see 48.8). It gives some evidence for an appropriate choice of a word of length 2 to be substituted.

Example

```
gap> TzPrintPairs( P, 3 );
#I  1.  3  occurrences of  f2 * f3
#I  2.  2  occurrences of  f2^-1 * f3
#I  3.  2  occurrences of  f1 * f3
```

48.5 Changing Presentations

The functions described in this section may be used to change a presentation. Note, however, that in general they do not perform Tietze transformations because they change or may change the isomorphism type of the group defined by the presentation.

48.5.1 AddGenerator

▷ `AddGenerator(`*P*`)` (function)

extends the presentation *P* by a new generator.

Let *i* be the smallest positive integer which has not yet been used as a generator number in the given presentation. `AddGenerator` defines a new abstract generator x_i with the name `"_x`*i*`"` and adds it to the list of generators of *P*.

You may access the generator x_i by typing *P*`!.`*i*. However, this is only practicable if you are running an interactive job because you have to know the value of *i*. Hence the proper way to access the new generator is to write `GeneratorsOfPresentation(P)[Length(GeneratorsOfPresentation(P))]`.

Example

```
gap> G := PerfectGroup( 120 );;
gap> H := Subgroup( G, [ G.1^G.2, G.3 ] );;
gap> P := PresentationSubgroup( G, H );
<presentation with 4 gens and 7 rels of total length 21>
gap> AddGenerator( P );
#I  now the presentation has 5 generators, the new generator is _x7
gap> gens := GeneratorsOfPresentation( P );
[ _x1, _x2, _x4, _x5, _x7 ]
gap> gen := gens[Length( gens )];
_x7
gap> gen = P!.7;
true
```

48.5.2 TzNewGenerator

▷ `TzNewGenerator(P)` (function)

is an internal function which defines a new abstract generator and adds it to the presentation `P`. It is called by `AddGenerator` (48.5.1) and by several Tietze transformation commands. As it does not know which global lists have to be kept consistent, you should not call it. Instead, you should call the function `AddGenerator` (48.5.1), if needed.

48.5.3 AddRelator

▷ `AddRelator(P, word)` (function)

adds the relator *word* to the presentation `P`, probably changing the group defined by `P`. *word* must be an abstract word in the generators of `P`.

48.5.4 RemoveRelator

▷ `RemoveRelator(P, n)` (function)

removes the `n`-th relator from the presentation `P`, probably changing the group defined by `P`.

48.6 Tietze Transformations

The commands in this section can be used to modify a presentation by Tietze transformations.

In general, the aim of such modifications will be to *simplify* the given presentation, i.e., to reduce the number of generators and the number of relators without increasing too much the sum of all relator lengths which we will call the *total length* of the presentation. Depending on the concrete presentation under investigation one may end up with a nice, short presentation or with a very huge one.

Unfortunately there is no algorithm which could be applied to find the shortest presentation which can be obtained by Tietze transformations from a given one. Therefore, what GAP offers are some lower-level Tietze transformation commands and, in addition, some higher-level commands which apply the lower-level ones in a kind of default strategy which of course cannot be the optimal choice for all presentations.

The design of these commands follows closely the concept of the ANU Tietze transformation program [Hav69] and its later revisions (see [HKRR84], [Rob88]).

48.6.1 TzGo

▷ `TzGo(P[, silent])` (function)

automatically performs suitable Tietze transformations of the given presentation `P`. It is perhaps the most convenient one among the interactive Tietze transformation commands. It offers a kind of default strategy which, in general, saves you from explicitly calling the lower-level commands it involves.

If *silent* is specified as `true`, the printing of the status line by `TzGo` is suppressed if the Tietze option `printLevel` (see 48.11) has a value less than 2.

48.6.2 SimplifyPresentation

▷ SimplifyPresentation(*P*) (function)

SimplifyPresentation is a synonym for TzGo (48.6.1).

Example

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1^9, F2.2^2, (F2.1*F2.2)^4, (F2.1^2*F2.2)^3 ];;
gap> a := G.1;; b := G.2;;
gap> H := Subgroup( G, [ (a*b)^2, (a^-1*b)^2 ] );;
gap> Index( G, H );
408
gap> P := PresentationSubgroup( G, H );
<presentation with 8 gens and 36 rels of total length 111>
gap> PrimaryGeneratorWords( P );
[ b, a*b*a ]
gap> TzOptions( P ).protected := 2;
2
gap> TzOptions( P ).printLevel := 2;
2
gap> SimplifyPresentation( P );
#I  eliminating _x7 = _x5^-1
#I  eliminating _x5 = _x4
#I  eliminating _x18 = _x3
#I  eliminating _x8 = _x3
#I  there are 4 generators and 8 relators of total length 21
#I  there are 4 generators and 7 relators of total length 18
#I  eliminating _x4 = _x3^-1*_x2^-1
#I  eliminating _x3 = _x2*_x1^-1
#I  there are 2 generators and 4 relators of total length 14
#I  there are 2 generators and 4 relators of total length 13
#I  there are 2 generators and 3 relators of total length 9
gap> TzPrintRelators( P );
#I  1. _x1^2
#I  2. _x2^3
#I  3. (_x2*_x1)^2
```

Roughly speaking, TzGo (48.6.1) consists of a loop over a procedure which involves two phases: In the *search phase* it calls TzSearch (48.7.2) and TzSearchEqual (48.7.3) described below which try to reduce the relator lengths by substituting common subwords of relators, in the *elimination phase* it calls the command TzEliminate (48.7.1) described below (or, more precisely, a subroutine of TzEliminate (48.7.1) in order to save some administrative overhead) which tries to eliminate generators that can be expressed as words in the remaining generators.

If TzGo (48.6.1) succeeds in reducing the number of generators, the number of relators, or the total length of all relators, it displays the new status before returning (provided that you did not set the print level to zero). However, it does not provide any output if all these three values have remained unchanged, even if the command TzSearchEqual (48.7.3) involved has changed the presentation such that another call of TzGo (48.6.1) might provide further progress. Hence, in such a case it makes sense to repeat the call of the command for several times (or to call the command TzGoGo (48.6.3) instead).

48.6.3 TzGoGo

▷ TzGoGo(*P*) (function)

calls the command TzGo (48.6.1) again and again until it does not reduce the presentation any more.

The result of the Tietze transformations can be affected substantially by the options parameters (see 48.11). To demonstrate the effect of the eliminationsLimit parameter, we will give an example in which we handle a subgroup of index 240 in a group of order 40320 given by a presentation due to B. H. Neumann. First we construct a presentation of the subgroup, and then we apply to it the command TzGoGo for different values of the parameter eliminationsLimit (including the default value 100). In fact, we also alter the printLevel parameter, but this is only done in order to suppress most of the output. In all cases the resulting presentations cannot be improved any more by applying the command TzGoGo again, i.e., they are the best results which we can get without substituting new generators.

Example

```
gap> F3 := FreeGroup( "a", "b", "c" );;
gap> G := F3 / [ F3.1^3, F3.2^3, F3.3^3, (F3.1*F3.2)^5,
> (F3.1^-1*F3.2)^5, (F3.1*F3.3)^4, (F3.1*F3.3^-1)^4,
> F3.1*F3.2^-1*F3.1*F3.2*F3.3^-1*F3.1*F3.3*F3.1*F3.3^-1,
> (F3.2*F3.3)^3, (F3.2^-1*F3.3)^4 ];;
gap> a := G.1;; b := G.2;; c := G.3;;
gap> H := Subgroup( G, [ a, c ] );;
gap> for i in [ 61, 62, 63, 90, 97 ] do
> Pi := PresentationSubgroup( G, H );
> TzOptions( Pi ).eliminationsLimit := i;
> Print("#I eliminationsLimit set to ",i,"\n");
> TzOptions( Pi ).printLevel := 0;
> TzGoGo( Pi );
> TzPrintStatus( Pi );
> od;
#I eliminationsLimit set to 61
#I  there are 2 generators and 104 relators of total length 7012
#I eliminationsLimit set to 62
#I  there are 2 generators and 7 relators of total length 56
#I eliminationsLimit set to 63
#I  there are 3 generators and 97 relators of total length 5998
#I eliminationsLimit set to 90
#I  there are 3 generators and 11 relators of total length 68
#I eliminationsLimit set to 97
#I  there are 4 generators and 109 relators of total length 3813
```

Similarly, we demonstrate the influence of the saveLimit parameter by just continuing the preceding example for some different values of the saveLimit parameter (including its default value 10), but without changing the eliminationsLimit parameter which keeps its default value 100.

Example

```
gap> for i in [ 7 .. 11 ] do
> Pi := PresentationSubgroup( G, H );
> TzOptions( Pi ).saveLimit := i;
> Print( "#I saveLimit set to ", i, "\n" );
```

```
> TzOptions( Pi ).printLevel := 0;
> TzGoGo( Pi );
> TzPrintStatus( Pi );
> od;
#I saveLimit set to 7
#I  there are 3 generators and 99 relators of total length 2713
#I saveLimit set to 8
#I  there are 2 generators and 103 relators of total length 11982
#I saveLimit set to 9
#I  there are 2 generators and 6 relators of total length 41
#I saveLimit set to 10
#I  there are 3 generators and 118 relators of total length 13713
#I saveLimit set to 11
#I  there are 3 generators and 11 relators of total length 58
```

48.7 Elementary Tietze Transformations

48.7.1 TzEliminate

▷ `TzEliminate(`*P[, gen]*`)` (function)
▷ `TzEliminate(`*P[, n]*`)` (function)

tries to eliminate a generator from a presentation *P* via Tietze transformations.

Any relator which contains some generator just once can be used to substitute that generator by a word in the remaining generators. If such generators and relators exist, then `TzEliminate` chooses a generator for which the product of its number of occurrences and the length of the substituting word is minimal, and then it eliminates this generator from the presentation, provided that the resulting total length of the relators does not exceed the associated Tietze option parameter `spaceLimit` (see 48.11). The default value of that parameter is `infinity` (18.2.1), but you may alter it appropriately.

If a generator *gen* has been specified, `TzEliminate` eliminates it if possible, i. e. if there is a relator in which *gen* occurs just once. If no second argument has been specified, `TzEliminate` eliminates some appropriate generator if possible and if the resulting total length of the relators will not exceed the Tietze options parameter `lengthLimit`.

If an integer *n* has been specified, `TzEliminate` tries to eliminate up to *n* generators. Note that the calls `TzEliminate(`*P*`)` and `TzEliminate(`*P*`,1)` are equivalent.

48.7.2 TzSearch

▷ `TzSearch(`*P*`)` (function)

searches for relator subwords which, in some relator, have a complement of shorter length and which occur in other relators, too, and uses them to reduce these other relators.

The idea is to find pairs of relators r_1 and r_2 of length l_1 and l_2, respectively, such that $l_1 \leq l_2$ and r_1 and r_2 coincide (possibly after inverting or conjugating one of them) in some maximal subword w, say, of length greater than $l_1/2$, and then to substitute each copy of w in r_2 by the inverse complement of w in r_1.

Two of the Tietze option parameters which are listed in section 48.11 may strongly influence the performance and the results of the command `TzSearch`. These are the parameters `saveLimit` and

searchSimultaneous. The first of them has the following effect:

When TzSearch has finished its main loop over all relators, then, in general, there are relators which have changed and hence should be handled again in another run through the whole procedure. However, experience shows that it really does not pay to continue this way until no more relators change. Therefore, TzSearch starts a new loop only if the loop just finished has reduced the total length of the relators by at least saveLimit per cent.

The default value of saveLimit is 10 per cent.

To understand the effect of the option searchSimultaneous, we have to look in more detail at how TzSearch proceeds:

First, it sorts the list of relators by increasing lengths. Then it performs a loop over this list. In each step of this loop, the current relator is treated as *short relator* r_1, and a subroutine is called which loops over the succeeding relators, treating them as *long relators* r_2 and performing the respective comparisons and substitutions.

As this subroutine performs a very expensive process, it has been implemented as a C routine in the GAP kernel. For the given relator r_1 of length l_1, say, it first determines the *minimal match length* l which is $l_1/2+1$, if l_1 is even, or $(l_1+1)/2$, otherwise. Then it builds up a hash list for all subwords of length l occurring in the conjugates of r_1 or r_1^{-1}, and finally it loops over all long relators r_2 and compares the hash values of their subwords of length l against this list. A comparison of subwords which is much more expensive is only done if a hash match has been found.

To improve the efficiency of this process we allow the subroutine to handle several short relators simultaneously provided that they have the same minimal match length. If, for example, it handles n short relators simultaneously, then you save $n-1$ loops over the long relators r_2, but you pay for it by additional fruitless subword comparisons. In general, you will not get the best performance by always choosing the maximal possible number of short relators to be handled simultaneously. In fact, the optimal choice of the number will depend on the concrete presentation under investigation. You can use the parameter searchSimultaneous to prescribe an upper bound for the number of short relators to be handled simultaneously.

The default value of searchSimultaneous is 20.

48.7.3 TzSearchEqual

▷ TzSearchEqual(*P*) (function)

searches for Tietze relator subwords which, in some relator, have a complement of equal length and which occur in other relators, too, and uses them to modify these other relators.

The idea is to find pairs of relators r_1 and r_2 of length l_1 and l_2, respectively, such that l_1 is even, $l_1 \leq l_2$, and r_1 and r_2 coincide (possibly after inverting or conjugating one of them) in some maximal subword w, say, of length at least $l_1/2$. Let l be the length of w. Then, if $l > l_1/2$, the pair is handled as in TzSearch (48.7.2). Otherwise, if $l = l_1/2$, then TzSearchEqual substitutes each copy of w in r_2 by the inverse complement of w in r_1.

The Tietze option parameter searchSimultaneous is used by TzSearchEqual in the same way as described for TzSearch (48.7.2). However, TzSearchEqual does not use the parameter saveLimit: The loop over the relators is executed exactly once.

48.7.4 TzFindCyclicJoins

▷ `TzFindCyclicJoins(P)` (function)

searches for power and commutator relators in order to find pairs of generators which generate a common cyclic subgroup. It uses these pairs to introduce new relators, but it does not introduce any new generators as is done by `TzSubstituteCyclicJoins` (48.8.2).

More precisely: `TzFindCyclicJoins` searches for pairs of generators a and b such that (possibly after inverting or conjugating some relators) the set of relators contains the commutator $[a,b]$, a power a^n, and a product of the form $a^s b^t$ with s prime to n. For each such pair, `TzFindCyclicJoins` uses the Euclidean algorithm to express a as a power of b, and then it eliminates a.

48.8 Tietze Transformations that introduce new Generators

Some of the Tietze transformation commands listed so far may eliminate generators and hence change the given presentation to a presentation on a subset of the given set of generators, but they all do *not* introduce new generators. However, sometimes there will be the need to substitute certain words as new generators in order to improve a presentation. Therefore GAP offers the two commands `TzSubstitute` (48.8.1) and `TzSubstituteCyclicJoins` (48.8.2) which introduce new generators.

48.8.1 TzSubstitute

▷ `TzSubstitute(P, word)` (function)
▷ `TzSubstitute(P[, n[, eliminate]])` (function)

In the first form `TzSubstitute` expects `P` to be a presentation and `word` to be either an abstract word or a Tietze word in the generators of `P`. It substitutes the given word as a new generator of `P`. This is done as follows: First, `TzSubstitute` creates a new abstract generator, g say, and adds it to the presentation, then it adds a new relator $g^{-1} \cdot$ `word`.

In its second form, `TzSubstitute` substitutes a squarefree word of length 2 as a new generator and then eliminates a generator from the extended generator list. We will describe this process in more detail below.

The parameters `n` and `eliminate` are optional. If you specify arguments for them, then `n` is expected to be a positive integer, and `eliminate` is expected to be 0, 1, or 2. The default values are `n` $= 1$ and `eliminate` $= 0$.

`TzSubstitute` first determines the `n` most frequently occurring relator subwords of the form $g_1 g_2$, where g_1 and g_2 are different generators or their inverses, and sorts them by decreasing numbers of occurrences.

Let ab be the last word in that list, and let i be the smallest positive integer which has not yet been used as a generator number in the presentation `P` so far. `TzSubstitute` defines a new abstract generator x_i named `"_xi"` and adds it to `P` (see `AddGenerator` (48.5.1)). Then it adds the word $x_i^{-1}ab$ as a new relator to `P` and replaces all occurrences of ab in the relators by x_i. Finally, it eliminates some suitable generator from `P`.

The choice of the generator to be eliminated depends on the actual value of the parameter `eliminate`:

If `eliminate` is zero, `TzSubstitute` just calls the function `TzEliminate` (48.7.1). So it may happen that it is the just introduced generator x_i which now is deleted again so that you don't get

any remarkable progress in simplifying your presentation. On the first glance this does not look reasonable, but it is a consequence of the request that a call of TzSubstitute with *eliminate* = 0 must not increase the total length of the relators.

Otherwise, if *eliminate* is 1 or 2, TzSubstitute eliminates the respective factor of the substituted word *ab*, i. e., it eliminates *a* if *eliminate* = 1 or *b* if *eliminate* = 2. In this case, it may happen that the total length of the relators increases, but sometimes such an intermediate extension is the only way to finally reduce a given presentation.

There is still another property of the command TzSubstitute which should be mentioned. If, for instance, word is an abstract word, a call

Example

```
TzSubstitute( P, word );
```

is more or less equivalent to

Example

```
AddGenerator( P );
g := GeneratorsOfPresentation(P)[Length(GeneratorsOfPresentation(P))];
AddRelator( P, g^-1 * word );
```

However, there is a difference: If you are tracing generator images and preimages of *P* through the Tietze transformations applied to *P* (see 48.9), then TzSubstitute, as a Tietze transformation of *P*, will update and save the respective lists, whereas a call of the function AddGenerator (48.5.1) (which does not perform a Tietze transformation) will delete these lists and hence terminate the tracing.

Example

```
gap> G := PerfectGroup( IsSubgroupFpGroup, 960, 1 );
A5 2^4
gap> P := PresentationFpGroup( G );
<presentation with 6 gens and 21 rels of total length 84>
gap> GeneratorsOfPresentation( P );
[ a, b, s, t, u, v ]
gap> TzGoGo( P );
#I  there are 3 generators and 10 relators of total length 81
#I  there are 3 generators and 10 relators of total length 80
gap> TzPrintGenerators( P );
#I  1.  a   31 occurrences   involution
#I  2.  b   26 occurrences
#I  3.  t   23 occurrences   involution
gap> a := GeneratorsOfPresentation( P )[1];;
gap> b := GeneratorsOfPresentation( P )[2];;
gap> TzSubstitute( P, a*b );
#I  now the presentation has 4 generators, the new generator is _x7
#I  substituting new generator _x7 defined by a*b
#I  there are 4 generators and 11 relators of total length 83
gap> TzGo( P );
#I  there are 3 generators and 10 relators of total length 74
gap> TzPrintGenerators( P );
#I  1.  a   23 occurrences   involution
#I  2.  t   23 occurrences   involution
#I  3.  _x7   28 occurrences
```

As an example of an application of the command TzSubstitute in its second form we handle a subgroup of index 266 in the Janko group J_1.

Example

```
gap> F2 := FreeGroup( "a", "b" );;
gap> J1 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^7,
> Comm(F2.1,F2.2)^10, Comm(F2.1,F2.2^-1*(F2.1*F2.2)^2)^6 ];;
gap> a := J1.1;; b := J1.2;;
gap> H := Subgroup ( J1, [ a, b^(a*b*(a*b^-1)^2) ] );;
gap> P := PresentationSubgroup( J1, H );
<presentation with 23 gens and 82 rels of total length 530>
gap> TzGoGo( P );
#I  there are 3 generators and 47 relators of total length 1368
#I  there are 2 generators and 46 relators of total length 3773
#I  there are 2 generators and 46 relators of total length 2570
gap> TzGoGo( P );
#I  there are 2 generators and 46 relators of total length 2568
gap> TzGoGo( P );
```

Here we do not get any more progress without substituting a new generator.

Example

```
gap> TzSubstitute( P );
#I  substituting new generator _x28 defined by _x6*_x23^-1
#I  eliminating _x28 = _x6*_x23^-1
```

GAP cannot substitute a new generator without extending the total length, so we have to explicitly ask for it by using the second form of the command TzSubstitute. Our problem is to choose appropriate values for the arguments *n* and *eliminate*. For this purpose it may be helpful to print out a list of the most frequently occurring squarefree relator subwords of length 2.

Example

```
gap> TzPrintPairs( P );
#I  1.  504  occurrences of  _x6 * _x23^-1
#I  2.  504  occurrences of  _x6^-1 * _x23
#I  3.  448  occurrences of  _x6 * _x23
#I  4.  448  occurrences of  _x6^-1 * _x23^-1
gap> TzSubstitute( P, 2, 1 );
#I  substituting new generator _x29 defined by _x6^-1*_x23
#I  eliminating _x6 = _x23*_x29^-1
#I  there are 2 generators and 46 relators of total length 2867
gap> TzGoGo( P );
#I  there are 2 generators and 45 relators of total length 2417
#I  there are 2 generators and 45 relators of total length 2122
gap> TzSubstitute( P, 1, 2 );
#I  substituting new generator _x30 defined by _x23*_x29^-1
#I  eliminating _x29 = _x30^-1*_x23
#I  there are 2 generators and 45 relators of total length 2192
gap> TzGoGo( P );
#I  there are 2 generators and 42 relators of total length 1637
#I  there are 2 generators and 40 relators of total length 1286
#I  there are 2 generators and 36 relators of total length 807
#I  there are 2 generators and 32 relators of total length 625
#I  there are 2 generators and 22 relators of total length 369
```

```
#I  there are 2 generators and 18 relators of total length 213
#I  there are 2 generators and 13 relators of total length 141
#I  there are 2 generators and 12 relators of total length 121
#I  there are 2 generators and 10 relators of total length 101
gap> TzPrintPairs( P );
#I  1.  19  occurrences of  _x23 * _x30^-1
#I  2.  19  occurrences of  _x23^-1 * _x30
#I  3.  14  occurrences of  _x23 * _x30
#I  4.  14  occurrences of  _x23^-1 * _x30^-1
```

If we save a copy of the current presentation, then later we will be able to restart the computation from the current state.

Example

```
gap> P1 := ShallowCopy( P );
<presentation with 2 gens and 10 rels of total length 101>
```

Just for demonstration we make an inconvenient choice:

Example

```
gap> TzSubstitute( P, 3, 1 );
#I  substituting new generator _x31 defined by _x23*_x30
#I  eliminating _x23 = _x31*_x30^-1
#I  there are 2 generators and 10 relators of total length 122
gap> TzGoGo( P );
#I  there are 2 generators and 9 relators of total length 105
```

This presentation is worse than the one we have saved, so we restart from that presentation again.

Example

```
gap> P := ShallowCopy( P1 );
<presentation with 2 gens and 10 rels of total length 101>
gap> TzSubstitute( P, 2, 1);
#I  substituting new generator _x31 defined by _x23^-1*_x30
#I  eliminating _x23 = _x30*_x31^-1
#I  there are 2 generators and 10 relators of total length 107
gap> TzGoGo( P );
#I  there are 2 generators and 9 relators of total length 84
#I  there are 2 generators and 8 relators of total length 75
gap> TzSubstitute( P, 2, 1);
#I  substituting new generator _x32 defined by _x30^-1*_x31
#I  eliminating _x30 = _x31*_x32^-1
#I  there are 2 generators and 8 relators of total length 71
gap> TzGoGo( P );
#I  there are 2 generators and 7 relators of total length 56
#I  there are 2 generators and 5 relators of total length 36
gap> TzPrintRelators( P );
#I  1. _x32^5
#I  2. _x31^5
#I  3. (_x31^-1*_x32^-1)^3
#I  4. _x31*(_x32*_x31^-1)^2*_x32*_x31*_x32^-2
#I  5. _x31^-1*_x32^2*(_x31*_x32^-1*_x31)^2*_x32^2
```

48.8.2 TzSubstituteCyclicJoins

▷ `TzSubstituteCyclicJoins(P)` (function)

tries to find pairs of commuting generators a and b, say, such that the exponent of a (i. e. the least currently known positive integer n such that a^n is a relator in `P`) is prime to the exponent of b. For each such pair, their product ab is substituted as a new generator, and a and b are eliminated.

48.9 Tracing generator images through Tietze transformations

Any sequence of Tietze transformations applied to a presentation, starting from some presentation P_1 and ending up with some presentation P_2, defines an isomorphism, φ say, between the groups defined by P_1 and P_2, respectively. Sometimes it is desirable to know the images of the (old) generators of P_1 or the preimages of the (new) generators of P_2 under φ. The GAP Tietze transformation functions are able to trace these images. This is not automatically done because the involved words may grow to tremendous length, but it will be done if you explicitly request for it by calling the function `TzInitGeneratorImages` (48.9.1).

48.9.1 TzInitGeneratorImages

▷ `TzInitGeneratorImages(P)` (function)

expects `P` to be a presentation. It defines the current generators to be the "old generators" of `P` and initializes the (pre)image tracing. See `TzImagesOldGens` (48.9.3) and `TzPreImagesNewGens` (48.9.4) for details.

You can reinitialize the tracing of the generator images at any later state by just calling the function `TzInitGeneratorImages` again.

Note: A subsequent call of the function `DecodeTree` (48.10.1) will imply that the images and preimages are deleted and reinitialized after decoding the tree.

Moreover, if you introduce a new generator by calling the function `AddGenerator` (48.5.1) described in Section 48.5, this new generator cannot be traced in the old generators. Therefore `AddGenerator` (48.5.1) will terminate the tracing of the generator images and preimages and delete the respective lists whenever it is called.

48.9.2 OldGeneratorsOfPresentation

▷ `OldGeneratorsOfPresentation(P)` (function)

assumes that `P` is a presentation for which the generator images and preimages are being traced under Tietze transformations. It returns the list of old generators of `P`.

48.9.3 TzImagesOldGens

▷ `TzImagesOldGens(P)` (function)

assumes that `P` is a presentation for which the generator images and preimages are being traced under Tietze transformations. It returns a list l of words in the (current) `GeneratorsOfPresentation`

(48.1.3) value of P such that the i-th word $l[i]$ represents the i-th old generator of P, i. e., the i-th entry of the `OldGeneratorsOfPresentation` (48.9.2) value of P.

48.9.4 TzPreImagesNewGens

▷ `TzPreImagesNewGens(`*P*`)` (function)

assumes that P is a presentation for which the generator images and preimages are being traced under Tietze transformations. It returns a list l of words in the old generators of P (the `OldGeneratorsOfPresentation` (48.9.2) value of P) such that the i-th entry of l represents the i-th (current) generator of P (the `GeneratorsOfPresentation` (48.1.3) value of P).

48.9.5 TzPrintGeneratorImages

▷ `TzPrintGeneratorImages(`*P*`)` (function)

assumes that P is a presentation for which the generator images and preimages are being traced under Tietze transformations. It displays the preimages of the current generators as Tietze words in the old generators, and the images of the old generators as Tietze words in the current generators.

Example

```
gap> G := PerfectGroup( IsSubgroupFpGroup, 960, 1 );
A5 2^4
gap> P := PresentationFpGroup( G );
<presentation with 6 gens and 21 rels of total length 84>
gap> TzInitGeneratorImages( P );
gap> TzGo( P );
#I  there are 3 generators and 11 relators of total length 96
#I  there are 3 generators and 10 relators of total length 81
gap> TzPrintGeneratorImages( P );
#I  preimages of current generators as Tietze words in the old ones:
#I  1. [ 1 ]
#I  2. [ 2 ]
#I  3. [ 4 ]
#I  images of old generators as Tietze words in the current ones:
#I  1. [ 1 ]
#I  2. [ 2 ]
#I  3. [ 1, -2, 1, 3, 1, 2, 1 ]
#I  4. [ 3 ]
#I  5. [ -2, 1, 3, 1, 2 ]
#I  6. [ 1, 3, 1 ]
gap> gens := GeneratorsOfPresentation( P );
[ a, b, t ]
gap> oldgens := OldGeneratorsOfPresentation( P );
[ a, b, s, t, u, v ]
gap> TzImagesOldGens( P );
[ a, b, a*b^-1*a*t*a*b*a, t, b^-1*a*t*a*b, a*t*a ]
gap> for i in [ 1 .. Length( oldgens ) ] do
> Print( oldgens[i], " = ", TzImagesOldGens( P )[i], "\n" );
> od;
a = a
b = b
```

```
s = a*b^-1*a*t*a*b*a
t = t
u = b^-1*a*t*a*b
v = a*t*a
```

48.10 The Decoding Tree Procedure

48.10.1 DecodeTree

▷ `DecodeTree(P)` (function)

assumes that P is a subgroup presentation provided by the Reduced Reidemeister-Schreier or by the Modified Todd-Coxeter method (see `PresentationSubgroupRrs` (48.2.2), `PresentationNormalClosureRrs` (48.2.5), `PresentationSubgroupMtc` (48.2.4)). It eliminates the secondary generators of P (see Section 48.2) by applying the so called "decoding tree" procedure.

`DecodeTree` is called automatically by the command `PresentationSubgroupMtc` (48.2.4) where it reduces P to a presentation on the given (primary) subgroup generators.

In order to explain the effect of this command we need to insert a few remarks on the subgroup presentation commands described in section 48.2. All these commands have the common property that in the process of constructing a presentation for a given subgroup H of a finitely presented group G they first build up a highly redundant list of generators of H which consists of an (in general small) list of "primary" generators, followed by an (in general large) list of "secondary" generators, and then construct a presentation P_0, say, *on a sublist of these generators* by rewriting the defining relators of G. This sublist contains all primary, but, at least in general, by far not all secondary generators.

The role of the primary generators depends on the concrete choice of the subgroup presentation command. If the Modified Todd-Coxeter method is used, they are just the given generators of H, whereas in the case of the Reduced Reidemeister-Schreier algorithm they are constructed by the program.

Each of the secondary generators is defined by a word of length two in the preceding generators and their inverses. By historical reasons, the list of these definitions is called the *subgroup generators tree* though in fact it is not a tree but rather a kind of bush.

Now we have to distinguish two cases. If P_0 has been constructed by the Reduced Reidemeister-Schreier routines, it is a presentation of H. However, if the Modified Todd-Coxeter routines have been used instead, then the relators in P_0 are valid relators of H, but they do not necessarily define H. We handle these cases in turn, starting with the latter one.

In fact, we could easily receive a presentation of H also in this case if we extended P_0 by adding to it all the secondary generators which are not yet contained in it and all the definitions from the generators tree as additional generators and relators. Then we could recursively eliminate all secondary generators by Tietze transformations using the new relators. However, this procedure turns out to be too inefficient to be of interest.

Instead, we use the so called *decoding tree* procedure (see [AMW82], [AR84]). It proceeds as follows.

Starting from $P = P_0$, it runs through a number of steps in each of which it eliminates the current "last" generator (with respect to the list of all primary and secondary generators). If the last generator g, say, is a primary generator, then the procedure terminates. Otherwise it checks whether there is a relator in the current presentation which can be used to substitute g by a Tietze transformation. If

so, this is done. Otherwise, and only then, the tree definition of g is added to P as a new relator, and the generators involved are added as new generators if they have not yet been contained in P. Subsequently, g is eliminated.

Note that the extension of P by one or two new generators is *not* a Tietze transformation. In general, it will change the isomorphism type of the group defined by P. However, it is a remarkable property of this procedure, that at the end, i.e., as soon as all secondary generators have been eliminated, it provides a presentation $P = P_1$, say, which defines a group isomorphic to H. In fact, it is this presentation which is returned by the command DecodeTree and hence by the command PresentationSubgroupMtc (48.2.4).

If, in the other case, the presentation P_0 has been constructed by the Reduced Reidemeister-Schreier algorithm, then P_0 itself is a presentation of H, and the corresponding subgroup presentation command (PresentationSubgroupRrs (48.2.2) or PresentationNormalClosureRrs (48.2.5)) just returns P_0.

As mentioned in section 48.2, we recommend to further simplify this presentation before you use it. The standard way to do this is to start from P_0 and to apply suitable Tietze transformations, e. g., by calling the commands TzGo (48.6.1) or TzGoGo (48.6.3). This is probably the most efficient approach, but you will end up with a presentation on some unpredictable set of generators. As an alternative, GAP offers you the DecodeTree command which you can use to eliminate all secondary generators (provided that there are no space or time problems). For this purpose, the subgroup presentation commands do not only return the resulting presentation, but also the tree (together with some associated lists) as a kind of side result in a component P!.tree of the resulting presentation P.

Note, however, that the decoding tree routines will not work correctly any more on a presentation from which generators have already been eliminated by Tietze transformations. Therefore, to prevent you from getting wrong results by calling DecodeTree in such a situation, GAP will automatically remove the subgroup generators tree from a presentation as soon as one of the generators is substituted by a Tietze transformation.

Nevertheless, a certain misuse of the command is still possible, and we want to explicitly warn you from this. The reason is that the Tietze option parameters described in Section 48.11 apply to DecodeTree as well. Hence, in case of inadequate values of these parameters, it may happen that DecodeTree stops before all the secondary generators have vanished. In this case GAP will display an appropriate warning. Then you should change the respective parameters and continue the process by calling DecodeTree again. Otherwise, if you would apply Tietze transformations, it might happen because of the convention described above that the tree is removed and that you end up with a wrong presentation.

After a successful run of DecodeTree it is convenient to further simplify the resulting presentation by suitable Tietze transformations.

As an example of an explicit call of DecodeTree we compute two presentations of a subgroup of order 384 in a group of order 6912. In both cases we use the Reduced Reidemeister-Schreier algorithm, but in the first run we just apply the Tietze transformations offered by the TzGoGo (48.6.3) command with its default parameters, whereas in the second run we call the DecodeTree command before.

Example

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1*F2.2^2*F2.1^-1*F2.2^-1*F2.1^3*F2.2^-1,
>                F2.2*F2.1^2*F2.2^-1*F2.1^-1*F2.2^3*F2.1^-1 ];;
gap> a := G.1;;  b := G.2;;
gap> H := Subgroup( G, [ Comm(a^-1,b^-1), Comm(a^-1,b), Comm(a,b) ] );;
```

We use the Reduced Reidemeister Schreier method and default Tietze transformations to get a presentation for H.

Example

```
gap> P := PresentationSubgroupRrs( G, H );
<presentation with 18 gens and 35 rels of total length 169>
gap> TzGoGo( P );
#I  there are 3 generators and 20 relators of total length 488
#I  there are 3 generators and 20 relators of total length 466
```

We end up with 20 relators of total length 466. Now we repeat the procedure, but we call the decoding tree algorithm before doing the Tietze transformations.

Example

```
gap> P := PresentationSubgroupRrs( G, H );
<presentation with 18 gens and 35 rels of total length 169>
gap> DecodeTree( P );
#I  there are 9 generators and 26 relators of total length 185
#I  there are 6 generators and 23 relators of total length 213
#I  there are 3 generators and 20 relators of total length 252
#I  there are 3 generators and 20 relators of total length 244
gap> TzGoGo( P );
#I  there are 3 generators and 19 relators of total length 168
#I  there are 3 generators and 17 relators of total length 138
#I  there are 3 generators and 15 relators of total length 114
#I  there are 3 generators and 13 relators of total length 96
#I  there are 3 generators and 12 relators of total length 84
```

This time we end up with a shorter presentation.

As an example of an implicit call of the function `DecodeTree` via the command `PresentationSubgroupMtc` (48.2.4) we handle a subgroup of index 240 in a group of order 40320 given by a presentation due to B. H. Neumann. Note that we increase the level of `InfoFpGroup` (47.1.3) temporarily to get some additional output.

Example

```
gap> F3 := FreeGroup( "a", "b", "c" );;
gap> a := F3.1;;  b := F3.2;;  c := F3.3;;
gap> G := F3 / [ a^3, b^3, c^3, (a*b)^5, (a^-1*b)^5, (a*c)^4,
>     (a*c^-1)^4, a*b^-1*a*b*c^-1*a*c*a*c^-1, (b*c)^3, (b^-1*c)^4 ];;
gap> a := G.1;;  b := G.2;;  c := G.3;;
gap> H := Subgroup( G, [ a, c ] );;
gap> SetInfoLevel( InfoFpGroup, 1 );
gap> P := PresentationSubgroupMtc( G, H );;
#I  index = 240  total = 4737  max = 4507
#I  MTC defined 2 primary and 4444 secondary subgroup generators
#I  there are 246 generators and 617 relators of total length 2893
#I  calling DecodeTree
#I  there are 114 generators and 385 relators of total length 1860
#I  there are 69 generators and 294 relators of total length 1855
#I  there are 43 generators and 235 relators of total length 2031
#I  there are 35 generators and 207 relators of total length 2348
#I  there are 25 generators and 181 relators of total length 3055
#I  there are 19 generators and 165 relators of total length 3290
```

```
#I  there are 20 generators and 160 relators of total length 5151
#I  there are 23 generators and 159 relators of total length 8177
#I  there are 25 generators and 159 relators of total length 12241
#I  there are 29 generators and 159 relators of total length 18242
#I  there are 34 generators and 159 relators of total length 27364
#I  there are 38 generators and 159 relators of total length 41480
#I  there are 41 generators and 159 relators of total length 62732
#I  there are 45 generators and 159 relators of total length 88872
#I  there are 46 generators and 159 relators of total length 111092
#I  there are 44 generators and 155 relators of total length 158181
#I  there are 32 generators and 155 relators of total length 180478
#I  there are 7 generators and 133 relators of total length 29897
#I  there are 4 generators and 119 relators of total length 28805
#I  there are 3 generators and 116 relators of total length 35209
#I  there are 2 generators and 111 relators of total length 25658
#I  there are 2 generators and 111 relators of total length 22634
gap> TzGoGo( P );
#I  there are 2 generators and 108 relators of total length 11760
#I  there are 2 generators and 95 relators of total length 6482
#I  there are 2 generators and 38 relators of total length 1464
#I  there are 2 generators and 8 relators of total length 116
#I  there are 2 generators and 7 relators of total length 76
#I  there are 2 generators and 6 relators of total length 66
#I  there are 2 generators and 6 relators of total length 52
gap> TzPrintGenerators( P );
#I  1.  _x1   26 occurrences
#I  2.  _x2   26 occurrences
gap> TzPrint( P );
#I  generators: [ _x1, _x2 ]
#I  relators:
#I  1.  3  [ 1, 1, 1 ]
#I  2.  3  [ 2, 2, 2 ]
#I  3.  8  [ 2, -1, 2, -1, 2, -1, 2, -1 ]
#I  4.  8  [ 2, 1, 2, 1, 2, 1, 2, 1 ]
#I  5.  14  [ -1, -2, 1, 2, 1, -2, -1, 2, 1, -2, -1, -2, 1, 2 ]
#I  6.  16  [ 1, 2, 1, -2, 1, 2, 1, -2, 1, 2, 1, -2, 1, 2, 1, -2 ]
gap> K :=  FpGroupPresentation( P );
<fp group on the generators [ _x1, _x2 ]>
gap> SetInfoLevel( InfoFpGroup, 0 );
gap> Size( K );
168
```

48.11 Tietze Options

Several of the Tietze transformation commands described above are controlled by certain parameters, the *Tietze options*, which often have a tremendous influence on their performance and results. However, in each application of the commands, an appropriate choice of these option parameters will depend on the concrete presentation under investigation. Therefore we have implemented the Tietze options in such a way that they are associated to the presentation: Each presentation keeps its own set of Tietze option parameters as an attribute.

48.11.1 TzOptions

▷ `TzOptions(`*P*`)` (attribute)

is a record whose components direct the heuristics applied by the Tietze transformation functions.

You may alter the value of any of these Tietze options by just assigning a new value to the respective record component.

The following Tietze options are recognized by GAP:

`protected`**:**
The first `protected` generators in a presentation *P* are protected from being eliminated by the Tietze transformations functions. There are only two exceptions: The option `protected` is ignored by the functions `TzEliminate` (48.7.1) and `TzSubstitute` (48.8.1) because they explicitly specify the generator to be eliminated. The default value of `protected` is 0.

`eliminationsLimit`**:**
Whenever the elimination phase of the `TzGo` (48.6.1) command is entered for a presentation *P*, then it will eliminate at most `eliminationsLimit` generators (except for further ones which have turned out to be trivial). Hence you may use the `eliminationsLimit` parameter as a break criterion for the `TzGo` (48.6.1) command. Note, however, that it is ignored by the `TzEliminate` (48.7.1) command. The default value of `eliminationsLimit` is 100.

`expandLimit`**:**
Whenever the routine for eliminating more than 1 generators is called for a presentation *P* by the `TzEliminate` (48.7.1) command or the elimination phase of the `TzGo` (48.6.1) command, then it saves the given total length of the relators, and subsequently it checks the current total length against its value before each elimination. If the total length has increased to more than `expandLimit` per cent of its original value, then the routine returns instead of eliminating another generator. Hence you may use the `expandLimit` parameter as a break criterion for the `TzGo` (48.6.1) command. The default value of `expandLimit` is 150.

`generatorsLimit`**:**
Whenever the elimination phase of the `TzGo` (48.6.1) command is entered for a presentation *P* with n generators, then it will eliminate at most $n-$`generatorsLimit` generators (except for generators which turn out to be trivial). Hence you may use the `generatorsLimit` parameter as a break criterion for the `TzGo` (48.6.1) command. The default value of `generatorsLimit` is 0.

`lengthLimit`**:**
The Tietze transformation commands will never eliminate a generator of a presentation *P*, if they cannot exclude the possibility that the resulting total length of the relators exceeds the maximal GAP list length of $2^{31}-1$ or the value of the option `lengthLimit`. The default value of `lengthLimit` is $2^{31}-1$.

`loopLimit`**:**
Whenever the `TzGo` (48.6.1) command is called for a presentation *P*, then it will loop over at most `loopLimit` of its basic steps. Hence you may use the `loopLimit` parameter as a break criterion for the `TzGo` (48.6.1) command. The default value of `loopLimit` is `infinity` (18.2.1).

`printLevel`:
: Whenever Tietze transformation commands are called for a presentation P with `printLevel` $= 0$, they will not provide any output except for error messages. If `printLevel` $= 1$, they will display some reasonable amount of output which allows you to watch the progress of the computation and to decide about your next commands. In the case `printLevel` $= 2$, you will get a much more generous amount of output. Finally, if `printLevel` $= 3$, various messages on internal details will be added. The default value of `printLevel` is 1.

`saveLimit`:
: Whenever the `TzSearch` (48.7.2) command has finished its main loop over all relators of a presentation P, then it checks whether during this loop the total length of the relators has been reduced by at least `saveLimit` per cent. If this is the case, then `TzSearch` (48.7.2) repeats its procedure instead of returning. Hence you may use the `saveLimit` parameter as a break criterion for the `TzSearch` (48.7.2) command and, in particular, for the search phase of the `TzGo` (48.6.1) command. The default value of `saveLimit` is 10.

`searchSimultaneous`:
: Whenever the `TzSearch` (48.7.2) or the `TzSearchEqual` (48.7.3) command is called for a presentation P, then it is allowed to handle up to `searchSimultaneous` short relators simultaneously (see the description of the `TzSearch` (48.7.2) command for more details). The choice of this parameter may heavily influence the performance as well as the result of the `TzSearch` (48.7.2) and the `TzSearchEqual` (48.7.3) commands and hence also of the search phase of the `TzGo` (48.6.1) command. The default value of `searchSimultaneous` is 20.

48.11.2 TzPrintOptions

▷ `TzPrintOptions(`P`)` (function)

prints the current values of the Tietze options of the presentation P.

Example

```
gap> TzPrintOptions( P );
#I  protected          = 0
#I  eliminationsLimit  = 100
#I  expandLimit        = 150
#I  generatorsLimit    = 0
#I  lengthLimit        = 2147483647
#I  loopLimit          = infinity
#I  printLevel         = 1
#I  saveLimit          = 10
#I  searchSimultaneous = 20
```

Chapter 49

Group Products

This chapter describes the various group product constructions that are possible in GAP.

At the moment for some of the products methods are available only if both factors are given in the same representation or only for certain types of groups such as permutation groups and pc groups when the product can be naturally represented as a group of the same kind.

GAP does not guarantee that a product of two groups will be in a particular representation. (Exceptions are `WreathProductImprimitiveAction` (49.4.2) and `WreathProductProductAction` (49.4.3) which are construction that makes sense only for permutation groups, see `WreathProduct` (49.4.1)).

GAP however will try to choose an efficient representation, so products of permutation groups or pc groups often will be represented as a group of the same kind again.

Therefore the only guaranteed way to relate a product to its factors is via the embedding and projection homomorphisms, see 49.6.

49.1 Direct Products

The direct product of groups is the cartesian product of the groups (considered as element sets) with component-wise multiplication.

49.1.1 DirectProduct

▷ `DirectProduct(`*`G[, H, ...]`*`)` (function)
▷ `DirectProductOp(`*`list, expl`*`)` (operation)

These functions construct the direct product of the groups given as arguments. `DirectProduct` takes an arbitrary positive number of arguments and calls the operation `DirectProductOp`, which takes exactly two arguments, namely a nonempty list *`list`* of groups and one of these groups, *`expl`*. (This somewhat strange syntax allows the method selection to choose a reasonable method for special cases, e.g., if all groups are permutation groups or pc groups.)

GAP will try to choose an efficient representation for the direct product. For example the direct product of permutation groups will be a permutation group again and the direct product of pc groups will be a pc group.

If the groups are in different representations a generic direct product will be formed which may not be particularly efficient for many calculations. Instead it may be worth to convert all factors to a

common representation first, before forming the product.

For a direct product P, calling `Embedding` (32.2.10) with P and n yields the homomorphism embedding the n-th factor into P; calling `Projection` (32.2.11) with `P` and `n` yields the projection of P onto the n-th factor, see 49.6.

Example

```
gap> g:=Group((1,2,3),(1,2));;
gap> d:=DirectProduct(g,g,g);
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ])
gap> Size(d);
216
gap> e:=Embedding(d,2);
2nd embedding into Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9),
  (7,8) ])
gap> Image(e,(1,2));
(4,5)
gap> Image(Projection(d,2),(1,2,3)(4,5)(8,9));
(1,2)
```

49.2 Semidirect Products

The semidirect product of a group N with a group G acting on N via a homomorphism α from G into the automorphism group of N is the cartesian product $G \times N$ with the multiplication $(g,n) \cdot (h,m) = (gh, n^{h^\alpha} m)$.

49.2.1 SemidirectProduct

▷ `SemidirectProduct(`*`G, alpha, N`*`)` (operation)

▷ `SemidirectProduct(`*`autgp, N`*`)` (operation)

constructs the semidirect product of `N` with `G` acting via `alpha`, which must be a homomorphism from `G` into a group of automorphisms of `N`.

If `N` is a group, `alpha` must be a homomorphism from `G` into a group of automorphisms of `N`.

If `N` is a full row space over a field `F`, `alpha` must be a homomorphism from `G` into a matrix group of the right dimension over a subfield of `F`, or into a permutation group (in this case permutation matrices are taken).

In the second variant, *autgp* must be a group of automorphism of `N`, it is a shorthand for `SemidirectProduct(`*`autgp`*`,IdentityMapping(`*`autgp`*`),N)`. Note that (unless *autgrp* has been obtained by the operation `AutomorphismGroup` (40.7.1)) you have to test `IsGroupOfAutomorphisms` (40.7.2) for *autgrp* to ensure that GAP knows that *autgrp* consists of group automorphisms.

Example

```
gap> n:=AbelianGroup(IsPcGroup,[5,5]);
<pc group of size 25 with 2 generators>
gap> au:=DerivedSubgroup(AutomorphismGroup(n));;
gap> Size(au);
120
gap> p:=SemidirectProduct(au,n);
<permutation group with 5 generators>
gap> Size(p);
```

```
3000
gap> n:=Group((1,2),(3,4));;
gap> au:=AutomorphismGroup(n);;
gap> au:=First(Elements(au),i->Order(i)=3);;
gap> au:=Group(au);
<group with 1 generators>
gap> IsGroupOfAutomorphisms(au);
true
gap> SemidirectProduct(au,n);
<pc group with 3 generators>
gap> n:=AbelianGroup(IsPcGroup,[2,2]);
<pc group of size 4 with 2 generators>
gap> au:=AutomorphismGroup(n);
<group of size 6 with 2 generators>
gap> apc:=IsomorphismPcGroup(au);
CompositionMapping( Pcgs([ (2,3), (1,2,3) ]) ->
[ f1, f2 ], <action isomorphism> )
gap> g:=Image(apc);
Group([ f1, f2 ])
gap> apci:=InverseGeneralMapping(apc);
[ f1*f2^2, f1*f2 ] -> [ Pcgs([ f1, f2 ]) -> [ f1*f2, f2 ],
  Pcgs([ f1, f2 ]) -> [ f2, f1 ] ]
gap> IsGroupHomomorphism(apci);
true
gap> p:=SemidirectProduct(g,apci,n);
<pc group of size 24 with 4 generators>
gap> IsomorphismGroups(p,Group((1,2,3,4),(1,2)));
[ f1, f2, f3, f4 ] -> [ (3,4), (1,4,3), (1,2)(3,4), (1,3)(2,4) ]
gap> SemidirectProduct(SU(3,3),GF(9)^3);
<matrix group of size 4408992 with 3 generators>
gap> SemidirectProduct(Group((1,2,3),(2,3,4)),GF(5)^4);
<matrix group of size 7500 with 3 generators>
gap> g:=Group((3,4,5),(1,2,3));;
gap> mats:=[[[Z(2^2),0*Z(2)],[0*Z(2),Z(2^2)^2]],
>          [[Z(2)^0,Z(2)^0], [Z(2)^0,0*Z(2)]]];;
gap> hom:=GroupHomomorphismByImages(g,Group(mats),[g.1,g.2],mats);;
gap> SemidirectProduct(g,hom,GF(4)^2);
<matrix group of size 960 with 3 generators>
gap> SemidirectProduct(g,hom,GF(16)^2);
<matrix group of size 15360 with 4 generators>
```

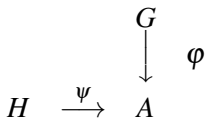
For a semidirect product P of G with N, calling `Embedding` (32.2.10) with P and `1` yields the embedding of G, calling `Embedding` (32.2.10) with P and `2` yields the embedding of N; calling `Projection` (32.2.11) with P yields the projection of P onto G, see 49.6.

Example

```
gap> Size(Image(Embedding(p,1)));
6
gap> Embedding(p,2);
[ f1, f2 ] -> [ f3, f4 ]
gap> Projection(p);
[ f1, f2, f3, f4 ] -> [ f1, f2, <identity> of ..., <identity> of ... ]
```

49.3 Subdirect Products

The subdirect product of the groups G and H with respect to the epimorphisms $\varphi: G \to A$ and $\psi: H \to A$ (for a common group A) is the subgroup of the direct product $G \times H$ consisting of the elements (g,h) for which $g^\varphi = h^\psi$. It is the pull-back of the following diagram.



49.3.1 SubdirectProduct

▷ `SubdirectProduct(`*G, H, Ghom, Hhom*`)` (operation)

constructs the subdirect product of *G* and *H* with respect to the epimorphisms *Ghom* from *G* onto a group A and *Hhom* from *H* onto the same group A.

For a subdirect product P, calling `Projection` (32.2.11) with P and n yields the projection on the n-th factor. (In general the factors do not embed into a subdirect product.)

Example

```
gap> g:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> hom:=GroupHomomorphismByImagesNC(g,g,[(1,2,3),(1,2)],[(),(1,2)]);
[ (1,2,3), (1,2) ] -> [ (), (1,2) ]
gap> s:=SubdirectProduct(g,g,hom,hom);
Group([ (1,2,3), (1,2)(4,5), (4,5,6) ])
gap> Size(s);
18
gap> p:=Projection(s,2);
2nd projection of Group([ (1,2,3), (1,2)(4,5), (4,5,6) ])
gap> Image(p,(1,3,2)(4,5,6));
(1,2,3)
```

49.3.2 SubdirectProducts

▷ `SubdirectProducts(`*G, H*`)` (function)

this function computes all subdirect products of *G* and *H* up to conjugacy in the direct product of Parent(*G*) and Parent(*H*). The subdirect products are returned as subgroups of this direct product.

49.4 Wreath Products

The wreath product of a group G with a permutation group P acting on n points is the semidirect product of the normal subgroup G^n with the group P which acts on G^n by permuting the components.

Note that GAP always considers the domain of a permutation group to be the points moved by elements of the group as returned by `MovedPoints` (42.3.3), i.e. it is not possible to have a domain to include fixed points, I.e. $P = \langle (1,2,3) \rangle$ and $P = \langle (1,3,5) \rangle$ result in isomorphic wreath products. (If fixed points are desired the wreath product $G \wr T$ has to be formed with a transitive overgroup T of P and then the pre-image of P under the projection $G \wr T \to T$ has to be taken.)

49.4.1 WreathProduct

▷ `WreathProduct(G, H[, hom])` (operation)
▷ `StandardWreathProduct(G, H)` (operation)

`WreathProduct` constructs the wreath product of the group `G` with the group `H`, acting as a permutation group.

If a third argument `hom` is given, it must be a homomorphism from `H` into a permutation group, and the action of this group on its moved points is considered.

If only two arguments are given, `H` must be a permutation group.

`StandardWreathProduct` returns the wreath product for the (right regular) permutation action of `H` on its elements.

For a wreath product W of `G` with a permutation group P of degree n and $1 \leq i \leq n$ calling `Embedding` (32.2.10) with W and i yields the embedding of `G` in the i-th component of the direct product of the base group G^n of W. For $i = n+1$, `Embedding` (32.2.10) yields the embedding of P into W. Calling `Projection` (32.2.11) with W yields the projection onto the acting group P, see 49.6.

Example

```
gap> g:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> p:=Group((1,2,3));
Group([ (1,2,3) ])
gap> w:=WreathProduct(g,p);
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8),
  (1,4,7)(2,5,8)(3,6,9) ])
gap> Size(w);
648
gap> Embedding(w,1);
1st embedding into Group( [ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9),
  (7,8), (1,4,7)(2,5,8)(3,6,9) ] )
gap> Image(Embedding(w,3));
Group([ (7,8,9), (7,8) ])
gap> Image(Embedding(w,4));
Group([ (1,4,7)(2,5,8)(3,6,9) ])
gap> Image(Projection(w),(1,4,8,2,6,7,3,5,9));
(1,2,3)
```

49.4.2 WreathProductImprimitiveAction

▷ `WreathProductImprimitiveAction(G, H)` (function)

For two permutation groups `G` and `H`, this function constructs the wreath product of `G` and `H` in the imprimitive action. If `G` acts on l points and `H` on m points this action will be on $l \cdot m$ points, it will be imprimitive with m blocks of size l each.

The operations `Embedding` (32.2.10) and `Projection` (32.2.11) operate on this product as described for general wreath products.

Example

```
gap> w:=WreathProductImprimitiveAction(g,p);;
gap> LargestMovedPoint(w);
9
```

49.4.3 WreathProductProductAction

▷ `WreathProductProductAction(G, H)` (function)

For two permutation groups `G` and `H`, this function constructs the wreath product in product action. If `G` acts on l points and `H` on m points this action will be on l^m points.

The operations `Embedding` (32.2.10) and `Projection` (32.2.11) operate on this product as described for general wreath products.

Example

```
gap> w:=WreathProductProductAction(g,p);
<permutation group of size 648 with 7 generators>
gap> LargestMovedPoint(w);
27
```

49.4.4 KuKGenerators

▷ `KuKGenerators(G, beta, alpha)` (function)

If `beta` is a homomorphism from `G` into a transitive permutation group, U the full preimage of the point stabilizer and `alpha` a homomorphism defined on (a superset) of U, this function returns images of the generators of `G` when mapping to the wreath product $(U\,$`alpha`$)\wr(G\,$`beta`$)$. (This is the Krasner-Kaloujnine embedding theorem.)

Example

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> hom:=GroupHomomorphismByImages(g,Group((1,2)),
> GeneratorsOfGroup(g),[(1,2),(1,2)]);;
gap> u:=PreImage(hom,Stabilizer(Image(hom),1));
Group([ (2,3,4), (1,2,4) ])
gap> hom2:=GroupHomomorphismByImages(u,Group((1,2,3)),
> GeneratorsOfGroup(u),[ (1,2,3), (1,2,3) ]);;
gap> KuKGenerators(g,hom,hom2);
[ (1,4)(2,5)(3,6), (1,6)(2,4)(3,5) ]
```

49.5 Free Products

Let G and H be groups with presentations $\langle X \mid R\rangle$ and $\langle Y \mid S\rangle$, respectively. Then the free product $G*H$ is the group with presentation $\langle X\cup Y \mid R\cup S\rangle$. This construction can be generalized to an arbitrary number of groups.

49.5.1 FreeProduct

▷ `FreeProduct(G[, H, ...])` (function)

▷ `FreeProduct(list)` (function)

constructs a finitely presented group which is the free product of the groups given as arguments. If the group arguments are not finitely presented groups, then `IsomorphismFpGroup` (47.11.1) must be defined for them.

The operation `Embedding` (32.2.10) operates on this product.

Example

```
gap> g := DihedralGroup(8);;
gap> h := CyclicGroup(5);;
gap> fp := FreeProduct(g,h,h);
<fp group on the generators [ f1, f2, f3, f4, f5 ]>
gap> fp := FreeProduct([g,h,h]);
<fp group on the generators [ f1, f2, f3, f4, f5 ]>
gap> Embedding(fp,2);
[ f1 ] -> [ f4 ]
```

49.6 Embeddings and Projections for Group Products

The relation between a group product and its factors is provided via homomorphisms, the embeddings in the product and the projections from the product. Depending on the kind of product only some of these are defined.

49.6.1 Embedding (for group products)

▷ `Embedding(`*P*`, `*nr*`)` (function)

returns the *nr*-th embedding in the group product *P*. The actual meaning of this embedding is described in the manual section for the appropriate product.

49.6.2 Projection (for group products)

▷ `Projection(`*P*`, `*nr*`)` (function)

returns the (*nr*-th) projection of the group product *P*. The actual meaning of the projection returned is described in the manual section for the appropriate product.

Chapter 50

Group Libraries

When you start GAP, it already knows several groups. Currently GAP initially knows the following groups:

- some basic groups, such as cyclic groups or symmetric groups (see 50.1),
- Classical matrix groups (see 50.2),
- the transitive permutation groups of degree at most 30 (see 50.6),
- a library of groups of small order (see 50.7),
- the finite perfect groups of size at most 10^6 (excluding 11 sizes) (see 50.8).
- the primitive permutation groups of degree < 2499 (see 50.9),
- the irreducible solvable subgroups of $GL(n,p)$ for $n > 1$ and $p^n < 256$ (see 50.11),
- the irreducible maximal finite integral matrix groups of dimension at most 31 (see 50.12),
- the crystallographic groups of dimension at most 4

There is usually no relation between the groups in the different libraries and a group may occur in different libraries in different incarnations.

Note that a system administrator may choose to install all, or only a few, or even none of the libraries. So some of the libraries mentioned below may not be available on your installation.

50.1 Basic Groups

There are several infinite families of groups which are parametrized by numbers. GAP provides various functions to construct these groups. The functions always permit (but do not require) one to indicate a filter (see 13.2), for example `IsPermGroup` (43.1.1), `IsMatrixGroup` (44.1.1) or `IsPcGroup` (46.3.1), in which the group shall be constructed. There always is a default filter corresponding to a "natural" way to describe the group in question. Note that not every group can be constructed in every filter, there may be theoretical restrictions (`IsPcGroup` (46.3.1) only works for solvable groups) or methods may be available only for a few filters.

Certain filters may admit additional hints. For example, groups constructed in `IsMatrixGroup` (44.1.1) may be constructed over a specified field, which can be given as second argument of the function that constructs the group; The default field is `Rationals` (17.1.1).

50.1.1 TrivialGroup

▷ TrivialGroup(*[filter]*) (function)

constructs a trivial group in the category given by the filter *filter*. If *filter* is not given it defaults to IsPcGroup (46.3.1).

Example

```
gap> TrivialGroup();
<pc group of size 1 with 0 generators>
gap> TrivialGroup( IsPermGroup );
Group(())
```

50.1.2 CyclicGroup

▷ CyclicGroup(*[filt,]n*) (function)

constructs the cyclic group of size *n* in the category given by the filter *filt*. If *filt* is not given it defaults to IsPcGroup (46.3.1).

Example

```
gap> CyclicGroup(12);
<pc group of size 12 with 3 generators>
gap> CyclicGroup(IsPermGroup,12);
Group([ (1,2,3,4,5,6,7,8,9,10,11,12) ])
gap> matgrp1:= CyclicGroup( IsMatrixGroup, 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp1 );
Rationals
gap> matgrp2:= CyclicGroup( IsMatrixGroup, GF(2), 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp2 );
GF(2)
```

50.1.3 AbelianGroup

▷ AbelianGroup(*[filt,]ints*) (function)

constructs an abelian group in the category given by the filter *filt* which is of isomorphism type $C_{ints[1]} \times C_{ints[2]} \times \ldots \times C_{ints[n]}$, where *ints* must be a list of positive integers. If *filt* is not given it defaults to IsPcGroup (46.3.1). The generators of the group returned are the elements corresponding to the integers in *ints*.

Example

```
gap> AbelianGroup([1,2,3]);
<pc group of size 6 with 3 generators>
```

50.1.4 ElementaryAbelianGroup

▷ ElementaryAbelianGroup(*[filt,]n*) (function)

constructs the elementary abelian group of size *n* in the category given by the filter *filt*. If *filt* is not given it defaults to IsPcGroup (46.3.1).

Example

```
gap> ElementaryAbelianGroup(8192);
<pc group of size 8192 with 13 generators>
```

50.1.5 FreeAbelianGroup

▷ FreeAbelianGroup(*[filt,]rank*) (function)

constructs the free abelian group of rank *n* in the category given by the filter *filt*. If *filt* is not given it defaults to IsFpGroup (47.1.2).

Example

```
gap> FreeAbelianGroup(4);
<fp group on the generators [ f1, f2, f3, f4 ]>
```

50.1.6 DihedralGroup

▷ DihedralGroup(*[filt,]n*) (function)

constructs the dihedral group of size *n* in the category given by the filter *filt*. If *filt* is not given it defaults to IsPcGroup (46.3.1).

Example

```
gap> DihedralGroup(10);
<pc group of size 10 with 2 generators>
```

50.1.7 QuaternionGroup

▷ QuaternionGroup(*[filt,]n*) (function)
▷ DicyclicGroup(*[filt,]n*) (function)

constructs the generalized quaternion group (or dicyclic group) of size *n* in the category given by the filter *filt*. Here, *n* is a multiple of 4. If *filt* is not given it defaults to IsPcGroup (46.3.1). Methods are also available for permutation and matrix groups (of minimal degree and minimal dimension in coprime characteristic).

Example

```
gap> QuaternionGroup(32);
<pc group of size 32 with 5 generators>
gap> g:=QuaternionGroup(IsMatrixGroup,CF(16),32);
Group([ [ [ 0, 1 ], [ -1, 0 ] ], [ [ E(16), 0 ], [ 0, -E(16)^7 ] ] ])
```

50.1.8 ExtraspecialGroup

▷ ExtraspecialGroup(*[filt,]order, exp*) (function)

Let *order* be of the form p^{2n+1}, for a prime integer p and a positive integer n. ExtraspecialGroup returns the extraspecial group of order *order* that is determined by *exp*, in the category given by the filter *filt*.

If p is odd then admissible values of *exp* are the exponent of the group (either p or p^2) or one of '+', "+", '-', "-". For $p = 2$, only the above plus or minus signs are admissible.

If *filt* is not given it defaults to IsPcGroup (46.3.1).

Example

```
gap> ExtraspecialGroup( 27, 3 );
<pc group of size 27 with 3 generators>
gap> ExtraspecialGroup( 27, '+' );
<pc group of size 27 with 3 generators>
gap> ExtraspecialGroup( 8, "-" );
<pc group of size 8 with 3 generators>
```

50.1.9 AlternatingGroup

▷ AlternatingGroup(*[filt,]deg*) (function)
▷ AlternatingGroup(*[filt,]dom*) (function)

constructs the alternating group of degree *deg* in the category given by the filter *filt*. If *filt* is not given it defaults to IsPermGroup (43.1.1). In the second version, the function constructs the alternating group on the points given in the set *dom* which must be a set of positive integers.

Example

```
gap> AlternatingGroup(5);
Alt( [ 1 .. 5 ] )
```

50.1.10 SymmetricGroup

▷ SymmetricGroup(*[filt,]deg*) (function)
▷ SymmetricGroup(*[filt,]dom*) (function)

constructs the symmetric group of degree *deg* in the category given by the filter *filt*. If *filt* is not given it defaults to IsPermGroup (43.1.1). In the second version, the function constructs the symmetric group on the points given in the set *dom* which must be a set of positive integers.

Example

```
gap> SymmetricGroup(10);
Sym( [ 1 .. 10 ] )
```

Note that permutation groups provide special treatment of symmetric and alternating groups, see 43.4.

50.1.11 MathieuGroup

▷ MathieuGroup(*[filt,]degree*) (function)

constructs the Mathieu group of degree *degree* in the category given by the filter *filt*, where *degree* must be in the set $\{9,10,11,12,21,22,23,24\}$. If *filt* is not given it defaults to IsPermGroup (43.1.1).

Example

```
gap> MathieuGroup( 11 );
Group([ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6) ])
```

50.1.12 SuzukiGroup

▷ SuzukiGroup(*[filt,]q*) (function)
▷ Sz(*[filt,]q*) (function)

Constructs a group isomorphic to the Suzuki group Sz(*q*) over the field with *q* elements, where *q* is a non-square power of 2.

If *filt* is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the Suzuki group itself.

Example

```
gap> SuzukiGroup( 32 );
Sz(32)
```

50.1.13 ReeGroup

▷ ReeGroup(*[filt,]q*) (function)
▷ Ree(*[filt,]q*) (function)

Constructs a group isomorphic to the Ree group ${}^2G_2(q)$ where $q = 3^{1+2m}$ for m a non-negative integer.

If *filt* is not given it defaults to IsMatrixGroup (44.1.1) and the generating matrices are based on [KLM01]. (No particular choice of a generating set is guaranteed.)

Example

```
gap> ReeGroup( 27 );
Ree(27)
```

50.2 Classical Groups

The following functions return classical groups. For the linear, symplectic, and unitary groups (the latter in dimension at least 3), the generators are taken from [Tay87]. For the unitary groups in dimension 2, the isomorphism of $\mathrm{SU}(2,q)$ and $\mathrm{SL}(2,q)$ is used, see for example [Hup67]. The generators of the general and special orthogonal groups are taken from [IE94] and [KL90], except that the generators of the groups in odd dimension in even characteristic are constructed via the isomorphism to a symplectic group, see for example [Car72]. The generators of the groups $\Omega^{\varepsilon}(d,q)$ are taken from [RT98], except that the generators of $\mathrm{SO}(5,2)$ are taken for $\Omega(5,2)$. The generators for the semilinear groups are constructed from the generators of the corresponding linear groups plus one additional generator that describes the action of the group of field automorphisms; for prime integers p and positive integers f, this yields the matrix groups $Gamma\mathrm{L}(d,p^f)$ and $Sigma\mathrm{L}(d,p^f)$ as groups of $df \times df$ matrices over the field with p elements.

For symplectic and orthogonal matrix groups returned by the functions described below, the invariant bilinear form is stored as the value of the attribute InvariantBilinearForm (44.5.1). Analogously, the invariant sesquilinear form defining the unitary groups is stored as the value of the attribute

InvariantSesquilinearForm (44.5.3)). The defining quadratic form of orthogonal groups is stored as the value of the attribute InvariantQuadraticForm (44.5.5).

Note that due to the different sources for the generators, the invariant forms for the groups $\Omega(e,d,q)$ are in general different from the forms for $\mathrm{SO}(e,d,q)$ and $\mathrm{GO}(e,d,q)$.

50.2.1 GeneralLinearGroup

▷ GeneralLinearGroup(*[filt,]d, R*) (function)
▷ GL(*[filt,]d, R*) (function)
▷ GeneralLinearGroup(*[filt,]d, q*) (function)
▷ GL(*[filt,]d, q*) (function)

The first two forms construct a group isomorphic to the general linear group GL(*d*, *R*) of all $d \times d$ matrices that are invertible over the ring *R*, in the category given by the filter *filt*.

The third and the fourth form construct the general linear group over the finite field with *q* elements.

If *filt* is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the general linear group as a matrix group in its natural action (see also IsNaturalGL (44.4.2), IsNaturalGLnZ (44.6.4)).

Currently supported rings *R* are finite fields, the ring Integers (14), and residue class rings Integers mod *m*, see 14.5.

Example

```
gap> GL(4,3);
GL(4,3)
gap> GL(2,Integers);
GL(2,Integers)
gap> GL(3,Integers mod 12);
GL(3,Z/12Z)
```

Using the OnLines (41.2.12) operation it is possible to obtain the corresponding projective groups in a permutation action:

Example

```
gap> g:=GL(4,3);;Size(g);
24261120
gap> pgl:=Action(g,Orbit(g,Z(3)^0*[1,0,0,0],OnLines),OnLines);;
gap> Size(pgl);
12130560
```

If you are interested only in the projective group as a permutation group and not in the correspondence between its moved points and the points in the projective space, you can also use PGL (50.2.11).

50.2.2 SpecialLinearGroup

▷ SpecialLinearGroup(*[filt,]d, R*) (function)
▷ SL(*[filt,]d, R*) (function)
▷ SpecialLinearGroup(*[filt,]d, q*) (function)

▷ SL([filt,]d, q) (function)

The first two forms construct a group isomorphic to the special linear group SL(d, R) of all those $d \times d$ matrices over the ring R whose determinant is the identity of R, in the category given by the filter filt.

The third and the fourth form construct the special linear group over the finite field with q elements.

If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the special linear group as a matrix group in its natural action (see also IsNaturalSL (44.4.4), IsNaturalSLnZ (44.6.5)).

Currently supported rings R are finite fields, the ring Integers (14), and residue class rings Integers mod m, see 14.5.

Example

```
gap> SpecialLinearGroup(2,2);
SL(2,2)
gap> SL(3,Integers);
SL(3,Integers)
gap> SL(4,Integers mod 4);
SL(4,Z/4Z)
```

50.2.3 GeneralUnitaryGroup

▷ GeneralUnitaryGroup([filt,]d, q) (function)
▷ GU([filt,]d, q) (function)

constructs a group isomorphic to the general unitary group GU(d, q) of those $d \times d$ matrices over the field with q^2 elements that respect a fixed nondegenerate sesquilinear form, in the category given by the filter filt.

If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the general unitary group itself.

Example

```
gap> GeneralUnitaryGroup( 3, 5 );
GU(3,5)
```

50.2.4 SpecialUnitaryGroup

▷ SpecialUnitaryGroup([filt,]d, q) (function)
▷ SU([filt,]d, q) (function)

constructs a group isomorphic to the special unitary group GU(d, q) of those $d \times d$ matrices over the field with q^2 elements whose determinant is the identity of the field and that respect a fixed nondegenerate sesquilinear form, in the category given by the filter filt.

If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the special unitary group itself.

Example

```
gap> SpecialUnitaryGroup( 3, 5 );
SU(3,5)
```

50.2.5 SymplecticGroup

▷ SymplecticGroup([filt,]d, q) (function)
▷ SymplecticGroup([filt,]d, ring) (function)
▷ Sp([filt,]d, q) (function)
▷ Sp([filt,]d, ring) (function)
▷ SP([filt,]d, q) (function)
▷ SP([filt,]d, ring) (function)

constructs a group isomorphic to the symplectic group Sp(d, q) of those $d \times d$ matrices over the field with q elements (respectively the ring ring) that respect a fixed nondegenerate symplectic form, in the category given by the filter filt.

If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the symplectic group itself.

At the moment finite fields or residue class rings Integers mod q, with q an odd prime power are supported.

Example

```
gap> SymplecticGroup( 4, 2 );
Sp(4,2)
gap> g:=SymplecticGroup(6,Integers mod 9);
Sp(6,Z/9Z)
gap> Size(g);
95928796265538862080
```

50.2.6 GeneralOrthogonalGroup

▷ GeneralOrthogonalGroup([filt,][e,]d, q) (function)
▷ GO([filt,][e,]d, q) (function)

constructs a group isomorphic to the general orthogonal group GO(e, d, q) of those $d \times d$ matrices over the field with q elements that respect a non-singular quadratic form (see InvariantQuadraticForm (44.5.5)) specified by e, in the category given by the filter filt.

The value of e must be 0 for odd d (and can optionally be omitted in this case), respectively one of 1 or -1 for even d. If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the general orthogonal group itself.

Note that in [KL90], GO is defined as the stabilizer $\Delta(V,F,\kappa)$ of the quadratic form, up to scalars, whereas our GO is called $I(V,F,\kappa)$ there.

50.2.7 SpecialOrthogonalGroup

▷ SpecialOrthogonalGroup([filt,][e,]d, q) (function)
▷ SO([filt,][e,]d, q) (function)

SpecialOrthogonalGroup returns a group isomorphic to the special orthogonal group SO(e, d, q), which is the subgroup of all those matrices in the general orthogonal group (see GeneralOrthogonalGroup (50.2.6)) that have determinant one, in the category given by the filter filt. (The index of SO(e, d, q) in GO(e, d, q) is 2 if q is odd, and 1 if q is even.) Also

interesting is the group Omega(e, d, q), see Omega (50.2.8), which is always of index 2 in SO(e, d, q).

If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the special orthogonal group itself.

Example

```
gap> GeneralOrthogonalGroup( 3, 7 );
GO(0,3,7)
gap> GeneralOrthogonalGroup( -1, 4, 3 );
GO(-1,4,3)
gap> SpecialOrthogonalGroup( 1, 4, 4 );
GO(+1,4,4)
```

50.2.8 Omega (construct an orthogonal group)

▷ Omega([filt,][e,]d, q) (operation)

constructs a group isomorphic to the group Ω(e, d, q) of those $d \times d$ matrices over the field with q elements that respect a non-singular quadratic form (see InvariantQuadraticForm (44.5.5)) specified by e, and that have square spinor norm in odd characteristic or Dickson invariant 0 in even characteristic, respectively, in the category given by the filter filt. This group has always index two in SO(e, d, q), see SpecialOrthogonalGroup (50.2.7).

The value of e must be 0 for odd d (and can optionally be omitted in this case), respectively one of 1 or -1 for even d. If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group is the group Ω(e, d, q) itself.

Example

```
gap> g:= Omega( 3, 5 );  StructureDescription( g );
Omega(0,3,5)
"A5"
gap> g:= Omega( 1, 4, 4 );  StructureDescription( g );
Omega(+1,4,4)
"A5 x A5"
gap> g:= Omega( -1, 4, 3 );  StructureDescription( g );
Omega(-1,4,3)
"A6"
```

50.2.9 GeneralSemilinearGroup

▷ GeneralSemilinearGroup([filt,]d, q) (function)
▷ GammaL([filt,]d, q) (function)

GeneralSemilinearGroup returns a group isomorphic to the general semilinear group ΓL(d, q) of semilinear mappings of the vector space GF(q)^d.

If filt is not given it defaults to IsMatrixGroup (44.1.1), and the returned group consists of matrices of dimension d f over the field with p elements, where $q = p^f$, for a prime integer p.

50.2.10 SpecialSemilinearGroup

▷ SpecialSemilinearGroup([filt,]d, q) (function)
▷ SigmaL([filt,]d, q) (function)

SpecialSemilinearGroup returns a group isomorphic to the special semilinear group ΣL(d, q) of those semilinear mappings of the vector space GF(q)^d (see GeneralSemilinearGroup (50.2.9)) whose linear part has determinant one.

If *filt* is not given it defaults to IsMatrixGroup (44.1.1), and the returned group consists of matrices of dimension $d\ f$ over the field with p elements, where $q = p^f$, for a prime integer p.

50.2.11 ProjectiveGeneralLinearGroup

▷ ProjectiveGeneralLinearGroup(*[filt,]d, q*) (function)
▷ PGL(*[filt,]d, q*) (function)

constructs a group isomorphic to the projective general linear group PGL(d, q) of those $d \times d$ matrices over the field with q elements, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to IsPermGroup (43.1.1), and the returned group is the action on lines of the underlying vector space.

50.2.12 ProjectiveSpecialLinearGroup

▷ ProjectiveSpecialLinearGroup(*[filt,]d, q*) (function)
▷ PSL(*[filt,]d, q*) (function)

constructs a group isomorphic to the projective special linear group PSL(d, q) of those $d \times d$ matrices over the field with q elements whose determinant is the identity of the field, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to IsPermGroup (43.1.1), and the returned group is the action on lines of the underlying vector space.

50.2.13 ProjectiveGeneralUnitaryGroup

▷ ProjectiveGeneralUnitaryGroup(*[filt,]d, q*) (function)
▷ PGU(*[filt,]d, q*) (function)

constructs a group isomorphic to the projective general unitary group PGU(d, q) of those $d \times d$ matrices over the field with q^2 elements that respect a fixed nondegenerate sesquilinear form, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to IsPermGroup (43.1.1), and the returned group is the action on lines of the underlying vector space.

50.2.14 ProjectiveSpecialUnitaryGroup

▷ ProjectiveSpecialUnitaryGroup(*[filt,]d, q*) (function)
▷ PSU(*[filt,]d, q*) (function)

constructs a group isomorphic to the projective special unitary group PSU(d, q) of those $d \times d$ matrices over the field with q^2 elements that respect a fixed nondegenerate sesquilinear form and have determinant 1, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to IsPermGroup (43.1.1), and the returned group is the action on lines of the underlying vector space.

50.2.15 ProjectiveSymplecticGroup

▷ ProjectiveSymplecticGroup([filt,]d, q) (function)
▷ PSP([filt,]d, q) (function)
▷ PSp([filt,]d, q) (function)

constructs a group isomorphic to the projective symplectic group PSp(*d*,*q*) of those $d \times d$ matrices over the field with *q* elements that respect a fixed nondegenerate symplectic form, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to IsPermGroup (43.1.1), and the returned group is the action on lines of the underlying vector space.

50.2.16 ProjectiveOmega

▷ ProjectiveOmega([filt,][e,]d, q) (function)
▷ POmega([filt,][e,]d, q) (function)

constructs a group isomorphic to the projective group PΩ(*e*, *d*, *q*) of Ω(*e*, *d*, *q*), modulo the centre (see Omega (50.2.8)), in the category given by the filter *filt*.

If *filt* is not given it defaults to IsPermGroup (43.1.1), and the returned group is the action on lines of the underlying vector space.

50.3 Conjugacy Classes in Classical Groups

For general and special linear groups (see GeneralLinearGroup (50.2.1) and SpecialLinearGroup (50.2.2)) GAP has an efficient method to generate representatives of the conjugacy classes. This uses results from linear algebra on normal forms of matrices. If you know how to do this for other types of classical groups, please, tell us.

Example

```
gap> g := SL(4,9);
SL(4,9)
gap> NrConjugacyClasses(g);
861
gap> cl := ConjugacyClasses(g);;
gap> Length(cl);
861
```

50.3.1 NrConjugacyClassesGL

▷ NrConjugacyClassesGL(n, q) (function)
▷ NrConjugacyClassesGU(n, q) (function)
▷ NrConjugacyClassesSL(n, q) (function)
▷ NrConjugacyClassesSU(n, q) (function)
▷ NrConjugacyClassesPGL(n, q) (function)

▷ NrConjugacyClassesPGU(*n*, *q*) (function)
▷ NrConjugacyClassesPSL(*n*, *q*) (function)
▷ NrConjugacyClassesPSU(*n*, *q*) (function)
▷ NrConjugacyClassesSLIsogeneous(*n*, *q*, *f*) (function)
▷ NrConjugacyClassesSUIsogeneous(*n*, *q*, *f*) (function)

The first of these functions compute for given positive integer *n* and prime power *q* the number of conjugacy classes in the classical groups GL(*n*, *q*), GU(*n*, *q*), SL(*n*, *q*), SU(*n*, *q*), PGL(*n*, *q*), PGU(*n*, *q*), PSL(*n*, *q*), PSL(*n*, *q*), respectively. (See also ConjugacyClasses (39.10.2) and Section 50.2.)

For each divisor *f* of *n* there is a group of Lie type with the same order as SL(*n*, *q*), such that its derived subgroup modulo its center is isomorphic to PSL(*n*, *q*). The various such groups with fixed *n* and *q* are called *isogeneous*. (Depending on congruence conditions on *q* and *n* several of these groups may actually be isomorphic.) The function NrConjugacyClassesSLIsogeneous computes the number of conjugacy classes in this group. The extreme cases $f = 1$ and $f = n$ lead to the groups SL(*n*, *q*) and PGL(*n*, *q*), respectively.

The function NrConjugacyClassesSUIsogeneous is the analogous one for the corresponding unitary groups.

The formulae for the number of conjugacy classes are taken from [Mac81].

Example

```
gap> NrConjugacyClassesGL(24,27);
225283995449391744060672885806099 52
gap> NrConjugacyClassesPSU(19,17);
15052300411163848367708
gap> NrConjugacyClasses(SL(16,16));
1229782938228219920
```

50.4 Constructors for Basic Groups

All functions described in the previous sections call constructor operations to do the work. The names of the constructors are obtained from the names of the functions by appending "Cons", so for example CyclicGroup (50.1.2) calls the constructor

CyclicGroupCons(*cat*, *n*)

The first argument *cat* for each method of this constructor must be the category for which the method is installed. For example the method for constructing a cyclic permutation group is installed as follows (see InstallMethod (78.2.1) for the meaning of the arguments.

Example

```
InstallMethod( CyclicGroupCons,
    "regular perm group",
    true,
    [ IsPermGroup and IsRegularProp and IsFinite, IsInt and IsPosRat ], 0,
    function( filter, n )

    ...

    end );
```

50.5 Selection Functions

`All`*`Library`*`Groups( `*`fun1`*`, `*`val1`*`, ... )`

For a number of the group libraries two *selection functions* are provided. Each `All`*`Library`*`Groups` selection function permits one to select *all* groups from the library *`Library`* that have a given set of properties. Currently, the library selection functions provided, of this type, are `AllSmallGroups` (50.7.2), `AllIrreducibleSolvableGroups` (50.11.3), `AllTransitiveGroups`, and `AllPrimitiveGroups`. Corresponding to each of these there is a `One`*`Library`*`Group` function (see below) which returns at most one group.

These functions take an arbitrary number of pairs (but at least one pair) of arguments. The first argument in such a pair is a function that can be applied to the groups in the library, and the second argument is either a single value that this function must return in order to have this group included in the selection, or a list of such values. For the function `AllSmallGroups` (50.7.2) the first such function must be `Size` (30.4.6), and, unlike the other library selection functions, it supports an alternative syntax where `Size` (30.4.6) is omitted (see `AllSmallGroups` (50.7.2)). Also, see `AllIrreducibleSolvableGroups` (50.11.3), for details pertaining to this function.

For an example, let us consider the selection function for the library of transitive groups (also see 50.6). The command

Example

```
gap> AllTransitiveGroups(NrMovedPoints,[10..15],
>                        Size,         [1..100],
>                        IsAbelian,    false    );
```

returns a list of all transitive groups with degree between 10 and 15 and size less than 100 that are not abelian.

Thus `AllTransitiveGroups` behaves as if it was implemented by a function similar to the one defined below, where `TransitiveGroupsList` is a list of all transitive groups. (Note that in the definition below we assume for simplicity that `AllTransitiveGroups` accepts exactly 4 arguments. It is of course obvious how to change this definition so that the function would accept a variable number of arguments.)

Example

```
AllTransitiveGroups := function( fun1, val1, fun2, val2 )
local    groups, g, i;
  groups := [];
  for i  in [ 1 .. Length( TransitiveGroupsList ) ] do
    g := TransitiveGroupsList[i];
    if      fun1(g) = val1  or IsList(val1) and fun1(g) in val1
        and fun2(g) = val2  or IsList(val2) and fun2(g) in val2
     then
      Add( groups, g );
    fi;
  od;
  return groups;
end;
```

Note that the real selection functions are considerably more difficult, to improve the efficiency. Most important, each recognizes a certain set of properties which are precomputed for the library without having to compute them anew for each group. This will substantially speed up the selection process. In the description of each library we will list the properties that are stored for this library.

`One`*`Library`*`Group( `*`fun1`*`, `*`val1`*`, ... )`

For each `All`*`Library`*`Groups` function (see above) there is a corresponding function `One`*`Library`*`Group` on exactly the same arguments, i.e., there are `OneSmallGroup`, `OneIrreducibleSolvableGroup`, `OneTransitiveGroup`, and `OnePrimitiveGroup`. Each function simply returns *one* group in the library that has the prescribed properties, instead of *all* such groups. It returns `fail` if no such group exists in the library.

50.6 Transitive Permutation Groups

The transitive groups library currently contains representatives for all transitive permutation groups of degree at most 30. Two permutations groups of the same degree are considered to be equivalent, if there is a renumbering of points, which maps one group into the other one. In other words, if they lie in the same conjugacy class under operation of the full symmetric group by conjugation.

The selection functions (see 50.5) for the transitive groups library are `AllTransitiveGroups` and `OneTransitiveGroup`. They obtain the following attributes from the database without having to compute them anew:

`NrMovedPoints` (42.3.4), `Size` (30.4.6), `Transitivity` (41.10.2), and `IsPrimitive` (41.10.7).

This library was computed by Gregory Butler, John McKay, Gordon Royle and Alexander Hulpke. The list of transitive groups up to degree 11 was published in [BM83], the list of degree 12 was published in [Roy87], degree 14 and 15 were published in [But93] and degrees 16-30 were published in [Hul96] and [Hul05]. (Groups of prime degree of course are primitive and were known long before.)

The arrangement and the names of the groups of degree up to 15 is the same as given in [CHM98]. With the exception of the symmetric and alternating group (which are represented as `SymmetricGroup` (50.1.10) and `AlternatingGroup` (50.1.9)) the generators for these groups also conform to this paper with the only difference that 0 (which is not permitted in GAP for permutations to act on) is always replaced by the degree.

50.6.1 TransitiveGroup

▷ `TransitiveGroup(`*`deg, nr`*`)` (function)

returns the *`nr`*-th transitive group of degree *`deg`*. Both *`deg`* and *`nr`* must be positive integers. The transitive groups of equal degree are sorted with respect to their size, so for example `TransitiveGroup( `*`deg`*`, 1 )` is a transitive group of degree and size *`deg`*, e.g, the cyclic group of size *`deg`*, if *`deg`* is a prime.

50.6.2 NrTransitiveGroups

▷ `NrTransitiveGroups(`*`deg`*`)` (function)

returns the number of transitive groups of degree *`deg`* stored in the library of transitive groups. The function returns `fail` if *`deg`* is beyond the range of the library.

Example

```
gap> TransitiveGroup(10,22);
S(5)[x]2
gap> l:=AllTransitiveGroups(NrMovedPoints,12,Size,1440,IsSolvable,false);
[ S(6)[x]2, M_10.2(12)=A_6.E_4(12)=[S_6[1/720]{M_10}S_6]2 ]
```

```
gap> List(l,IsSolvable);
[ false, false ]
```

50.6.3 TransitiveIdentification

▷ `TransitiveIdentification(G)` (attribute)

Let `G` be a permutation group, acting transitively on a set of up to 30 points. Then `TransitiveIdentification` will return the position of this group in the transitive groups library. This means, if `G` acts on m points and `TransitiveIdentification` returns n, then `G` is permutation isomorphic to the group `TransitiveGroup(m,n)`.

Note: The points moved do *not* need to be [1..n], the group $\langle (2,3,4),(2,3)\rangle$ is considered to be transitive on 3 points. If the group has several orbits on the points moved by it the result of `TransitiveIdentification` is undefined.

Example

```
gap> TransitiveIdentification(Group((1,2),(1,2,3)));
2
```

50.7 Small Groups

The Small Groups library gives access to all groups of certain "small" orders. The groups are sorted by their orders and they are listed up to isomorphism; that is, for each of the available orders a complete and irredundant list of isomorphism type representatives of groups is given. Currently, the library contains the following groups:

- those of order at most 2000 except 1024 (423 164 062 groups);
- those of cubefree order at most 50 000 (395 703 groups);
- those of order p^7 for the primes $p = 3,5,7,11$ (907 489 groups);
- those of order p^n for $n \leq 6$ and all primes p
- those of order $q^n \cdot p$ for q^n dividing 2^8, 3^6, 5^5 or 7^4 and all primes p with $p \neq q$;
- those of squarefree order;
- those whose order factorises into at most 3 primes.

The first three items in this list cover an explicit range of orders; the last four provide access to infinite families of groups having orders of certain types.

The library also has an identification function: it returns the library number of a given group. This function determines library numbers using invariants of groups. The function is available for all orders in the library except for the orders 512 and 1536 and except for the orders p^5, p^6 and p^7 above 2000.

The library is organised in 11 layers. Each layer contains the groups of certain orders and their corresponding group identification routines. It is possible to install the first n layers of the group library and the first m layers of the group identification for each $1 \leq m \leq n \leq 11$. This might be useful to save disk space. There is an extensive `README` file for the Small Groups library available in

the `small` directory of the GAP distribution containing detailed information on the layers. A brief description of the layers is given here:

(1) the groups whose order factorises into at most 3 primes.

(2) the remaining groups of order at most 1000 except 512 and 768.

(3) the remaining groups of order $2^n \cdot p$ with $n \leq 8$ and p an odd prime.

(4) the remaining groups of order 5^5, 7^4 and of order $q^n \cdot p$ for q^n dividing 3^6, 5^5 or 7^4 and $p \neq q$ a prime.

(5) the remaining groups of order at most 2000 except 1024, 1152, 1536 and 1920.

(6) the groups of orders 1152 and 1920.

(7) the groups of order 512.

(8) the groups of order 1536.

(9) the remaining groups of order p^n for $4 \leq n \leq 6$.

(10) the remaining groups of cubefree order at most 50 000 and of squarefree order.

(11) the remaining groups of order p^7 for $p = 3, 5, 7, 11$.

The data in this library has been carefully checked and cross-checked. It is believed to be reliable. However, no absolute guarantees are given and users should, as always, make their own checks in critical cases.

The data occupies about 30 MB (storing over 400 million groups in about 200 megabits). The group identification occupies about 47 MB of which 18 MB is used for the groups in layer (6). More information on the Small Groups library can be found on `http://www.icm.tu-bs.de/ag_algebra/software/small/`

This library has been constructed by Hans Ulrich Besche, Bettina Eick and E. A. O'Brien. A survey on this topic and an account of the history of group constructions can be found in [BEO02]. Further detailed information on the construction of this library is available in [New77], [O'B90], [O'B91], [BE99a], [BE99b], [BE01], [BEO01], [EO99a], [EO99b], [NOVL04], [Gir03], [DE05], [OVL05]. The Small Groups library incorporates the GAP 3 libraries `TwoGroup` and `ThreeGroup`. The data from these libraries was directly included into the Small Groups library, and the ordering there was preserved. The Small Groups library replaces the Gap 3 library of solvable groups of order at most 100. However, both the organisation and data descriptions of these groups has changed in the Small Groups library.

50.7.1 SmallGroup (for group order and index)

▷ `SmallGroup(`*`order, i`*`)` (function)
▷ `SmallGroup(`*`pair`*`)` (function)

returns the *i*-th group of order *order* in the catalogue. If the group is solvable, it will be given as a PcGroup; otherwise it will be given as a permutation group. If the groups of order *order* are not installed, the function reports an error and enters a break loop.

50.7.2 AllSmallGroups

▷ `AllSmallGroups(`*arg*`)` (function)

returns all groups with certain properties as specified by *arg*. If *arg* is a number n, then this function returns all groups of order n. However, the function can also take several arguments which then must be organized in pairs `function` and `value`. In this case the first function must be `Size` (30.4.6) and the first value an order or a range of orders. If value is a list then it is considered a list of possible function values to include. The function returns those groups of the specified orders having those properties specified by the remaining functions and their values.

Precomputed information is stored for the properties `IsAbelian` (35.4.9), `IsNilpotentGroup` (39.15.3), `IsSupersolvableGroup` (39.15.8), `IsSolvableGroup` (39.15.6), `RankPGroup` (39.15.22), `PClassPGroup` (39.15.21), `LGLength` (45.13.6), `FrattinifactorSize` and `FrattinifactorId` for the groups of order at most 2000 which have more than three prime factors, except those of order 512, 768, 1024, 1152, 1536, 1920 and those of order $p^n \cdot q > 1000$ with $n > 2$.

50.7.3 OneSmallGroup

▷ `OneSmallGroup(`*arg*`)` (function)

returns one group with certain properties as specified by *arg*. The permitted arguments are those supported by `AllSmallGroups` (50.7.2).

50.7.4 NumberSmallGroups

▷ `NumberSmallGroups(`*order*`)` (function)

returns the number of groups of order *order*.

50.7.5 IdSmallGroup

▷ `IdSmallGroup(`*G*`)` (attribute)
▷ `IdGroup(`*G*`)` (attribute)

returns the library number of *G*; that is, the function returns a pair `[`*order*`, `*i*`]` where *G* is isomorphic to `SmallGroup(` *order*`, `*i*` )`.

50.7.6 IdsOfAllSmallGroups

▷ `IdsOfAllSmallGroups(`*arg*`)` (function)

similar to `AllSmallGroups` but returns ids instead of groups. This may prevent workspace overflows, if a large number of groups are expected in the output.

50.7.7 IdGap3SolvableGroup

▷ IdGap3SolvableGroup(*G*) (attribute)
▷ Gap3CatalogueIdGroup(*G*) (attribute)

returns the catalogue number of *G* in the GAP 3 catalogue of solvable groups; that is, the function returns a pair [*order*, *i*] meaning that *G* is isomorphic to the group SolvableGroup(*order*, *i*) in GAP 3.

50.7.8 SmallGroupsInformation

▷ SmallGroupsInformation(*order*) (function)

prints information on the groups of the specified order.

50.7.9 UnloadSmallGroupsData

▷ UnloadSmallGroupsData() (function)

GAP loads all necessary data from the library automatically, but it does not delete the data from the workspace again. Usually, this will be not necessary, since the data is stored in a compressed format. However, if a large number of groups from the library have been loaded, then the user might wish to remove the data from the workspace and this can be done by the above function call.

Example

```
gap> G := SmallGroup( 768, 1000000 );
<pc group of size 768 with 9 generators>
gap> G := SmallGroup( [768, 1000000] );
<pc group of size 768 with 9 generators>
gap> AllSmallGroups( 6 );
[ <pc group of size 6 with 2 generators>,
  <pc group of size 6 with 2 generators> ]
gap> AllSmallGroups( Size, 120, IsSolvableGroup, false );
[ Group(
    [ (1,2,4,8)(3,6,9,5)(7,12,13,17)(10,14,11,15)(16,20,21,24)(18,22,
        19,23), (1,3,7)(2,5,10)(4,9,13)(6,11,8)(12,16,20)(14,18,
        22)(15,19,23)(17,21,24) ]), Group([ (1,2,3,4,5), (1,2) ]),
  Group([ (1,2,3,5,4), (1,3)(2,4)(6,7) ]) ]
gap> G := OneSmallGroup( 120, IsNilpotentGroup, false );
<pc group of size 120 with 5 generators>
gap> IdSmallGroup(G);
[ 120, 1 ]
gap> G := OneSmallGroup( Size, [1..1000], IsSolvableGroup, false );
Group([ (1,2,3,4,5), (1,2,3) ])
gap> IdSmallGroup(G);
[ 60, 5 ]
gap> UnloadSmallGroupsData();
gap> IdSmallGroup( GL( 2,3 ) );
[ 48, 29 ]
gap> IdSmallGroup( Group( (1,2,3,4),(4,5) ) );
[ 120, 34 ]
gap> IdsOfAllSmallGroups( Size, 60, IsSupersolvableGroup, true );
```

```
[ [ 60, 1 ], [ 60, 2 ], [ 60, 3 ], [ 60, 4 ], [ 60, 6 ], [ 60, 7 ],
  [ 60, 8 ], [ 60, 10 ], [ 60, 11 ], [ 60, 12 ], [ 60, 13 ] ]
gap> NumberSmallGroups( 512 );
10494213
gap> NumberSmallGroups( 2^8 * 23 );
1083472
```

Example

```
gap> NumberSmallGroups( 2^9 * 23 );
Error, the library of groups of size 11776 is not available called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap>
```

Example

```
gap> SmallGroupsInformation( 32 );

  There are 51 groups of order 32.
  They are sorted by their ranks.
     1 is cyclic.
     2 - 20 have rank 2.
     21 - 44 have rank 3.
     45 - 50 have rank 4.
     51 is elementary abelian.

  For the selection functions the values of the following attributes
  are precomputed and stored:
     IsAbelian, PClassPGroup, RankPGroup, FrattinifactorSize and
     FrattinifactorId.

  This size belongs to layer 2 of the SmallGroups library.
  IdSmallGroup is available for this size.
```

50.8 Finite Perfect Groups

The GAP library of finite perfect groups provides, up to isomorphism, a list of all perfect groups whose sizes are less than 10^6 excluding the following sizes:

- For $n = 61440$, 122880, 172032, 245760, 344064, 491520, 688128, or 983040, the perfect groups of size n have not completely been determined yet. The library neither provides the number of these groups nor the groups themselves.
- For $n = 86016$, 368640, or 737280, the library does not yet contain the perfect groups of size n, it only provides their numbers which are 52, 46, and 54, respectively.

Except for these eleven sizes, the list of altogether 1097 perfect groups in the library is complete. It relies on results of Derek F. Holt and Wilhelm Plesken which are published in their book "Perfect Groups" [HP89]. Moreover, they have supplied us with files with presentations of 488 of the groups.

In terms of these, the remaining 607 nontrivial groups in the library can be described as 276 direct products, 107 central products, and 224 subdirect products. They are computed automatically by suitable GAP functions whenever they are needed. Two additional groups omitted from the book "Perfect Groups" have also been included.

We are grateful to Derek Holt and Wilhelm Plesken for making their groups available to the GAP community by contributing their files. It should be noted that their book contains a lot of further information for many of the library groups. So we would like to recommend it to any GAP user who is interested in the groups.

The library has been brought into GAP format by Volkmar Felsch.

As all groups are stored by presentations, a permutation representation is obtained by coset enumeration. Note that some of the library groups do not have a faithful permutation representation of small degree. Computations in these groups may be rather time consuming.

50.8.1 SizesPerfectGroups

▷ `SizesPerfectGroups()` (function)

This is the ordered list of all numbers up to 10^6 that occur as sizes of perfect groups. One can iterate over the perfect groups library with:

Example

```
gap> for n in SizesPerfectGroups() do
>      for k in [1..NrPerfectLibraryGroups(n)] do
>        pg := PerfectGroup(n,k);
>      od;
>    od;
```

50.8.2 PerfectGroup

▷ `PerfectGroup(`*`[filt, ]size[, n]`*`)` (function)
▷ `PerfectGroup(`*`[filt, ]sizenumberpair`*`)` (function)

returns a group which is isomorphic to the library group specified by the size number `[` *`size`*`,` *`n`* `]` or by the two separate arguments *`size`* and *`n`*, assuming a default value of $n = 1$. The optional argument *`filt`* defines the filter in which the group is returned. Possible filters so far are `IsPermGroup` (43.1.1) and `IsSubgroupFpGroup` (47.1.1). In the latter case, the generators and relators used coincide with those given in [HP89].

Example

```
gap> G := PerfectGroup(IsPermGroup,6048,1);
U3(3)
gap> G:=PerfectGroup(IsPermGroup,823080,2);
A5 2^1 19^2 C 19^1
gap> NrMovedPoints(G);
6859
```

50.8.3 PerfectIdentification

▷ `PerfectIdentification(`*`G`*`)` (attribute)

This attribute is set for all groups obtained from the perfect groups library and has the value `[size,nr]` if the group is obtained with these parameters from the library.

50.8.4 NumberPerfectGroups

▷ `NumberPerfectGroups(size)` (function)

returns the number of non-isomorphic perfect groups of size *size* for each positive integer *size* up to 10^6 except for the eight sizes listed at the beginning of this section for which the number is not yet known. For these values as well as for any argument out of range it returns `fail`.

50.8.5 NumberPerfectLibraryGroups

▷ `NumberPerfectLibraryGroups(size)` (function)

returns the number of perfect groups of size *size* which are available in the library of finite perfect groups. (The purpose of the function is to provide a simple way to formulate a loop over all library groups of a given size.)

50.8.6 SizeNumbersPerfectGroups

▷ `SizeNumbersPerfectGroups(factor1, factor2, ...)` (function)

`SizeNumbersPerfectGroups` returns a list of pairs, each entry consisting of a group order and the number of those groups in the library of perfect groups that contain the specified factors *factor1*, *factor2*, ... among their composition factors.

Each argument must either be the name of a simple group or an integer which stands for the product of the sizes of one or more cyclic factors. (In fact, the function replaces all integers among the arguments by their product.)

The following text strings are accepted as simple group names.

- `An` or `A(n)` for the alternating groups A_n, $5 \leq n \leq 9$, for example `A5` or `A(6)`.
- `Ln(q)` or `L(n,q)` for PSL(n,q), where $n \in \{2,3\}$ and q a prime power, ranging
  - for $n = 2$ from 4 to 125
  - for $n = 3$ from 2 to 5
- `Un(q)` or `U(n,q)` for PSU(n,q), where $n \in \{3,4\}$ and q a prime power, ranging
  - for $n = 3$ from 3 to 5
  - for $n = 4$ from 2 to 2
- `Sp4(4)` or `S(4,4)` for the symplectic group Sp$(4,4)$,
- `Sz(8)` for the Suzuki group Sz(8),
- `Mn` or `M(n)` for the Mathieu groups M_{11}, M_{12}, and M_{22}, and
- `Jn` or `J(n)` for the Janko groups J_1 and J_2.

Note that, for most of the groups, the preceding list offers two different names in order to be consistent with the notation used in [HP89] as well as with the notation used in the `DisplayCompositionSeries` (39.17.6) command of GAP. However, as the names are compared as text strings, you are restricted to the above choice. Even expressions like `L2(2^5)` are not accepted.

As the use of the term $\mathrm{PSU}(n,q)$ is not unique in the literature, we mention that in this library it denotes the factor group of $\mathrm{SU}(n,q)$ by its centre, where $\mathrm{SU}(n,q)$ is the group of all $n \times n$ unitary matrices with entries in $GF(q^2)$ and determinant 1.

The purpose of the function is to provide a simple way to formulate a loop over all library groups which contain certain composition factors.

50.8.7 DisplayInformationPerfectGroups

▷ `DisplayInformationPerfectGroups(`*`size[, n]`*`)` (function)
▷ `DisplayInformationPerfectGroups(`*`sizenumberpair`*`)` (function)

`DisplayInformationPerfectGroups` displays some invariants of the *`n`*-th group of order *`size`* from the perfect groups library.

If no value of *`n`* has been specified, the invariants will be displayed for all groups of size *`size`* available in the library.

Alternatively, also a list of length two may be entered as the only argument, with entries *`size`* and *`n`*.

The information provided for G includes the following items:

- a headline containing the size number `[ `*`size`*`, `*`n`*` ]` of G in the form *`size`*`.`*`n`* (the suffix `.`*`n`* will be suppressed if, up to isomorphism, G is the only perfect group of order *`size`*),
- a message if G is simple or quasisimple, i.e., if the factor group of G by its centre is simple,
- the "description" of the structure of G as it is given by Holt and Plesken in [HP89] (see below),
- the size of the centre of G (suppressed, if G is simple),
- the prime decomposition of the size of G,
- orbit sizes for a faithful permutation representation of G which is provided by the library (see below),
- a reference to each occurrence of G in the tables of section 5.3 of [HP89]. Each of these references consists of a class number and an internal number (i,j) under which G is listed in that class. For some groups, there is more than one reference because these groups belong to more than one of the classes in the book.

Example

```
gap> DisplayInformationPerfectGroups( 30720, 3 );
#I Perfect group 30720:  A5 ( 2^4 E N 2^1 E 2^4 ) A
#I   size = 2^11*3*5  orbit size = 240
#I   Holt-Plesken class 1 (9,3)
gap> DisplayInformationPerfectGroups( 30720, 6 );
#I Perfect group 30720:  A5 ( 2^4 x 2^4 ) C N 2^1
#I   centre = 2  size = 2^11*3*5  orbit size = 384
#I   Holt-Plesken class 1 (9,6)
gap> DisplayInformationPerfectGroups( Factorial( 8 ) / 2 );
```

```
#I Perfect group 20160.1:  A5 x L3(2) 2^1
#I   centre = 2  size = 2^6*3^2*5*7  orbit sizes = 5 + 16
#I   Holt-Plesken class 31 (1,1) (occurs also in class 32)
#I Perfect group 20160.2:  A5 2^1 x L3(2)
#I   centre = 2  size = 2^6*3^2*5*7  orbit sizes = 7 + 24
#I   Holt-Plesken class 31 (1,2) (occurs also in class 32)
#I Perfect group 20160.3:  ( A5 x L3(2) ) 2^1
#I   centre = 2  size = 2^6*3^2*5*7  orbit size = 192
#I   Holt-Plesken class 31 (1,3)
#I Perfect group 20160.4:  simple group  A8
#I   size = 2^6*3^2*5*7  orbit size = 8
#I   Holt-Plesken class 26 (0,1)
#I Perfect group 20160.5:  simple group  L3(4)
#I   size = 2^6*3^2*5*7  orbit size = 21
#I   Holt-Plesken class 27 (0,1)
```

50.8.8 More about the Perfect Groups Library

For any library group G, the library files do not only provide a presentation, but, in addition, a list of one or more subgroups $S_1, \ldots, S_r$ of G such that there is a faithful permutation representation of G of degree $\sum_{i=1}^{r}[G : S_i]$ on the set $\{S_i g \mid 1 \leq i \leq r, g \in G\}$ of the cosets of the S_i. This allows one to construct the groups as permutation groups. The function `DisplayInformationPerfectGroups` (50.8.7) displays only the available degree. The message

Example

```
orbit size = 8
```

in the above example means that the available permutation representation is transitive and of degree 8, whereas the message

Example

```
orbit sizes = 5 + 16
```

means that a nontransitive permutation representation is available which acts on two orbits of size 5 and 16 respectively.

The notation used in the "description" of a group is explained in section 5.1.2 of [HP89]. We quote the respective page from there:

Within a class $Q\#p$, an isomorphism type of groups will be denoted by an ordered pair of integers (r,n), where $r \geq 0$ and $n > 0$. More precisely, the isomorphism types in $Q\#p$ of order $p^r|Q|$ will be denoted by $(r,1),(r,2),(r,3),\ldots$. Thus Q will always get the size number $(0,1)$.

In addition to the symbol (r,n), the groups in $Q\#p$ will also be given a more descriptive name. The purpose of this is to provide a very rough idea of the structure of the group. The names are derived in the following manner. First of all, the isomorphism classes of irreducible F_pQ-modules M with $|Q|.|M| \leq 10^6$, where F_p is the field of order p, are assigned symbols. These will either be simply p^x, where x is the dimension of the module, or, if there is more than one isomorphism class of irreducible modules having the same dimension, they will be denoted byp^x, $p^{x'}$, etc. The one-dimensional module with trivial Q-action will therefore be denoted by p^1. These symbols will be listed under the description of Q. The group name consists essentially of a list of the composition factors working from the top of the group downwards; hence it always starts with the name of Q itself. (This convention is the most convenient in our context, but it is different from that adopted in

the ATLAS [CCN+85], for example, where composition factors are listed in the reverse order. For example, we denote a group isomorphic to $SL(2,5)$ by $A_5 2^1$ rather than $2.A_5$.)

Some other symbols are used in the name, in order to give some idea of the relationship between these composition factors, and splitting properties. We shall now list these additional symbols.

$\times$ between two factors denotes a direct product of F_pQ-modules or groups.

C (for "commutator") between two factors means that the second lies in the commutator subgroup of the first. Similarly, a segment of the form $(f_1 \times f_2)Cf_3$ would mean that the factors f_1 and f_2 commute modulo f_3 and f_3 lies in $[f_1, f_2]$.

A (for "abelian") between two factors indicates that the second is in the pth power (but not the commutator subgroup) of the first. "A" may also follow the factors, if bracketed.

E (for "elementary abelian") between two factors indicates that together they generate an elementary abelian group (modulo subsequent factors), but that the resulting F_pQ-module extension does not split.

N (for "nonsplit") before a factor indicates that Q (or possibly its covering group) splits down as far at this factor but not over the factor itself. So "Qf_1Nf_2" means that the normal subgroup f_1f_2 of the group has no complement but, modulo f_2, f_1, does have a complement.

Brackets have their obvious meaning. Summarizing, we have:

$\times$ = direct product;

C = commutator subgroup;

A = abelian;

E = elementary abelian; and

N = nonsplit.

Here are some examples.

(i) $A_5(2^4E2^1E2^4)A$ means that the pairs 2^4E2^1 and 2^1E2^4 are both elementary abelian of exponent 4.

(ii) $A_5(2^4E2^1A)C2^1$ means that $O_2(G)$ is of symplectic type 2^{1+5}, with Frattini factor group of type 2^4E2^1. The "A" after the 2^1 indicates that G has a central cyclic subgroup 2^1A2^1 of order 4.

(iii) $L_3(2)((2^1E)\times(N2^3E2^{3'}A)C)2^{3'}$ means that the $2^{3'}$ factor at the bottom lies in the commutator subgroup of the pair $2^3E2^{3'}$ in the middle, but the lower pair $2^{3'}A2^{3'}$ is abelian of exponent 4. There is also a submodule $2^1E2^{3'}$, and the covering group $L_3(2)2^1$ of $L_3(2)$ does not split over the 2^3 factor. (Since G is perfect, it goes without saying that the extension $L_3(2)2^1$ cannot split itself.)

We must stress that this notation does not always succeed in being precise or even unambiguous, and the reader is free to ignore it if it does not seem helpful.

If such a group description has been given in the book for G (and, in fact, this is the case for most of the library groups), it is displayed by `DisplayInformationPerfectGroups` (50.8.7). Otherwise the function provides a less explicit description of the (in these cases unique) Holt-Plesken class to which G belongs, together with a serial number if this is necessary to make it unique.

50.9 Primitive Permutation Groups

GAP contains a library of primitive permutation groups which includes, up to permutation isomorphism (i.e., up to conjugacy in the corresponding symmetric group), all primitive permutation groups of degree < 2500, calculated in [RD05], in particular,

- the primitive permutation groups up to degree 50, calculated by C. Sims,
- the primitive groups with insoluble socles of degree < 1000 as calculated in [DM88],
- the solvable (hence affine) primitive permutation groups of degree < 256 as calculated by M. Short [Sho92],
- some insolvable affine primitive permutation groups of degree < 256 as calculated in [The97].
- The solvable primitive groups of degree up to 999 as calculated in [EH03].
- The primitive groups of affine type of degree up to 999 as calculated in [RDU03].

Not all groups are named, those which do have names use ATLAS notation. Not all names are necessary unique!

The list given in [RD05] is believed to be complete, correcting various omissions in [DM88], [Sho92] and [The97].

In detail, we guarantee the following properties for this and further versions (but *not* versions which came before GAP 4.2) of the library:

- All groups in the library are primitive permutation groups of the indicated degree.
- The positions of the groups in the library are stable. That is `PrimitiveGroup(`*n*`,`*nr*`)` will always give you a permutation isomorphic group. Note however that we do not guarantee to keep the chosen S_n-representative, the generating set or the name for eternity.
- Different groups in the library are not conjugate in S_n.
- If a group in the library has a primitive subgroup with the same socle, this group is in the library as well.

(Note that the arrangement of groups is not guaranteed to be in increasing size, though it holds for many degrees.)

The selection functions (see 50.5) for the primitive groups library are `AllPrimitiveGroups` and `OnePrimitiveGroup`. They obtain the following properties from the database without having to compute them anew:

`NrMovedPoints` (42.3.4), `Size` (30.4.6), `Transitivity` (41.10.2), `ONanScottType` (43.5.1), `IsSimpleGroup` (39.15.10), `IsSolvableGroup` (39.15.6), and `SocleTypePrimitiveGroup` (43.5.2).

(Note, that for groups of degree up to 2499, O'Nan-Scott types 4a, 4b and 5 cannot occur.)

50.9.1 PrimitiveGroup

▷ PrimitiveGroup(*deg, nr*) (function)

returns the primitive permutation group of degree *deg* with number *nr* from the list.

The arrangement of the groups differs from the arrangement of primitive groups in the list of C. Sims, which was used in GAP 3. See SimsNo (50.10.2).

50.9.2 NrPrimitiveGroups

▷ NrPrimitiveGroups(*deg*) (function)

returns the number of primitive permutation groups of degree *deg* in the library.

Example

```
gap> NrPrimitiveGroups(25);
28
gap> PrimitiveGroup(25,19);
5^2:((Q(8):3)'4)
gap> PrimitiveGroup(25,20);
ASL(2, 5)
gap> PrimitiveGroup(25,22);
AGL(2, 5)
gap> PrimitiveGroup(25,23);
(A(5) x A(5)):2
```

50.9.3 PrimitiveGroupsIterator

▷ PrimitiveGroupsIterator(*attr1, val1, attr2, val2, ...*) (function)

returns an iterator through AllPrimitiveGroups(*attr1*,*val1*,*attr2*,*val2*,...) without creating all these groups at the same time.

50.9.4 COHORTS_PRIMITIVE_GROUPS

▷ COHORTS_PRIMITIVE_GROUPS (global variable)

In [DM88] the primitive groups are sorted in "cohorts" according to their socle. For each degree, the variable COHORTS_PRIMITIVE_GROUPS (50.9.4) contains a list of the cohorts for the primitive groups of this degree. Each cohort is represented by a list of length 2, the first entry specifies the socle type (see SocleTypePrimitiveGroup (43.5.2)), the second entry listing the index numbers of the groups in this degree.

For example in degree 49, we have four cohorts with socles $(\mathbb{Z}/7\mathbb{Z})^2$, $L_2(7)^2$, A_7^2 and A_{49} respectively. the group PrimitiveGroup(49,36), which is isomorphic to $(A_7 \times A_7) : 2^2$, lies in the third cohort with socle $(A_7 \times A_7)$.

Example

```
gap> COHORTS_PRIMITIVE_GROUPS[49];
[ [ rec( parameter := 7, series := "Z", width := 2 ),
      [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
          18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
```

```
          33 ] ],
    [ rec( parameter := [ 2, 7 ], series := "L", width := 2 ), [ 34 ] ],
    [ rec( parameter := 7, series := "A", width := 2 ),
        [ 35, 36, 37, 38 ] ],
    [ rec( parameter := 49, series := "A", width := 1 ), [ 39, 40 ] ] ]
```

50.10 Index numbers of primitive groups

50.10.1 PrimitiveIdentification

▷ `PrimitiveIdentification(`*G*`)` (attribute)

For a primitive permutation group for which an S_n-conjugate exists in the library of primitive permutation groups (see 50.9), this attribute returns the index position. That is *G* is conjugate to `PrimitiveGroup(NrMovedPoints(`*G*`),PrimitiveIdentification(`*G*`))`.

Methods only exist if the primitive groups library is installed.

Note: As this function uses the primitive groups library, the result is only guaranteed to the same extent as this library. If it is incomplete, `PrimitiveIdentification` might return an existing index number for a group not in the library.

Example

```
gap> PrimitiveIdentification(Group((1,2),(1,2,3)));
2
```

50.10.2 SimsNo

▷ `SimsNo(`*G*`)` (attribute)

If *G* is a primitive group obtained by `PrimitiveGroup` (50.9.1) (respectively one of the selection functions) this attribute contains the number of the isomorphic group in the original list of C. Sims. (This is the arrangement as it was used in GAP 3.)

Example

```
gap> g:=PrimitiveGroup(25,2);
5^2:S(3)
gap> SimsNo(g);
3
```

As mentioned in the previous section, the index numbers of primitive groups in GAP are guaranteed to remain stable. (Thus, missing groups will be added to the library at the end of each degree.) In particular, it is safe to refer to a primitive group of type *deg*, *nr* in the GAP library.

50.10.3 PRIMITIVE_INDICES_MAGMA

▷ `PRIMITIVE_INDICES_MAGMA` (global variable)

The system Magma also provides a list of primitive groups (see [RDU03]). For historical reasons, its indexing up to degree 999 differs from the one used by GAP. The variable `PRIMITIVE_INDICES_MAGMA` (50.10.3) can be used to obtain this correspondence. The

magma index number of the GAP group PrimitiveGroup(*deg*,*nr*) is stored in the entry PRIMITIVE_INDICES_MAGMA[*deg*][*nr*], for degree at most 999.

Vice versa, the group of degree *deg* with Magma index number *nr* has the GAP index Position(PRIMITIVE_INDICES_MAGMA[*deg*],*nr*), in particular it can be obtained by the GAP command

PrimitiveGroup(*deg*,Position(PRIMITIVE_INDICES_MAGMA[*deg*],*nr*));

50.11 Irreducible Solvable Matrix Groups

50.11.1 IrreducibleSolvableGroupMS

▷ IrreducibleSolvableGroupMS(*n, p, i*) (function)

This function returns a representative of the *i*-th conjugacy class of irreducible solvable subgroup of GL(n, p), where n is an integer > 1, p is a prime, and $p^n < 256$.

The numbering of the representatives should be considered arbitrary. However, it is guaranteed that the *i*-th group on this list will lie in the same conjugacy class in all future versions of GAP, unless two (or more) groups on the list are discovered to be duplicates, in which case IrreducibleSolvableGroupMS will return fail for all but one of the duplicates.

For values of n, p, and i admissible to IrreducibleSolvableGroup (50.11.6), IrreducibleSolvableGroupMS returns a representative of the same conjugacy class of subgroups of GL(n, p) as IrreducibleSolvableGroup (50.11.6). Note that it currently adds two more groups (missing from the original list by Mark Short) for $n = 2$, $p = 13$.

50.11.2 NumberIrreducibleSolvableGroups

▷ NumberIrreducibleSolvableGroups(*n, p*) (function)

This function returns the number of conjugacy classes of irreducible solvable subgroup of GL(n, p).

50.11.3 AllIrreducibleSolvableGroups

▷ AllIrreducibleSolvableGroups(*func1, val1, func2, val2, ...*) (function)

This function returns a list of conjugacy class representatives G of matrix groups over a prime field such that $f(G) = v$ or $f(G) \in v$, for all pairs (f,v) in (*func1*, *val1*), (*func2*, *val2*), ... The following possibilities for the functions f are particularly efficient, because the values can be read off the information in the data base: DegreeOfMatrixGroup (or Dimension (57.3.3) or DimensionOfMatrixGroup (44.2.1)) for the linear degree, Characteristic (31.10.1) for the field characteristic, Size (30.4.6), IsPrimitiveMatrixGroup (or IsLinearlyPrimitive), and MinimalBlockDimension>.

50.11.4 OneIrreducibleSolvableGroup

▷ OneIrreducibleSolvableGroup(*func1, val1, func2, val2, ...*) (function)

This function returns one solvable subgroup G of a matrix group over a prime field such that $f(G) = v$ or $f(G) \in v$, for all pairs (f, v) in (*func1*, *val1*), (*func2*, *val2*), ... The following possibilities for the functions f are particularly efficient, because the values can be read off the information in the data base: `DegreeOfMatrixGroup` (or `Dimension` (57.3.3) or `DimensionOfMatrixGroup` (44.2.1)) for the linear degree, `Characteristic` (31.10.1) for the field characteristic, `Size` (30.4.6), `IsPrimitiveMatrixGroup` (or `IsLinearlyPrimitive`), and `MinimalBlockDimension>`.

50.11.5 PrimitiveIndexIrreducibleSolvableGroup

▷ `PrimitiveIndexIrreducibleSolvableGroup` (global variable)

This variable provides a way to get from irreducible solvable groups to primitive groups and vice versa. For the group G = `IrreducibleSolvableGroup( n, p, k )` and $d = p^n$, the entry `PrimitiveIndexIrreducibleSolvableGroup[d][i]` gives the index number of the semidirect product $p^n : G$ in the library of primitive groups.

Searching for an index in this list with `Position` (21.16.1) gives the translation in the other direction.

50.11.6 IrreducibleSolvableGroup

▷ `IrreducibleSolvableGroup(n, p, i)` (function)

This function is obsolete, because for $n = 2$, $p = 13$, two groups were missing from the underlying database. It has been replaced by the function `IrreducibleSolvableGroupMS` (50.11.1). Please note that the latter function does not guarantee any ordering of the groups in the database. However, for values of n, p, and i admissible to `IrreducibleSolvableGroup`, `IrreducibleSolvableGroupMS` (50.11.1) returns a representative of the same conjugacy class of subgroups of GL(n, p) as `IrreducibleSolvableGroup` did before.

50.12 Irreducible Maximal Finite Integral Matrix Groups

A library of irreducible maximal finite integral matrix groups is provided with GAP. It contains $\mathbb{Q}$-class representatives for all of these groups of dimension at most 31, and $\mathbb{Z}$-class representatives for those of dimension at most 11 or of dimension 13, 17, 19, or 23.

The groups provided in this library have been determined by Wilhelm Plesken, partially as joint work with Michael Pohst, or by members of his institute (Lehrstuhl B für Mathematik, RWTH Aachen). In particular, the data for the groups of dimensions 2 to 9 have been taken from the output of computer calculations which they performed in 1979 (see [PP77], [PP80]). The $\mathbb{Z}$-class representatives of the groups of dimension 10 have been determined and computed by Bernd Souvignier ([Sou94]), and those of dimensions 11, 13, and 17 have been recomputed for this library from the circulant Gram matrices given in [Ple85], using the stand-alone programs for the computation of short vectors and Bravais groups which have been developed in Plesken's institute. The $\mathbb{Z}$-class representatives of the groups of dimensions 19 and 23 had already been determined in [Ple85]. Gabriele Nebe has recomputed them for us. Her main contribution to this library, however, is that she has determined and computed the $\mathbb{Q}$-class representatives of the groups of non-prime dimensions between 12 and 24 and the groups of dimensions 25 to 31 (see [PN95], [NP95b], [Neb95], [Neb96]).

The library has been brought into GAP format by Volkmar Felsch. He has applied several GAP routines to check certain consistency of the data. However, the credit and responsibility for the lists remain with the authors. We are grateful to Wilhelm Plesken, Gabriele Nebe, and Bernd Souvignier for supplying their results to GAP.

In the preceding acknowledgement, we used some notations that will also be needed in the sequel. We first define these.

Any integral matrix group G of dimension n is a subgroup of $GL_n(\mathbb{Z})$ as well as of $GL_n(\mathbb{Q})$ and hence lies in some conjugacy class of integral matrix groups under $GL_n(\mathbb{Z})$ and also in some conjugacy class of rational matrix groups under $GL_n(\mathbb{Q})$. As usual, we call these classes the $\mathbb{Z}$-class and the $\mathbb{Q}$-class of G, respectively. Note that any conjugacy class of subgroups of $GL_n(\mathbb{Q})$ contains at least one $\mathbb{Z}$-class of subgroups of $GL_n(\mathbb{Z})$ and hence can be considered as the $\mathbb{Q}$-class of some integral matrix group.

In the context of this library we are only concerned with $\mathbb{Z}$-classes and $\mathbb{Q}$-classes of subgroups of $GL_n(\mathbb{Z})$ which are irreducible and maximal finite in $GL_n(\mathbb{Z})$ (we will call them *i.m.f.* subgroups of $GL_n(\mathbb{Z})$). We can distinguish two types of these groups:

First, there are those i.m.f. subgroups of $GL_n(\mathbb{Z})$ which are also maximal finite subgroups of $GL_n(\mathbb{Q})$. Let us denote the set of their $\mathbb{Q}$-classes by $Q_1(n)$. It is clear from the above remark that $Q_1(n)$ just consists of the $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Q})$.

Secondly, there is the set $Q_2(n)$ of the $\mathbb{Q}$-classes of the remaining i.m.f. subgroups of $GL_n(\mathbb{Z})$, i.e., of those which are not maximal finite subgroups of $GL_n(\mathbb{Q})$. For any such group G, say, there is at least one class $C \in Q_1(n)$ such that G is conjugate under $\mathbb{Q}$ to a proper subgroup of some group $H \in C$. In fact, the class C is uniquely determined for any group G occurring in the library (though there seems to be no reason to assume that this property should hold in general). Hence we may call C the *rational i.m.f. class* of G. Finally, we will denote the number of classes in $Q_1(n)$ and $Q_2(n)$ by $q_1(n)$ and $q_2(n)$, respectively.

As an example, let us consider the case $n = 4$. There are 6 $\mathbb{Z}$-classes of i.m.f. subgroups of $GL_4(\mathbb{Z})$ with representative subgroups $G_1, \ldots, G_6$ of isomorphism types $G_1 \cong W(F_4)$, $G_2 \cong D_{12} \wr C_2$, $G_3 \cong G_4 \cong C_2 \times S_5$, $G_5 \cong W(B_4)$, and $G_6 \cong (D_{12} Y D_{12}) : C_2$. The corresponding $\mathbb{Q}$-classes, $R_1, \ldots, R_6$, say, are pairwise different except that R_3 coincides with R_4. The groups G_1, G_2, and G_3 are i.m.f. subgroups of $GL_4(\mathbb{Q})$, but G_5 and G_6 are not because they are conjugate under $GL_4(\mathbb{Q})$ to proper subgroups of G_1 and G_2, respectively. So we have $Q_1(4) = \{R_1, R_2, R_3\}$, $Q_2(4) = \{R_5, R_6\}$, $q_1(4) = 3$, and $q_2(4) = 2$.

The $q_1(n)$ $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Q})$ have been determined for each dimension $n \leq 31$. The current GAP library provides integral representative groups for all these classes. Moreover, all $\mathbb{Z}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Z})$ are known for $n \leq 11$ and for $n \in \{13, 17, 19, 23\}$. For these dimensions, the library offers integral representative groups for all $\mathbb{Q}$-classes in $Q_1(n)$ and $Q_2(n)$ as well as for all $\mathbb{Z}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Z})$.

Any group G of dimension n given in the library is represented as the automorphism group $G = Aut(F, L) = \{g \in GL_n(\mathbb{Z}) \mid Lg = L, gFg^{tr} = F\}$ of a positive definite symmetric $n \times n$ matrix $F \in \mathbb{Z}^{n \times n}$ on an n-dimensional lattice $L \cong \mathbb{Z}^{1 \times n}$ (for details see e.g. [PN95]). GAP provides for G a list of matrix generators and the *Gram matrix* F.

The positive definite quadratic form defined by F defines a *norm* vFv^{tr} for each vector $v \in L$, and there is only a finite set of vectors of minimal norm. These vectors are often simply called the *short vectors*. Their set splits into orbits under G, and G being irreducible acts faithfully on each of these orbits by multiplication from the right. GAP provides for each of these orbits the orbit size and a representative vector.

Like most of the other GAP libraries, the library of i.m.f. integral matrix groups supplies an extraction function, `ImfMatrixGroup`. However, as the library involves only 525 different groups, there

is no need for a selection or an example function. Instead, there are two functions, `ImfInvariants` (50.12.3) and `DisplayImfInvariants` (50.12.2), which provide some $\mathbb{Z}$-class invariants that can be extracted from the library without actually constructing the representative groups themselves. The difference between these two functions is that the latter one displays the resulting data in some easily readable format, whereas the first one returns them as record components so that you can properly access them.

We shall give an individual description of each of the library functions, but first we would like to insert a short remark concerning their names: Any self-explaining name of a function handling *irreducible maximal finite integral matrix groups* would have to include this term in full length and hence would grow extremely long. Therefore we have decided to use the abbreviation `Imf` instead in order to restrict the names to some reasonable length.

The first three functions can be used to formulate loops over the classes.

50.12.1 ImfNumberQQClasses

▷ `ImfNumberQQClasses(`*dim*`)` (function)
▷ `ImfNumberQClasses(`*dim*`)` (function)
▷ `ImfNumberZClasses(`*dim*`, q)` (function)

`ImfNumberQQClasses` returns the number $q_1($*dim*$)$ of $\mathbb{Q}$-classes of i.m.f. rational matrix groups of dimension *dim*. Valid values of *dim* are all positive integers up to 31.

Note: In order to enable you to loop just over the classes belonging to $Q_1($*dim*$)$, we have arranged the list of $\mathbb{Q}$-classes of dimension *dim* for any dimension *dim* in the library such that, whenever the classes of $Q_2($*dim*$)$ are known, too, i.e., in the cases $dim \leq 11$ or $dim \in \{13,17,19,23\}$, the classes of $Q_1($*dim*$)$ precede those of $Q_2($*dim*$)$ and hence are numbered from 1 to $q_1($*dim*$)$.

`ImfNumberQClasses` returns the number of $\mathbb{Q}$-classes of groups of dimension *dim* which are available in the library. If $dim \leq 11$ or $dim \in \{13,17,19,23\}$, this is the number $q_1($*dim*$)+q_2($*dim*$)$ of $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_{dim}(\mathbb{Z})$. Otherwise, it is just the number $q_1($*dim*$)$ of $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_{dim}(\mathbb{Q})$. Valid values of *dim* are all positive integers up to 31.

`ImfNumberZClasses` returns the number of $\mathbb{Z}$-classes in the q-th $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*. Valid values of *dim* are all positive integers up to 11 and all primes up to 23.

50.12.2 DisplayImfInvariants

▷ `DisplayImfInvariants(`*dim*`, q[, z])` (function)

`DisplayImfInvariants` displays the following $\mathbb{Z}$-class invariants of the groups in the z-th $\mathbb{Z}$-class in the q-th $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*:

- its $\mathbb{Z}$-class number in the form *dim*`.`q`.`z, if *dim* is at most 11 or a prime at most 23, or its $\mathbb{Q}$-class number in the form *dim*`.`q, else,
- a message if the group is solvable,
- the size of the group,
- the isomorphism type of the group,

- the elementary divisors of the associated quadratic form,
- the sizes of the orbits of short vectors (these sizes are the degrees of the faithful permutation representations which you may construct using the functions `IsomorphismPermGroup` (50.12.5) or `IsomorphismPermGroupImfGroup` (50.12.6) below),
- the norm of the associated short vectors,
- only in case that the group is not an i.m.f. group in $GL_n(\mathbb{Q})$: an appropriate message, including the $\mathbb{Q}$-class number of the corresponding rational i.m.f. class.

If you specify the value 0 for any of the parameters *dim*, *q*, or *z*, the command will loop over all available dimensions, $\mathbb{Q}$-classes of given dimension, or $\mathbb{Z}$-classes within the given $\mathbb{Q}$-class, respectively. Otherwise, the values of the arguments must be in range. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \notin \{13,17,19,23\}$. The default value of *z* is 1. This value of *z* will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the *q*-th $\mathbb{Q}$-class. The greatest legal value of *dim* is 31.

Example

```
gap> DisplayImfInvariants( 3, 1, 0 );
#I Z-class 3.1.1:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = W(B3)
#I   elementary divisors = 1^3
#I   orbit size = 6, minimal norm = 1
#I Z-class 3.1.2:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = C2 x W(A3)
#I   elementary divisors = 1*4^2
#I   orbit size = 8, minimal norm = 3
#I Z-class 3.1.3:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = C2 x W(A3)
#I   elementary divisors = 1^2*4
#I   orbit size = 12, minimal norm = 2
gap> DisplayImfInvariants( 8, 15, 1 );
#I Z-class 8.15.1:  Solvable, size = 2^5*3^4
#I   isomorphism type = C2 x (S3 wr S3)
#I   elementary divisors = 1*3^3*9^3*27
#I   orbit size = 54, minimal norm = 8
#I   not maximal finite in GL(8,Q), rational imf class is 8.5
gap> DisplayImfInvariants( 20, 23 );
#I Q-class 20.23:  Size = 2^5*3^2*5*11
#I   isomorphism type = (PSL(2,11) x D12).C2
#I   elementary divisors = 1^18*11^2
#I   orbit size = 3*660 + 2*1980 + 2640 + 3960, minimal norm = 4
```

Note that the function `DisplayImfInvariants` uses a kind of shorthand to display the elementary divisors. E. g., the expression `1*3^3*9^3*27` in the preceding example stands for the elementary divisors $1,3,3,3,9,9,9,27$. (See also the next example which shows that the function `ImfInvariants` (50.12.3) provides the elementary divisors in form of an ordinary GAP list.)

In the description of the isomorphism types the following notations are used:

A x *B*
: denotes a direct product of a group *A* by a group *B*,

A `subd` *B*
: denotes a subdirect product of *A* by *B*,

A `Y` *B*
: denotes a central product of *A* by *B*,

A `wr` *B*
: denotes a wreath product of *A* by *B*,

A`:`*B* denotes a split extension of *A* by *B*,

A`.`*B* denotes just an extension of *A* by *B* (split or nonsplit).

The groups involved are

- the cyclic groups C_n, dihedral groups D_n, and generalized quaternion groups Q_n of order n, denoted by `Cn`, `Dn`, and `Qn`, respectively,
- the alternating groups A_n and symmetric groups S_n of degree n, denoted by `An` and `Sn`, respectively,
- the linear groups $GL_n(q)$, $PGL_n(q)$, $SL_n(q)$, and $PSL_n(q)$, denoted by `GL(n,q)`, `PGL(n,q)`, `SL(n,q)`, and `PSL(n,q)`, respectively,
- the unitary groups $SU_n(q)$ and $PSU_n(q)$, denoted by `SU(n,q)` and `PSU(n,q)`, respectively,
- the symplectic groups $Sp(n,q)$ and $PSp(n,q)$, denoted by `Sp(n,q)` and `PSp(n,q)`, respectively,
- the orthogonal groups $O_8^+(2)$ and $PO_8^+(2)$, denoted by `O+(8,2)` and `PO+(8,2)`, respectively,
- the extraspecial groups 2_+^{1+8}, 3_+^{1+2}, 3_+^{1+4}, and 5_+^{1+2}, denoted by `2+^(1+8)`, `3+^(1+2)`, `3+^(1+4)`, and `5+^(1+2)`, respectively,
- the Chevalley group $G_2(3)$, denoted by `G2(3)`,
- the twisted Chevalley group ${}^3D_4(2)$, denoted by `3D4(2)`,
- the Suzuki group $Sz(8)$, denoted by `Sz(8)`,
- the Weyl groups $W(A_n)$, $W(B_n)$, $W(D_n)$, $W(E_n)$, and $W(F_4)$, denoted by `W(An)`, `W(Bn)`, `W(Dn)`, `W(En)`, and `W(F4)`, respectively,
- the sporadic simple groups Co_1, Co_2, Co_3, HS, J_2, M_{12}, M_{22}, M_{23}, M_{24}, and Mc, denoted by `Co1`, `Co2`, `Co3`, `HS`, `J2`, `M12`, `M22`, `M23`, `M24`, and `Mc`, respectively,
- a point stabilizer of index 11 in M_{11}, denoted by `M10`.

As mentioned above, the data assembled by the function `DisplayImfInvariants` are "cheap data" in the sense that they can be provided by the library without loading any of its large matrix files or performing any matrix calculations. The following function allows you to get proper access to these cheap data instead of just displaying them.

50.12.3 ImfInvariants

▷ `ImfInvariants(dim, q[, z])` (function)

`ImfInvariants` returns a record which provides some $\mathbb{Z}$-class invariants of the groups in the `z`-th $\mathbb{Z}$-class in the `q`-th $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension `dim`. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of `z` is 1. This value of `z` will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the `q`-th $\mathbb{Q}$-class. The greatest legal value of `dim` is 31.

The resulting record contains six or seven components:

`size`
: the size of any representative group `G`,

`isSolvable`
: is `true` if `G` is solvable,

`isomorphismType`
: a text string describing the isomorphism type of `G` (in the same notation as used by the function `DisplayImfInvariants` above),

`elementaryDivisors`
: the elementary divisors of the associated Gram matrix `F` (in the same format as the result of the function `ElementaryDivisorsMat` (24.9.1),

`minimalNorm`
: the norm of the associated short vectors,

`sizesOrbitsShortVectors`
: the sizes of the orbits of short vectors under `F`,

`maximalQClass`
: the $\mathbb{Q}$-class number of an i.m.f. group in $GL_n(\mathbb{Q})$ that contains `G` as a subgroup (only in case that not `G` itself is an i.m.f. subgroup of $GL_n(\mathbb{Q})$).

Note that four of these data, namely the group size, the solvability, the isomorphism type, and the corresponding rational i.m.f. class, are not only $\mathbb{Z}$-class invariants, but also $\mathbb{Q}$-class invariants.

Note further that, though the isomorphism type is a $\mathbb{Q}$-class invariant, you will sometimes get different descriptions for different $\mathbb{Z}$-classes of the same $\mathbb{Q}$-class (as, e.g., for the classes 3.1.1 and 3.1.2 in the last example above). The purpose of this behaviour is to provide some more information about the underlying lattices.

Example

```
gap> ImfInvariants( 8, 15, 1 );
rec( elementaryDivisors := [ 1, 3, 3, 3, 9, 9, 9, 27 ],
  isSolvable := true, isomorphismType := "C2 x (S3 wr S3)",
  maximalQClass := 5, minimalNorm := 8, size := 2592,
  sizesOrbitsShortVectors := [ 54 ] )
gap> ImfInvariants( 24, 1 ).size;
104093968527333324538616217600000
```

```
gap> ImfInvariants( 23, 5, 2 ).sizesOrbitsShortVectors;
[ 552, 53130 ]
gap> for i in [ 1 .. ImfNumberQClasses( 22 ) ] do
>    Print( ImfInvariants( 22, i ).isomorphismType, "\n" ); od;
C2 wr S22 = W(B22)
(C2 x PSU(6,2)).S3
(C2 x S3) wr S11 = (C2 x W(A2)) wr S11
(C2 x S12) wr C2 = (C2 x W(A11)) wr C2
C2 x S3 x S12 = C2 x W(A2) x W(A11)
(C2 x HS).C2
(C2 x Mc).C2
C2 x S23 = C2 x W(A22)
C2 x PSL(2,23)
C2 x PSL(2,23)
C2 x PGL(2,23)
C2 x PGL(2,23)
```

50.12.4 ImfMatrixGroup

▷ ImfMatrixGroup(*dim, q[, z]*) (function)

ImfMatrixGroup is the essential extraction function of this library (note that its name has been changed from ImfMatGroup in GAP 3 to ImfMatrixGroup in GAP 4). It returns a representative group, G say, of the z-th $\mathbb{Z}$-class in the q-th $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the q-th $\mathbb{Q}$-class. The greatest legal value of *dim* is 31.

Example

```
gap> G := ImfMatrixGroup( 5, 1, 3 );
ImfMatrixGroup(5,1,3)
gap> for m in GeneratorsOfGroup( G ) do PrintArray( m ); od;
[ [  -1,   0,   0,   0,   0 ],
  [   0,   1,   0,   0,   0 ],
  [   0,   0,   0,   1,   0 ],
  [  -1,  -1,  -1,  -1,   2 ],
  [  -1,   0,   0,   0,   1 ] ]
[ [  0,  1,  0,  0,  0 ],
  [  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  1,  0 ],
  [  1,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1 ] ]
```

The attributes Size (30.4.6) and IsSolvable will be properly set in the resulting matrix group G. In addition, it has two attributes IsImfMatrixGroup and ImfRecord where the first one is just a logical flag set to true and the latter one is a record. Except for the group size and the solvability flag, this record contains the same components as the resulting record of the function ImfInvariants (50.12.3) described above, namely the components isomorphismType, elementaryDivisors, minimalNorm, and sizesOrbitsShortVectors and, if G is not a rational i.m.f. group, maximalQClass. Moreover, it has the two components

`form`
the associated Gram matrix F, and

`repsOrbitsShortVectors`
representatives of the orbits of short vectors under F.

The last one of these components will be required by the function `IsomorphismPermGroup` (50.12.5) below.

Example

```
gap> Size( G );
3840
gap> imf := ImfRecord( G );;
gap> imf.isomorphismType;
"C2 wr S5 = C2 x W(D5)"
gap> PrintArray( imf.form );
[ [  4,  0,  0,  0,  2 ],
  [  0,  4,  0,  0,  2 ],
  [  0,  0,  4,  0,  2 ],
  [  0,  0,  0,  4,  2 ],
  [  2,  2,  2,  2,  5 ] ]
gap> imf.elementaryDivisors;
[ 1, 4, 4, 4, 4 ]
gap> imf.minimalNorm;
4
```

If you want to perform calculations in such a matrix group G you should be aware of the fact that the permutation group routines of GAP are much more efficient than the matrix group routines. Hence we recommend that you do your computations, whenever possible, in the isomorphic permutation group which is induced by the action of G on one of the orbits of the associated short vectors. You may call one of the following functions `IsomorphismPermGroup` (50.12.5) or `IsomorphismPermGroupImfGroup` (50.12.6) to get an isomorphism to such a permutation group (note that these GAP 4 functions have replaced the GAP 3 functions `PermGroup` and `PermGroupImfGroup`).

50.12.5 IsomorphismPermGroup (for Imf matrix groups)

▷ `IsomorphismPermGroup(G)` (method)

returns an isomorphism, φ say, from the given i.m.f. integral matrix group G to a permutation group $P := \varphi(G)$ acting on a minimal orbit, S say, of short vectors of G such that each matrix $m \in G$ is mapped to the permutation induced by its action on S.

Note that in case of a large orbit the construction of φ may be space and time consuming. Fortunately, there are only six $\mathbb{Q}$-classes in the library for which the smallest orbit of short vectors is of size greater than 20000, the worst case being the orbit of size 196560 for the Leech lattice ($dim = 24$, $q = 3$).

The inverse isomorphism φ^{-1} from P to G is constructed by determining a $\mathbb{Q}$-base $B \subset S$ of $\mathbb{Q}^{1\times dim}$ in S and, in addition, the associated base change matrix M which transforms B into the standard base of $\mathbb{Z}^{1\times dim}$. This allows a simple computation of the preimage $\varphi^{-1}(p)$ of any permutation $p \in P$, as follows. If, for $1 \leq i \leq dim$, b_i is the position number in S of the i-th base vector in B, it suffices to

look up the vector whose position number in S is the image of b_i under p and to multiply this vector by M to get the i-th row of $\varphi^{-1}(p)$.

You may use the functions `Image` (32.4.6) and `PreImage` (32.5.6) to switch from G to P and back from P to G.

As an example, let us continue the preceding example and compute the solvable residuum of the group G.

Example

```
gap> # Perform the computations in an isomorphic permutation group.
gap> phi := IsomorphismPermGroup( G );;
gap> P := Image( phi );
Group([ (1,7,6)(2,9)(4,5,10), (2,3,4,5)(6,9,8,7) ])
gap> D := DerivedSubgroup( P );
Group([ (1,2,10,9)(3,8)(4,5)(6,7), (1,6)(2,7,9,4)(3,8)(5,10), (1,9)
(2,10)(3,6,8,5)(4,7) ])
gap> Size( D );
960
gap> IsPerfectGroup( D );
true
gap> # We have found the solvable residuum of P,
gap> # now move the results back to the matrix group G.
gap> R := PreImage( phi, D );
<matrix group of size 960 with 3 generators>
gap> for m in GeneratorsOfGroup( R ) do PrintArray( m ); od;
[ [  -1,  -1,  -1,  -1,   2 ],
  [   0,  -1,   0,   0,   0 ],
  [   0,   0,   0,   1,   0 ],
  [   0,   0,   1,   0,   0 ],
  [  -1,  -1,   0,   0,   1 ] ]
[ [   0,   0,  -1,   0,   0 ],
  [   0,  -1,   0,   0,   0 ],
  [   1,   0,   0,   0,   0 ],
  [  -1,  -1,  -1,  -1,   2 ],
  [   0,  -1,  -1,   0,   1 ] ]
[ [  -1,  -1,  -1,  -1,   2 ],
  [   0,   0,   0,  -1,   0 ],
  [   0,   0,  -1,   0,   0 ],
  [   0,   1,   0,   0,   0 ],
  [   0,   0,  -1,  -1,   1 ] ]
```

50.12.6 IsomorphismPermGroupImfGroup

▷ `IsomorphismPermGroupImfGroup(`*G*`, `*n*`)` (function)

`IsomorphismPermGroupImfGroup` returns an isomorphism, φ say, from the given i.m.f. integral matrix group *G* to a permutation group P acting on the *n*-th orbit, S say, of short vectors of *G* such that each matrix $m \in$ *G* is mapped to the permutation induced by its action on S.

The only difference to the above function `IsomorphismPermGroup` (50.12.5) is that you can specify the orbit to be used. In fact, as the orbits of short vectors are sorted by increasing sizes, the function `IsomorphismPermGroup( G )` has been implemented such that it is equivalent to `IsomorphismPermGroupImfGroup( G, 1 )`.

Example

```
gap> ImfInvariants( 12, 9 ).sizesOrbitsShortVectors;
[ 120, 300 ]
gap> G := ImfMatrixGroup( 12, 9 );
ImfMatrixGroup(12,9)
gap> phi1 := IsomorphismPermGroupImfGroup( G, 1 );;
gap> P1 := Image( phi1 );
<permutation group of size 2400 with 2 generators>
gap> LargestMovedPoint( P1 );
120
gap> phi2 := IsomorphismPermGroupImfGroup( G, 2 );;
gap> P2 := Image( phi2 );
<permutation group of size 2400 with 2 generators>
gap> LargestMovedPoint( P2 );
300
```

Chapter 51

Semigroups and Monoids

This chapter describes functions for creating semigroups and monoids and determining information about them.

51.1 Semigroups

51.1.1 IsSemigroup

▷ `IsSemigroup(`*`D`*`)` (property)

returns `true` if the object *`D`* is a semigroup. A *semigroup* is a magma (see 35) with associative multiplication.

51.1.2 Semigroup

▷ `Semigroup(`*`gen1, gen2, ...`*`)` (function)
▷ `Semigroup(`*`gens`*`)` (function)

In the first form, `Semigroup` returns the semigroup generated by the arguments *`gen1`*, *`gen2`*, ..., that is, the closure of these elements under multiplication. In the second form, `Semigroup` returns the semigroup generated by the elements in the homogeneous list *`gens`*; a square matrix as only argument is treated as one generator, not as a list of generators.

It is *not* checked whether the underlying multiplication is associative, use `Magma` (35.2.1) and `IsAssociative` (35.4.7) if you want to check whether a magma is in fact a semigroup.

Example

```
gap> a:= Transformation( [ 2, 3, 4, 1 ] );
Transformation( [ 2, 3, 4, 1 ] )
gap> b:= Transformation( [ 2, 2, 3, 4 ] );
Transformation( [ 2, 2 ] )
gap> s:= Semigroup(a, b);
<transformation semigroup on 4 pts with 2 generators>
```

51.1.3 Subsemigroup

▷ `Subsemigroup(`*`S, gens`*`)` (function)
▷ `SubsemigroupNC(`*`S, gens`*`)` (function)

are just synonyms of Submagma (35.2.7) and SubmagmaNC (35.2.7), respectively.

Example

```
gap> a:=GeneratorsOfSemigroup(s)[1];
Transformation( [ 2, 3, 4, 1 ] )
gap> t:=Subsemigroup(s,[a]);
<commutative transformation semigroup on 4 pts with 1 generator>
```

51.1.4 IsSubsemigroup

▷ IsSubsemigroup(*S*, *T*) (operation)

Returns: true or false.

This operation returns true if the semigroup *T* is a subsemigroup of the semigroup *S* and false if it is not.

Example

```
gap> f:=Transformation( [ 5, 6, 7, 1, 4, 3, 2, 7 ] );
Transformation( [ 5, 6, 7, 1, 4, 3, 2, 7 ] )
gap> T:=Semigroup(f);;
gap> IsSubsemigroup(FullTransformationSemigroup(4), T);
false
gap> S:=Semigroup(f);; T:=Semigroup(f^2);;
gap> IsSubsemigroup(S, T);
true
```

51.1.5 SemigroupByGenerators

▷ SemigroupByGenerators(*gens*) (operation)

is the underlying operation of Semigroup (51.1.2).

51.1.6 AsSemigroup

▷ AsSemigroup(*C*) (attribute)

If *C* is a collection whose elements form a semigroup (see IsSemigroup (51.1.1)) then AsSemigroup returns this semigroup. Otherwise fail is returned.

51.1.7 AsSubsemigroup

▷ AsSubsemigroup(*D*, *C*) (operation)

Let *D* be a domain and *C* a collection. If *C* is a subset of *D* that forms a semigroup then AsSubsemigroup returns this semigroup, with parent *D*. Otherwise fail is returned.

51.1.8 GeneratorsOfSemigroup

▷ GeneratorsOfSemigroup(*S*) (attribute)

Semigroup generators of a semigroup *D* are the same as magma generators, see GeneratorsOfMagma (35.4.1).

Example

```
gap> GeneratorsOfSemigroup(s);
[ Transformation( [ 2, 3, 4, 1 ] ), Transformation( [ 2, 2 ] ) ]
gap> GeneratorsOfSemigroup(t);
[ Transformation( [ 2, 3, 4, 1 ] ) ]
```

51.1.9 IsGeneratorsOfSemigroup

▷ IsGeneratorsOfSemigroup(*C*) (property)

This property reflects wheter the list or collection *C* generates a semigroup. IsAssociativeElementCollection (31.15.1) implies IsGeneratorsOfSemigroup, but is not used directly in semigroup code, because of conflicts with matrices.

Example

```
gap> IsGeneratorsOfSemigroup([Transformation([2,3,1])]);
true
```

51.1.10 FreeSemigroup

▷ FreeSemigroup(*[wfilt,]rank[, name]*) (function)
▷ FreeSemigroup(*[wfilt,]name1, name2, ...*) (function)
▷ FreeSemigroup(*[wfilt,]names*) (function)
▷ FreeSemigroup(*[wfilt,]infinity, name, init*) (function)

Called with a positive integer *rank*, FreeSemigroup returns a free semigroup on *rank* generators. If the optional argument *name* is given then the generators are printed as *name*1, *name*2 etc., that is, each name is the concatenation of the string *name* and an integer from 1 to *range*. The default for *name* is the string "s".

Called in the second form, FreeSemigroup returns a free semigroup on as many generators as arguments, printed as *name1*, *name2* etc.

Called in the third form, FreeSemigroup returns a free semigroup on as many generators as the length of the list *names*, the i-th generator being printed as *names*$[i]$.

Called in the fourth form, FreeSemigroup returns a free semigroup on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either IsSyllableWordsFamily (37.6.6) or IsLetterWordsFamily (37.6.2) or IsWLetterWordsFamily (37.6.4) or IsBLetterWordsFamily (37.6.4). This filter then specifies the representation used for the elements of the free semigroup (see 37.6). If no such filter is given, a letter representation is used.

Example

```
gap> f1 := FreeSemigroup( 3 );
<free semigroup on the generators [ s1, s2, s3 ]>
gap> f2 := FreeSemigroup( 3 , "generator" );
<free semigroup on the generators
[ generator1, generator2, generator3 ]>
gap> f3 := FreeSemigroup( "gen1" , "gen2" );
```

```
<free semigroup on the generators [ gen1, gen2 ]>
gap> f4 := FreeSemigroup( ["gen1" , "gen2"] );
<free semigroup on the generators [ gen1, gen2 ]>
```

Also see Chapter 51.

Each free object defines a unique alphabet (and a unique family of words). Its generators are the letters of this alphabet, thus words of length one.

Example

```
gap> FreeGroup( 5 );
<free group on the generators [ f1, f2, f3, f4, f5 ]>
gap> FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> FreeGroup( infinity );
<free group with infinity generators>
gap> FreeSemigroup( "x", "y" );
<free semigroup on the generators [ x, y ]>
gap> FreeMonoid( 7 );
<free monoid on the generators [ m1, m2, m3, m4, m5, m6, m7 ]>
```

Remember that names are just a help for printing and do not necessarily distinguish letters. It is possible to create arbitrarily weird situations by choosing strange names for the letters.

Example

```
gap> f:= FreeGroup( "x", "x" );  gens:= GeneratorsOfGroup( f );;
<free group on the generators [ x, x ]>
gap> gens[1] = gens[2];
false
gap> f:= FreeGroup( "f1*f2", "f2^-1", "Group( [ f1, f2 ] )" );
<free group on the generators [ f1*f2, f2^-1, Group( [ f1, f2 ] ) ]>
gap> gens:= GeneratorsOfGroup( f );;
gap> gens[1]*gens[2];
f1*f2*f2^-1
gap> gens[1]/gens[3];
f1*f2*Group( [ f1, f2 ] )^-1
gap> gens[3]/gens[1]/gens[2];
Group( [ f1, f2 ] )*f1*f2^-1*f2^-1^-1
```

51.1.11 SemigroupByMultiplicationTable

▷ `SemigroupByMultiplicationTable(`*A*`)` (function)

returns the semigroup whose multiplication is defined by the square matrix *A* (see `MagmaByMultiplicationTable` (35.3.1)) if such a semigroup exists. Otherwise `fail` is returned.

Example

```
gap> SemigroupByMultiplicationTable([[1,2,3],[2,3,1],[3,1,2]]);
<semigroup with 3 generators>
gap> SemigroupByMultiplicationTable([[1,2,3],[2,3,1],[3,2,1]]);
fail
```

51.2 Monoids

51.2.1 IsMonoid

▷ IsMonoid(*D*) (property)

A *monoid* is a magma-with-one (see 35) with associative multiplication.

51.2.2 Monoid

▷ Monoid(*gen1, gen2, ...*) (function)
▷ Monoid(*gens[, id]*) (function)

In the first form, Monoid returns the monoid generated by the arguments *gen1*, *gen2*, ..., that is, the closure of these elements under multiplication and taking the 0-th power. In the second form, Monoid returns the monoid generated by the elements in the homogeneous list *gens*; a square matrix as only argument is treated as one generator, not as a list of generators. In the second form, the identity element *id* may be given as the second argument.

It is *not* checked whether the underlying multiplication is associative, use MagmaWithOne (35.2.2) and IsAssociative (35.4.7) if you want to check whether a magma-with-one is in fact a monoid.

51.2.3 Submonoid

▷ Submonoid(*M, gens*) (function)
▷ SubmonoidNC(*M, gens*) (function)

are just synonyms of SubmagmaWithOne (35.2.8) and SubmagmaWithOneNC (35.2.8), respectively.

51.2.4 MonoidByGenerators

▷ MonoidByGenerators(*gens[, one]*) (operation)

is the underlying operation of Monoid (51.2.2).

51.2.5 AsMonoid

▷ AsMonoid(*C*) (attribute)

If *C* is a collection whose elements form a monoid (see IsMonoid (51.2.1)) then AsMonoid returns this monoid. Otherwise fail is returned.

51.2.6 AsSubmonoid

▷ AsSubmonoid(*D, C*) (operation)

Let *D* be a domain and *C* a collection. If *C* is a subset of *D* that forms a monoid then AsSubmonoid returns this monoid, with parent *D*. Otherwise fail is returned.

51.2.7 GeneratorsOfMonoid

▷ `GeneratorsOfMonoid(`*M*`)` (attribute)

Monoid generators of a monoid *M* are the same as magma-with-one generators (see `GeneratorsOfMagmaWithOne` (35.4.2)).

51.2.8 TrivialSubmonoid

▷ `TrivialSubmonoid(`*M*`)` (attribute)

is just a synonym for `TrivialSubmagmaWithOne` (35.4.13).

51.2.9 FreeMonoid

▷ `FreeMonoid(`*[wfilt,]rank[, name]*`)` (function)
▷ `FreeMonoid(`*[wfilt,]name1, name2, ...*`)` (function)
▷ `FreeMonoid(`*[wfilt,]names*`)` (function)
▷ `FreeMonoid(`*[wfilt,]infinity, name, init*`)` (function)

Called with a positive integer *rank*, `FreeMonoid` returns a free monoid on *rank* generators. If the optional argument *name* is given then the generators are printed as *name*`1`, *name*`2` etc., that is, each name is the concatenation of the string *name* and an integer from `1` to *range*. The default for *name* is the string `"m"`.

Called in the second form, `FreeMonoid` returns a free monoid on as many generators as arguments, printed as *name1*, *name2* etc.

Called in the third form, `FreeMonoid` returns a free monoid on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*`[`*i*`]`.

Called in the fourth form, `FreeMonoid` returns a free monoid on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either `IsSyllableWordsFamily` (37.6.6) or `IsLetterWordsFamily` (37.6.2) or `IsWLetterWordsFamily` (37.6.4) or `IsBLetterWordsFamily` (37.6.4). This filter then specifies the representation used for the elements of the free monoid (see 37.6). If no such filter is given, a letter representation is used.

Also see Chapter 51.

51.2.10 MonoidByMultiplicationTable

▷ `MonoidByMultiplicationTable(`*A*`)` (function)

returns the monoid whose multiplication is defined by the square matrix *A* (see `MagmaByMultiplicationTable` (35.3.1)) if such a monoid exists. Otherwise `fail` is returned.

Example

```
gap> MonoidByMultiplicationTable([[1,2,3],[2,3,1],[3,1,2]]);
<monoid with 3 generators>
gap> MonoidByMultiplicationTable([[1,2,3],[2,3,1],[1,3,2]]);
fail
```

51.3 Inverse semigroups and monoids

51.3.1 InverseSemigroup

▷ `InverseSemigroup(`*obj1, obj2, ...*`)` (function)

Returns: An inverse semigroup.

If *obj1*, *obj2*, ... are (any combination) of associative elements with unique semigroup inverses, semigroups of such elements, or collections of such elements, then `InverseSemigroup` returns the inverse semigroup generated by the union of *obj1*, *obj2*, ... This equals the semigroup generated by the union of *obj1*, *obj2*, ... and their inverses.

For example if `S` and `T` are inverse semigroups, then `InverseSemigroup(S, f, Idempotents(T));` is the inverse semigroup generated by `Union(GeneratorsOfInverseSemigroup(S), [f], Idempotents(T)));`.

As present, the only associative elements with unique semigroup inverses, which do not always generate a group, are partial permutations; see Chapter 54.

Example

```
gap> S:=InverseSemigroup(
> PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] ) );;
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 8, 10 ],
> [ 7, 1, 4, 3, 2, 6, 5 ] );;
gap> S:=InverseSemigroup(S, f, Idempotents(SymmetricInverseSemigroup(5)));
<inverse partial perm semigroup on 10 pts with 34 generators>
gap> Size(S);
1233
```

51.3.2 InverseMonoid

▷ `InverseMonoid(`*obj1, obj2, ...*`)` (function)

Returns: An inverse monoid.

If *obj1*, *obj2*, ... are (any combination) of associative elements with unique semigroup inverses, semigroups of such elements, or collections of such elements, then `InverseMonoid` returns the inverse monoid generated by the union of *obj1*, *obj2*, ... This equals the monoid generated by the union of *obj1*, *obj2*, ... and their inverses.

As present, the only associative elements with unique semigroup inverses are partial permutations; see Chapter 54.

For example if `S` and `T` are inverse monoids, then `InverseMonoid(S, f, Idempotents(T));` is the inverse monoid generated by `Union(GeneratorsOfInverseMonoid(S), [f], Idempotents(T)));`.

Example

```
gap> S:=InverseMonoid(
> PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] ) );;
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 8, 10 ],
> [ 7, 1, 4, 3, 2, 6, 5 ] );;
gap> S:=InverseMonoid(S, f, Idempotents(SymmetricInverseSemigroup(5)));
<inverse partial perm monoid on 10 pts with 35 generators>
gap> Size(S);
1243
```

51.3.3 GeneratorsOfInverseSemigroup

▷ GeneratorsOfInverseSemigroup(*S*) (attribute)

Returns: The generators of an inverse semigroup.

If *S* is an inverse semigroup, then GeneratorsOfInverseSemigroup returns the generators used to define *S*, i.e. an inverse semigroup generating set for *S*.

The value of GeneratorsOfSemigroup(*S*), for an inverse semigroup *S*, is the union of inverse semigroup generator and their inverses. So, *S* is the semigroup, as opposed to inverse semigroup, generated by the elements of GeneratorsOfInverseSemigroup(*S*) and their inverses.

If *S* is an inverse monoid, then GeneratorsOfInverseSemigroup returns the generators used to define *S*, as described above, and the identity of *S*.

Example

```
gap> S:=InverseMonoid(
>  PartialPerm( [ 1, 2 ], [ 1, 4 ] ),
>  PartialPerm( [ 1, 2, 4 ], [ 3, 4, 1 ] ) );;
gap> GeneratorsOfSemigroup(S);
[ <identity partial perm on [ 1, 2, 3, 4 ]>, [2,4](1), [2,4,1,3],
  [4,2](1), [3,1,4,2] ]
gap> GeneratorsOfInverseSemigroup(S);
[ [2,4](1), [2,4,1,3], <identity partial perm on [ 1, 2, 3, 4 ]> ]
gap> GeneratorsOfMonoid(S);
[ [2,4](1), [2,4,1,3], [4,2](1), [3,1,4,2] ]
```

51.3.4 GeneratorsOfInverseMonoid

▷ GeneratorsOfInverseMonoid(*S*) (attribute)

Returns: The generators of an inverse monoid.

If *S* is an inverse monoid, then GeneratorsOfInverseMonoid returns the generators used to define *S*, i.e. an inverse monoid generating set for *S*.

There are four different possible generating sets which define an inverse monoid. More precisely, an inverse monoid can be generated as an inverse monoid, inverse semigroup, monoid, or semigroup. The different generating sets in each case can be obtained using GeneratorsOfInverseMonoid, GeneratorsOfInverseSemigroup (51.3.3), GeneratorsOfMonoid (51.2.7), and GeneratorsOfSemigroup (51.1.8), respectively.

Example

```
gap> S:=InverseMonoid(
>  PartialPerm( [ 1, 2 ], [ 1, 4 ] ),
>  PartialPerm( [ 1, 2, 4 ], [ 3, 4, 1 ] ) );;
gap> GeneratorsOfInverseMonoid(S);
[ [2,4](1), [2,4,1,3] ]
```

51.3.5 IsInverseSubsemigroup

▷ IsInverseSubsemigroup(*S*, *T*) (operation)

Returns: true or false.

If the semigroup *T* is an inverse subsemigroup of the semigroup *S*, then this operation returns true.

Example

```
gap> T:=InverseSemigroup(RandomPartialPerm(4));;
gap> IsInverseSubsemigroup(SymmetricInverseSemigroup(4), T);
```

```
true
gap> T:=Semigroup(Transformation( [ 1, 2, 4, 5, 6, 3, 7, 8 ] ),
> Transformation( [ 3, 3, 4, 5, 6, 2, 7, 8 ] ),
> Transformation([ 1, 2, 5, 3, 6, 8, 4, 4 ] ));;
gap> IsInverseSubsemigroup(FullTransformationSemigroup(8), T);
true
```

51.4 Properties of Semigroups

The following functions determine information about semigroups.

51.4.1 IsRegularSemigroup

▷ `IsRegularSemigroup(`*S*`)` (property)

returns `true` if *S* is regular, i.e., if every $\mathscr{D}$-class of *S* is regular.

51.4.2 IsRegularSemigroupElement

▷ `IsRegularSemigroupElement(`*S*`, `*x*`)` (operation)

returns `true` if *x* has a general inverse in *S*, i.e., there is an element $y \in S$ such that $xyx = x$ and $yxy = y$.

51.4.3 InversesOfSemigroupElement

▷ `InversesOfSemigroupElement(`*S*`, `*x*`)` (operation)

Returns: The inverses of an element of a semigroup.

`InversesOfSemigroupElement` returns a list of the inverses of the element *x* in the semigroup *S*.

An element *y* in *S* is an *inverse* of *x* if `x*y*x=x` and `y*x*y=y`. The element *x* has an inverse if and only if *x* is a regular element of *S*.

Example

```
gap> S:=Semigroup([ Transformation( [ 3, 1, 4, 2, 5, 2, 1, 6, 1 ] ),
>  Transformation( [ 5, 7, 8, 8, 7, 5, 9, 1, 9 ] ),
>  Transformation( [ 7, 6, 2, 8, 4, 7, 5, 8, 3 ] ) ]);;
gap> x:=Transformation( [ 3, 1, 4, 2, 5, 2, 1, 6, 1 ] );;
gap> InversesOfSemigroupElement(S, x);
[  ]
gap> IsRegularSemigroupElement(S, x);
false
gap> x:=Transformation( [ 1, 9, 7, 5, 5, 1, 9, 5, 1 ] );;
gap> Set(InversesOfSemigroupElement(S, x));
[ Transformation( [ 1, 2, 3, 5, 5, 1, 3, 5, 2 ] ),
  Transformation( [ 1, 5, 1, 1, 5, 1, 3, 1, 2 ] ),
  Transformation( [ 1, 5, 1, 2, 5, 1, 3, 2, 2 ] ) ]
gap> IsRegularSemigroupElement(S, x);
true
gap> S:=ReesZeroMatrixSemigroup(Group((1,2,3)),
```

```
> [ [ (), () ], [ (), 0 ], [ (), (1,2,3) ] ]);;
gap> x:=ReesZeroMatrixSemigroupElement(S, 2, (1,2,3), 3);;
gap> InversesOfSemigroupElement(S, x);
[ (1,(1,2,3),3), (1,(1,3,2),1), (2,(),3), (2,(1,2,3),1) ]
```

51.4.4 IsSimpleSemigroup

▷ `IsSimpleSemigroup(`*S*`)` (property)

is `true` if and only if the semigroup *S* has no proper ideals.

51.4.5 IsZeroSimpleSemigroup

▷ `IsZeroSimpleSemigroup(`*S*`)` (property)

is `true` if and only if the semigroup has no proper ideals except for 0, where *S* is a semigroup with zero. If the semigroup does not find its zero, then a break-loop is entered.

51.4.6 IsZeroGroup

▷ `IsZeroGroup(`*S*`)` (property)

is `true` if and only if the semigroup *S* is a group with zero adjoined.

51.4.7 IsReesCongruenceSemigroup

▷ `IsReesCongruenceSemigroup(`*S*`)` (property)

returns `true` if *S* is a Rees Congruence semigroup, that is, if all congruences of *S* are Rees Congruences.

51.4.8 IsInverseSemigroup

▷ `IsInverseSemigroup(`*S*`)` (property)
▷ `IsInverseMonoid(`*S*`)` (property)

Returns: `true` or `false`.

A semigroup is an *inverse semigroup* if every element `x` has a unique semigroup inverse, that is, a unique element `y` such that `x*y*x=x` and `y*x*y=y`.

A monoid that happens to be an inverse semigroup is called an *inverse monoid*.

Example

```
gap> S:=Semigroup( Transformation( [ 1, 2, 4, 5, 6, 3, 7, 8 ] ),
> Transformation( [ 3, 3, 4, 5, 6, 2, 7, 8 ] ),
> Transformation( [ 1, 2, 5, 3, 6, 8, 4, 4 ] ) );;
gap> IsInverseSemigroup(S);
true
```

51.5 Ideals of semigroups

Ideals of semigroups are the same as ideals of the semigroup when considered as a magma. For documentation on ideals for magmas, see Magma (35.2.1).

51.5.1 SemigroupIdealByGenerators

▷ SemigroupIdealByGenerators(*S*, *gens*) (operation)

S is a semigroup, *gens* is a list of elements of *S*. Returns the two-sided ideal of *S* generated by *gens*.

51.5.2 ReesCongruenceOfSemigroupIdeal

▷ ReesCongruenceOfSemigroupIdeal(*I*) (attribute)

A two sided ideal *I* of a semigroup *S* defines a congruence on *S* given by $\Delta \cup I \times I$.

51.5.3 IsLeftSemigroupIdeal

▷ IsLeftSemigroupIdeal(*I*) (property)
▷ IsRightSemigroupIdeal(*I*) (property)
▷ IsSemigroupIdeal(*I*) (property)

Categories of semigroup ideals.

51.6 Congruences for semigroups

An equivalence or a congruence on a semigroup is the equivalence or congruence on the semigroup considered as a magma. So, to deal with equivalences and congruences on semigroups, magma functions are used. For documentation on equivalences and congruences for magmas, see Magma (35.2.1).

51.6.1 IsSemigroupCongruence

▷ IsSemigroupCongruence(*c*) (property)

a magma congruence *c* on a semigroup.

51.6.2 IsReesCongruence

▷ IsReesCongruence(*c*) (property)

returns true if and only if the congruence *c* has at most one nonsingleton congruence class.

51.7 Quotients

Given a semigroup and a congruence on the semigroup, one can construct a new semigroup: the quotient semigroup. The following functions deal with quotient semigroups in GAP. For a semigroup S, elements of a quotient semigroup are equivalence classes of elements of the `QuotientSemigroupPreimage` (51.7.3) value under the congruence given by the value of `QuotientSemigroupCongruence` (51.7.3).

It is probably most useful for calculating the elements of the equivalence classes by using `Elements` (30.3.11) or by looking at the images of elements of `QuotientSemigroupPreimage` (51.7.3) under the map returned by `QuotientSemigroupHomomorphism` (51.7.3), which maps the `QuotientSemigroupPreimage` (51.7.3) value to `S`.

For intensive computations in a quotient semigroup, it is probably worthwhile finding another representation as the equality test could involve enumeration of the elements of the congruence classes being compared.

51.7.1 IsQuotientSemigroup

▷ `IsQuotientSemigroup(S)` (Category)

is the category of semigroups constructed from another semigroup and a congruence on it.

51.7.2 HomomorphismQuotientSemigroup

▷ `HomomorphismQuotientSemigroup(cong)` (function)

for a congruence *cong* and a semigroup `S`. Returns the homomorphism from `S` to the quotient of `S` by *cong*.

51.7.3 QuotientSemigroupPreimage

▷ `QuotientSemigroupPreimage(S)` (attribute)
▷ `QuotientSemigroupCongruence(S)` (attribute)
▷ `QuotientSemigroupHomomorphism(S)` (attribute)

for a quotient semigroup `S`.

51.8 Green's Relations

Green's equivalence relations play a very important role in semigroup theory. In this section we describe how they can be used in GAP.

The five Green's relations are R, L, J, H, D: two elements x, y from a semigroup S are R-related if and only if $xS^1 = yS^1$, L-related if and only if $S^1x = S^1y$ and J-related if and only if $S^1xS^1 = S^1yS^1$; finally, $H = R \wedge L$, and $D = R \circ L$.

Recall that relations R, L and J induce a partial order among the elements of the semigroup S: for two elements x, y from S, we say that x is less than or equal to y in the order on R if $xS^1 \subseteq yS^1$; similarly, x is less than or equal to y under L if $S^1x \subseteq S^1y$; finally x is less than or equal to y under J if $S^1xS^1 \subseteq S^1tS^1$. We extend this preorder to a partial order on equivalence classes in the natural way.

51.8.1 GreensRRelation

▷ GreensRRelation(*semigroup*) (attribute)
▷ GreensLRelation(*semigroup*) (attribute)
▷ GreensJRelation(*semigroup*) (attribute)
▷ GreensDRelation(*semigroup*) (attribute)
▷ GreensHRelation(*semigroup*) (attribute)

The Green's relations (which are equivalence relations) are attributes of the semigroup *semigroup*.

51.8.2 IsGreensRelation

▷ IsGreensRelation(*bin-relation*) (property)
▷ IsGreensRRelation(*equiv-relation*) (property)
▷ IsGreensLRelation(*equiv-relation*) (property)
▷ IsGreensJRelation(*equiv-relation*) (property)
▷ IsGreensHRelation(*equiv-relation*) (property)
▷ IsGreensDRelation(*equiv-relation*) (property)

Categories for the Green's relations.

51.8.3 IsGreensClass

▷ IsGreensClass(*equiv-class*) (property)
▷ IsGreensRClass(*equiv-class*) (property)
▷ IsGreensLClass(*equiv-class*) (property)
▷ IsGreensJClass(*equiv-class*) (property)
▷ IsGreensHClass(*equiv-class*) (property)
▷ IsGreensDClass(*equiv-class*) (property)

return `true` if the equivalence class *equiv-class* is a Green's class of any type, or of R, L, J, H, D type, respectively, or `false` otherwise.

51.8.4 IsGreensLessThanOrEqual

▷ IsGreensLessThanOrEqual(*C1, C2*) (operation)

returns `true` if the Green's class *C1* is less than or equal to *C2* under the respective ordering (as defined above), and `false` otherwise.

Only defined for R, L and J classes.

51.8.5 RClassOfHClass

▷ RClassOfHClass(*H*) (attribute)
▷ LClassOfHClass(*H*) (attribute)

are attributes reflecting the natural ordering over the various Green's classes. `RClassOfHClass` and `LClassOfHClass` return the R and L classes, respectively, in which an H class is contained.

51.8.6 EggBoxOfDClass

▷ `EggBoxOfDClass(`*`Dclass`*`)` (attribute)

returns for a Green's D class *`Dclass`* a matrix whose rows represent R classes and columns represent L classes. The entries are the H classes.

51.8.7 DisplayEggBoxOfDClass

▷ `DisplayEggBoxOfDClass(`*`Dclass`*`)` (function)

displays a "picture" of the D class *`Dclass`*, as an array of 1s and 0s. A 1 represents a group H class.

51.8.8 GreensRClassOfElement

▷ `GreensRClassOfElement(`*`S`*`, a)` (operation)
▷ `GreensLClassOfElement(`*`S`*`, a)` (operation)
▷ `GreensDClassOfElement(`*`S`*`, a)` (operation)
▷ `GreensJClassOfElement(`*`S`*`, a)` (operation)
▷ `GreensHClassOfElement(`*`S`*`, a)` (operation)

Creates the X class of the element `a` in the semigroup *`S`* where X is one of L, R, D, J, or H.

51.8.9 GreensRClasses

▷ `GreensRClasses(`*`semigroup`*`)` (attribute)
▷ `GreensLClasses(`*`semigroup`*`)` (attribute)
▷ `GreensJClasses(`*`semigroup`*`)` (attribute)
▷ `GreensDClasses(`*`semigroup`*`)` (attribute)
▷ `GreensHClasses(`*`semigroup`*`)` (attribute)

return the R, L, J, H, or D Green's classes, respectively for semigroup *`semigroup`*. `EquivalenceClasses` (33.7.3) for a Green's relation lead to one of these functions.

51.8.10 GroupHClassOfGreensDClass

▷ `GroupHClassOfGreensDClass(`*`Dclass`*`)` (attribute)

for a D class *`Dclass`* of a semigroup, returns a group H class of the D class, or `fail` if there is no group H class.

51.8.11 IsGroupHClass

▷ `IsGroupHClass(Hclass)` (property)

returns `true` if the Green's H class `Hclass` is a group, which in turn is true if and only if `Hclass`^{2} intersects `Hclass`.

51.8.12 IsRegularDClass

▷ `IsRegularDClass(Dclass)` (property)

returns `true` if the Greens D class `Dclass` is regular. A D class is regular if and only if each of its elements is regular, which in turn is true if and only if any one element of `Dclass` is regular. Idempotents are regular since $eee = e$ so it follows that a Green's D class containing an idempotent is regular. Conversely, it is true that a regular D class must contain at least one idempotent. (See [How76, Prop. 3.2].)

51.9 Rees Matrix Semigroups

In this section, we describe the functions in GAP for Rees matrix and 0-matrix semigroups and their subsemigroups. The importance of these semigroups lies in the fact that Rees matrix semigroups over groups are exactly the completely simple semigroups, and Rees 0-matrix semigroups over groups are the completely 0-simple semigroups.

Let I and J be sets, let S be a semigroup, and let $P = (p_{ji})_{j\in J, i\in I}$ be a $|J| \times |I|$ matrix with entries in S. Then the *Rees matrix semigroup* with underlying semigroup S and matrix P is just the direct product $I \times S \times J$ with multiplication defined by

$$(i,s,j)(k,t,l) = (i, s \cdot p_{j,k} \cdot t, l).$$

Rees 0-matrix semigroups are defined as follows. If I, J, S, and P are as above and 0 denotes a new element, then the *Rees 0-matrix semigroup* with underlying semigroup S and matrix P is $(I \times S \times J) \cup \{0\}$ with multiplication defined by

$$(i,s,j)(k,t,l) = (i, s \cdot p_{j,k} \cdot t, l)$$

when $p_{j,k}$ is not 0 and 0 if $p_{j,k}$ is 0.

If R is a Rees matrix or 0-matrix semigroup, then the *rows* of R is the index set I, the *columns* of R is the index set J, the semigroup S is the *underlying semigroup* of R, and the *matrix* P is the matrix of S.

Thoroughout this section, wherever the distinction is unimportant, we will refer to Rees matrix or 0-matrix semigroups collectively as Rees matrix semigroups.

Multiplication of elements of a Rees matrix semigroup obviously depends on the matrix used to create the semigroup. Hence elements of a Rees matrix semigroup can only be created with reference to the semigroup to which they belong. More specifically, every collection or semigroup of Rees matrix semigroup elements is created from a specific Rees matrix semigroup, which contains the whole family of its elements. So, it is not possible to multiply or compare elements belonging to distinct Rees matrix semigroups, since they belong to different families. This situation is similar to, say, free groups, and different to, say, permutations, which belong to a single family, and where

arbitrary permutations can be compared and multiplied without reference to any group containing them.

A subsemigroup of a Rees matrix semigroup is not necessarily a Rees matrix semigroup. Every semigroup consisting of elements of a Rees matrix semigroup satisfies the property IsReesMatrixSubsemigroup (51.9.6) and every semigroup of Rees 0-matrix semigroup elements satisfies IsReesZeroMatrixSubsemigroup (51.9.6).

Rees matrix and 0-matrix semigroups can be created using the operations ReesMatrixSemigroup (51.9.1) and ReesZeroMatrixSemigroup (51.9.1), respectively, from an underlying semigroup and a matrix. Rees matrix semigroups created in this way contain the whole family of their elements. Every element of a Rees matrix semigroup belongs to a unique semigroup created in this way; every subsemigroup of a Rees matrix semigroup is a subsemigroup of a unique semigroup created in this way.

Subsemigroups of Rees matrix semigroups can also be created by specifying generators. A subsemigroup of a Rees matrix semigroup $I \times U \times J$ satisfies IsReesMatrixSemigroup (51.9.7) if and only if it is equal to $I' \times U' \times J'$ where $I' \subseteq I$, $J' \subseteq J$, and U' is a subsemigroup of U. The analogous statements holds for Rees 0-matrix semigroups.

It is not necessarily the case that a simple subsemigroups of Rees matrix semigroups satisfies IsReesMatrixSemigroup (51.9.7). A Rees matrix semigroup is simple if and only if its underlying semigroup is simple. A finite semigroup is simple if and only if it is isomorphic to a Rees matrix semigroup over a group; this isomorphism can be obtained explicitly using IsomorphismReesMatrixSemigroup (51.9.3).

Similarly, 0-simple subsemigroups of Rees 0-matrix semigroups do not have to satisfy IsReesZeroMatrixSemigroup (51.9.7). A Rees 0-matrix semigroup with more than 2 elements is 0-simple if and only if every row and every column of its matrix contains a non-zero entry, and its underlying semigroup is simple. A finite semigroup is 0-simple if and only if it is isomorphic to a Rees 0-matrix semigroup over a group; again this isomorphism can be found by using IsomorphismReesMatrixSemigroup (51.9.3).

Elements of a Rees matrix or 0-matrix semigroup belong to the categories IsReesMatrixSemigroupElement (51.9.4) and IsReesZeroMatrixSemigroupElement (51.9.4), respectively. Such elements can be created directly using the functions ReesMatrixSemigroupElement (51.9.5) and ReesZeroMatrixSemigroupElement (51.9.5).

A semigroup in GAP can either satisfies IsReesMatrixSemigroup (51.9.7) or IsReesZeroMatrixSemigroup (51.9.7) but not both.

51.9.1 ReesMatrixSemigroup

▷ ReesMatrixSemigroup(*S*, *mat*) (operation)

▷ ReesZeroMatrixSemigroup(*S*, *mat*) (operation)

Returns: A Rees matrix or 0-matrix semigroup.

When *S* is a semigroup and *mat* is an m by n matrix with entries in *S*, the function ReesMatrixSemigroup returns the n by m Rees matrix semigroup over *S* with multiplication defined by *mat*.

The arguments of ReesZeroMatrixSemigroup should be a semigroup *S* and an m by n matrix *mat* with entries in *S* or equal to the integer 0. ReesZeroMatrixSemigroup returns the n by m Rees 0-matrix semigroup over *S* with multiplication defined by *mat*. In GAP a Rees 0-matrix semigroup always contains a multiplicative zero element, regardless of whether there are any entries in *mat* which are equal to 0.

Example

```
gap> G:=Random(AllGroups(Size, 32));;
gap> mat:=List([1..5], x-> List([1..3], y-> Random(G)));;
gap> S:=ReesMatrixSemigroup(G, mat);
<Rees matrix semigroup 3x5 over <pc group of size 32 with
 5 generators>>
gap> mat:=[[(), 0, (), ()], [0, 0, 0, 0]];;
gap> S:=ReesZeroMatrixSemigroup(DihedralGroup(IsPermGroup, 8), mat);
<Rees 0-matrix semigroup 4x2 over Group([ (1,2,3,4), (2,4) ])>
```

51.9.2 ReesMatrixSubsemigroup

▷ ReesMatrixSubsemigroup(*R, I, U, J*) (operation)

▷ ReesZeroMatrixSubsemigroup(*R, I, U, J*) (operation)

Returns: A Rees matrix or 0-matrix subsemigroup.

The arguments of ReesMatrixSubsemigroup should be a Rees matrix semigroup *R*, subsets *I* and *J* of the rows and columns of *R*, respectively, and a subsemigroup *S* of the underlying semigroup of *R*. ReesMatrixSubsemigroup returns the subsemigroup of *R* generated by the direct product of *I*, *U*, and *J*.

The usage and returned value of ReesZeroMatrixSubsemigroup is analogous when *R* is a Rees 0-matrix semigroup.

Example

```
gap> G:=CyclicGroup(IsPermGroup, 1007);;
gap> mat:=[[(), 0, 0], [0, (), 0], [0, 0, ()],
> [(), (), ()], [0, 0, ()]];;
gap> R:=ReesZeroMatrixSemigroup(G, mat);
<Rees 0-matrix semigroup 3x5 over
  <permutation group of size 1007 with 1 generators>>
gap> ReesZeroMatrixSubsemigroup(R, [1,3], G, [1..5]);
<Rees 0-matrix semigroup 2x5 over
  <permutation group of size 1007 with 1 generators>>
```

51.9.3 IsomorphismReesMatrixSemigroup

▷ IsomorphismReesMatrixSemigroup(*S*) (attribute)

Returns: An isomorphism.

Every finite simple semigroup is isomorphic to a Rees matrix semigroup over a group, and every finite 0-simple semigroup is isomorphic to a Rees 0-matrix semigroup over a group.

If the argument *S* is a simple semigroup, then IsomorphismReesMatrixSemigroup returns an isomorphism to a Rees matrix semigroup over a group. If the argument is a 0-simple semigroup, then an isomorphism to a Rees 0-matrix semigroup over a group is returned.

See IsSimpleSemigroup (51.4.4) and IsZeroSimpleSemigroup (51.4.5).

Example

```
gap> S:=Semigroup(Transformation( [ 2, 1, 1, 2, 1 ] ),
> Transformation( [ 3, 4, 3, 4, 4 ] ),
> Transformation( [ 3, 4, 3, 4, 3 ] ),
> Transformation( [ 4, 3, 3, 4, 4 ] ));;
gap> IsSimpleSemigroup(S);
true
```

```
gap> Range(IsomorphismReesMatrixSemigroup(S));
<Rees matrix semigroup 4x2 over Group([ (1,2) ])>
gap> mat:=[[(), 0, 0], [0, (), 0], [0, 0, ()]];;
gap> R:=ReesZeroMatrixSemigroup(Group((1,2,4,5,6)), mat);
<Rees 0-matrix semigroup 3x3 over Group([ (1,2,4,5,6) ])>
gap> U:=ReesZeroMatrixSubsemigroup(R, [1,2], Group(()), [2,3]);
<subsemigroup of 3x3 Rees 0-matrix semigroup with 4 generators>
gap> IsZeroSimpleSemigroup(U);
false
gap> U:=ReesZeroMatrixSubsemigroup(R, [2,3], Group(()), [2,3]);
<subsemigroup of 3x3 Rees 0-matrix semigroup with 3 generators>
gap> IsZeroSimpleSemigroup(U);
true
gap> Rows(U); Columns(U);
[ 2, 3 ]
[ 2, 3 ]
gap> V:=Range(IsomorphismReesMatrixSemigroup(U));
<Rees 0-matrix semigroup 2x2 over Group(())>
gap> Rows(V); Columns(V);
[ 1, 2 ]
[ 1, 2 ]
```

51.9.4 IsReesMatrixSemigroupElement

▷ IsReesMatrixSemigroupElement(*elt*) (Category)
▷ IsReesZeroMatrixSemigroupElement(*elt*) (Category)

Returns: true or false.

Every element of a Rees matrix semigroup belongs to the category IsReesMatrixSemigroupElement, and every element of a Rees 0-matrix semigroup belongs to the category IsReesZeroMatrixSemigroupElement.

Example

```
gap> G:=Group((1,2,3));;
gap> mat:=[ [ (), (1,3,2) ], [ (1,3,2), () ] ];;
gap> R:=ReesMatrixSemigroup(G, mat);
<Rees matrix semigroup 2x2 over Group([ (1,2,3) ])>
gap> GeneratorsOfSemigroup(R);
[ (1,(1,2,3),1), (2,(),2) ]
gap> IsReesMatrixSemigroupElement(last[1]);
true
gap> IsReesZeroMatrixSemigroupElement(last2[1]);
false
```

51.9.5 ReesMatrixSemigroupElement

▷ ReesMatrixSemigroupElement(*R, i, x, j*) (function)
▷ ReesZeroMatrixSemigroupElement(*R, i, x, j*) (function)

Returns: An element of a Rees matrix or 0-matrix semigroup.

The arguments of *ReesMatrixSemigroupElement* should be a Rees matrix subsemigroup *R*, elements *i* and *j* of the the rows and columns of *R*, respectively, and an element *x* of the underlying

semigroup of *R*. ReesMatrixSemigroupElement returns the element of *R* with row index *i*, underlying element *x* in the underlying semigroup of *R*, and column index *j*, if such an element exist, if such an element exists.

The usage of ReesZeroMatrixSemigroupElement is analogous to that of ReesMatrixSemigroupElement, when *R* is a Rees 0-matrix semigroup.

The row *i*, underlying element *x*, and column *j* of an element y of a Rees matrix (or 0-matrix) semigroup can be recovered from y using y[1], y[2], and y[3], respectively.

Example

```
gap> G:=Group((1,2,3));;
gap> mat:=[ [ 0, () ], [ (1,3,2), (1,3,2) ] ];;
gap> R:=ReesZeroMatrixSemigroup(G, mat);
<Rees 0-matrix semigroup 2x2 over Group([ (1,2,3) ])>
gap> ReesZeroMatrixSemigroupElement(R, 1, (1,2,3), 2);
(1,(1,2,3),2)
gap> MultiplicativeZero(R);
0
```

51.9.6 IsReesMatrixSubsemigroup

▷ IsReesMatrixSubsemigroup(*R*) (property)
▷ IsReesZeroMatrixSubsemigroup(*R*) (property)

Returns: true or false.

Every semigroup consisting of elements of a Rees matrix semigroup satisfies the property IsReesMatrixSubsemigroup and every semigroup of Rees 0-matrix semigroup elements satisfies IsReesZeroMatrixSubsemigroup.

Note that a subsemigroup of a Rees matrix semigroup is not necessarily a Rees matrix semigroup.

Example

```
gap> G:=DihedralGroup(32);;
gap> mat:=List([1..2], x-> List([1..10], x-> Random(G)));;
gap> R:=ReesMatrixSemigroup(G, mat);
<Rees matrix semigroup 10x2 over <pc group of size 32 with
 5 generators>>
gap> S:=Semigroup(GeneratorsOfSemigroup(R));
<subsemigroup of 10x2 Rees matrix semigroup with 14 generators>
gap> IsReesMatrixSubsemigroup(S);
true
gap> S:=Semigroup(GeneratorsOfSemigroup(R)[1]);
<subsemigroup of 10x2 Rees matrix semigroup with 1 generator>
gap> IsReesMatrixSubsemigroup(S);
true
```

51.9.7 IsReesMatrixSemigroup

▷ IsReesMatrixSemigroup(*R*) (property)
▷ IsReesZeroMatrixSemigroup(*R*) (property)

Returns: true or false.

A subsemigroup of a Rees matrix semigroup $I \times U \times J$ satisfies IsReesMatrixSemigroup if and only if it is equal to $I' \times U' \times J'$ where $I' \subseteq I$, $J' \subseteq J$, and U' is a subsemigroup of U. It can be costly to

check that a subsemigroup defined by generators satisfies IsReesMatrixSemigroup. The analogous statements holds for Rees 0-matrix semigroups.

It is not necessarily the case that a simple subsemigroups of Rees matrix semigroups satisfies IsReesMatrixSemigroup. A Rees matrix semigroup is simple if and only if its underlying semigroup is simple. A finite semigroup is simple if and only if it is isomorphic to a Rees matrix semigroup over a group; this isomorphism can be obtained explicitly using IsomorphismReesMatrixSemigroup (51.9.3).

Similarly, 0-simple subsemigroups of Rees 0-matrix semigroups do not have to satisfy IsReesZeroMatrixSemigroup. A Rees 0-matrix semigroup with more than 2 elements is 0-simple if and only if every row and every column of its matrix contains a non-zero entry, and its underlying semigroup is simple. A finite semigroup is 0-simple if and only if it is isomorphic to a Rees 0-matrix semigroup over a group; again this isomorphism can be found by using IsomorphismReesMatrixSemigroup (51.9.3).

Example

```
gap> G:=PSL(2,5);;
gap> mat:=[ [ 0, (), 0, (2,6,3,5,4) ],
> [ (), 0, (), 0 ], [ 0, 0, 0, () ] ];;
gap> R:=ReesZeroMatrixSemigroup(G, mat);
<Rees 0-matrix semigroup 4x3 over Group([ (3,5)(4,6), (1,2,5)
(3,4,6) ])>
gap> IsReesZeroMatrixSemigroup(R);
true
gap> U:=ReesZeroMatrixSubsemigroup(R, [1..3], Group(()), [1..2]);
<subsemigroup of 4x3 Rees 0-matrix semigroup with 4 generators>
gap> IsReesZeroMatrixSemigroup(U);
true
gap> V:=Semigroup(GeneratorsOfSemigroup(U));
<subsemigroup of 4x3 Rees 0-matrix semigroup with 4 generators>
gap> IsReesZeroMatrixSemigroup(V);
true
gap> S:=Semigroup(Transformation([1,1]), Transformation([1,2]));
<commutative transformation monoid on 2 pts with 1 generator>
gap> IsSimpleSemigroup(S);
false
gap> mat:=[[0, One(S), 0, One(S)], [One(S), 0, One(S), 0],
> [0, 0, 0, One(S)]];;
gap> R:=ReesZeroMatrixSemigroup(S, mat);;
gap> U:=ReesZeroMatrixSubsemigroup(R, [1..3],
> Semigroup(Transformation([1,1])), [1..2]);
<subsemigroup of 4x3 Rees 0-matrix semigroup with 6 generators>
gap> V:=Semigroup(GeneratorsOfSemigroup(U));
<subsemigroup of 4x3 Rees 0-matrix semigroup with 6 generators>
gap> IsReesZeroMatrixSemigroup(V);
true
gap> T:=Semigroup(
> ReesZeroMatrixSemigroupElement(R, 3, Transformation( [ 1, 1 ] ), 3),
> ReesZeroMatrixSemigroupElement(R, 2, Transformation( [ 1, 1 ] ), 2));
<subsemigroup of 4x3 Rees 0-matrix semigroup with 2 generators>
gap> IsReesZeroMatrixSemigroup(T);
false
```

51.9.8 Matrix

▷ `Matrix(R)` (attribute)

Returns: A matrix.

If `R` is a Rees matrix or 0-matrix semigroup, then `Matrix` returns the matrix used to define multiplication in `R`.

More specifically, if `R` is a Rees matrix or 0-matrix semigroup, which is a proper subsemigroup of another such semigroup, then `Matrix` returns the matrix used to define the Rees matrix (or 0-matrix) semigroup consisting of the whole family to which the elements of `R` belong. Thus, for example, a `1` by `1` Rees matrix semigroup can have a `65` by `15` matrix.

Arbitrary subsemigroups of Rees matrix or 0-matrix semigroups do not have a matrix. Such a subsemigroup `R` has a matrix if and only if it satisfies `IsReesMatrixSemigroup` (51.9.7) or `IsReesZeroMatrixSemigroup` (51.9.7).

Example

```
gap> G:=AlternatingGroup(5);;
gap> mat:=[[(), (), ()], [(), (), ()]];;
gap> R:=ReesMatrixSemigroup(G, mat);
<Rees matrix semigroup 3x2 over Alt( [ 1 .. 5 ] )>
gap> Matrix(R);
[ [ (), (), () ], [ (), (), () ] ]
gap> R:=ReesMatrixSubsemigroup(R, [1,2], Group(()), [2]);
<subsemigroup of 3x2 Rees matrix semigroup with 2 generators>
gap> Matrix(R);
[ [ (), (), () ], [ (), (), () ] ]
```

51.9.9 Rows and columns

▷ `Rows(R)` (attribute)

▷ `Columns(R)` (attribute)

Returns: The rows or columns of `R`.

`Rows` returns the rows of the Rees matrix or 0-matrix semigroup `R`. Note that the rows of the semigroup correspond to the columns of the matrix used to define multiplication in `R`.

`Columns` returns the columns of the Rees matrix or 0-matrix semigroup `R`. Note that the columns of the semigroup correspond to the rows of the matrix used to define multiplication in `R`.

Arbitrary subsemigroups of Rees matrix or 0-matrix semigroups do not have rows or columns. Such a subsemigroup `R` has rows and columns if and only if it satisfies `IsReesMatrixSemigroup` (51.9.7) or `IsReesZeroMatrixSemigroup` (51.9.7).

Example

```
gap> G:=Group((1,2,3));;
gap> mat:=List([1..100], x-> List([1..200], x->Random(G)));;
gap> R:=ReesZeroMatrixSemigroup(G, mat);
<Rees 0-matrix semigroup 200x100 over Group([ (1,2,3) ])>
gap> Rows(R);
[ 1 .. 200 ]
gap> Columns(R);
[ 1 .. 100 ]
```

51.9.10 UnderlyingSemigroup (for a Rees matrix semigroup)

▷ `UnderlyingSemigroup(R)` (attribute)
▷ `UnderlyingSemigroup(R)` (attribute)

Returns: A semigroup.

`UnderlyingSemigroup` returns the underlying semigroup of the Rees matrix or 0-matrix semigroup R.

Arbitrary subsemigroups of Rees matrix or 0-matrix semigroups do not have an underlying semigroup. Such a subsemigroup R has an underlying semigroup if and only if it satisfies `IsReesMatrixSemigroup` (51.9.7) or `IsReesZeroMatrixSemigroup` (51.9.7).

Example

```
gap> S:=Semigroup(Transformation( [ 2, 1, 1, 2, 1 ] ),
> Transformation( [ 3, 4, 3, 4, 4 ] ), Transformation([ 3, 4, 3, 4, 3 ] ),
> Transformation([ 4, 3, 3, 4, 4 ] ) );;
gap> R:=Range(IsomorphismReesMatrixSemigroup(S));
<Rees matrix semigroup 4x2 over Group([ (1,2) ])>
gap> UnderlyingSemigroup(R);
Group([ (1,2) ])
```

51.9.11 AssociatedReesMatrixSemigroupOfDClass

▷ `AssociatedReesMatrixSemigroupOfDClass(D)` (attribute)

Returns: A Rees matrix or 0-matrix semigroup.

If D is a regular $\mathscr{D}$-class of a finite semigroup `S`, then there is a standard way of associating a Rees matrix semigroup to D. If D is a subsemigroup of `S`, then D is simple and hence is isomorphic to a Rees matrix semigroup. In this case, the associated Rees matrix semigroup of D is just the Rees matrix semigroup isomorphic to D.

If D is not a subsemigroup of `S`, then we define a semigroup with elements D and a new element `0` with multiplication of $x, y \in D$ defined by:

$$xy = \begin{cases} x * y \text{ (in } S) & \text{if } x * y \in D \\ 0 & \text{if } xy \notin D. \end{cases}$$

The semigroup thus defined is 0-simple and hence is isomorphic to a Rees 0-matrix semigroup. This semigroup can also be described as the Rees quotient of the ideal generated by D by it maximal subideal. The associated Rees matrix semigroup of D is just the Rees 0-matrix semigroup isomorphic to the semigroup defined above.

Example

```
gap> S:=FullTransformationSemigroup(5);;
gap> D:=GreensDClasses(S)[3];
{Transformation( [ 1, 1, 1, 2, 3 ] )}
gap> AssociatedReesMatrixSemigroupOfDClass(D);
<Rees 0-matrix semigroup 25x10 over Group([ (1,2)(3,5)(4,6), (1,3)
(2,4)(5,6) ])>
```

Chapter 52

Finitely Presented Semigroups and Monoids

A *finitely presented semigroup* (resp. *finitely presented monoid*) is a quotient of a free semigroup (resp. free monoid) on a finite number of generators over a finitely generated congruence on the free semigroup (resp. free monoid).

Finitely presented semigroups are obtained by factoring a free semigroup by a set of relations (a generating set for the congruence), i.e., a set of pairs of words in the free semigroup.

Example

```
gap> f:=FreeSemigroup("a","b");;
gap> x:=GeneratorsOfSemigroup(f);;
gap> s:=f/[ [x[1]*x[2],x[2]*x[1]] ];
<fp semigroup on the generators [ a, b ]>
gap> GeneratorsOfSemigroup(s);
[ a, b ]
gap> RelationsOfFpSemigroup(s);
[ [ a*b, b*a ] ]
```

Finitely presented monoids are obtained by factoring a free monoid by a set of relations, i.e. a set of pairs of words in the free monoid.

Example

```
gap> f:=FreeMonoid("a","b");;
gap> x:=GeneratorsOfMonoid(f);
[ a, b ]
gap> e:=Identity(f);
<identity ...>
gap> m:=f/[ [x[1]*x[2],e] ];
<fp monoid on the generators [ a, b ]>
gap> RelationsOfFpMonoid(m);
[ [ a*b, <identity ...> ] ]
```

Notice that for GAP a finitely presented monoid is not a finitely presented semigroup.

Example

```
gap> IsFpSemigroup(m);
false
```

However, one can build a finitely presented semigroup isomorphic to that finitely presented monoid (see IsomorphismFpSemigroup (52.2.3)).

Also note that is not possible to refer to the generators by their names. These names are not variables, but just display figures. So, if one wants to access the generators by their names, one first has to introduce the respective variables and to assign the generators to them.

Example

```
gap> Unbind(a);
gap> f:=FreeSemigroup("a","b");;
gap> x:=GeneratorsOfSemigroup(f);;
gap> s:=f/[ [x[1]*x[2],x[2]*x[1]] ];;
gap> a;
Error, Variable: 'a' must have a value
gap> a:=GeneratorsOfSemigroup(s)[1];
a
gap> b:=GeneratorsOfSemigroup(s)[2];
b
gap> a in f;
false
gap> a in s;
true
```

The generators of the free semigroup (resp. free monoid) are different from the generators of the finitely presented semigroup (resp. finitely presented monoid) (even though they are displayed by the same names). This means that words in the generators of the free semigroup (resp. free monoid) are not elements of the finitely presented semigroup (resp. finitely presented monoid). Conversely elements of the finitely presented semigroup (resp. finitely presented monoid) are not words of the free semigroup (resp. free monoid).

Calculations comparing elements of an finitely presented semigroup may run into problems: there are finitely presented semigroups for which no algorithm exists (it is known that no such algorithm can exist) that will tell for two arbitrary words in the generators whether the corresponding elements in the finitely presented semigroup are equal. Therefore the methods used by GAP to compute in finitely presented semigroups may run into warning errors, run out of memory or run forever. If the finitely presented semigroup is (by theory) known to be finite the algorithms are guaranteed to terminate (if there is sufficient memory available), but the time needed for the calculation cannot be bounded a priori. The same can be said for monoids. (See 52.6.)

Example

```
gap> a*b=a^5;
false
gap> a^5*b^2*a=a^6*b^2;
true
```

Note that elements of a finitely presented semigroup (or monoid) are not printed in a unique way:

Example

```
gap> a^5*b^2*a;
a^5*b^2*a
gap> a^6*b^2;
a^6*b^2
```

52.1 IsSubsemigroupFpSemigroup (Filter)

52.1.1 IsSubsemigroupFpSemigroup

▷ `IsSubsemigroupFpSemigroup(t)` (attribute)

true if `t` is a finitely presented semigroup or a subsemigroup of a finitely presented semigroup (generally speaking, such a subsemigroup can be constructed with `Semigroup(gens)`, where *gens* is a list of elements of a finitely presented semigroup).

52.1.2 IsSubmonoidFpMonoid

▷ `IsSubmonoidFpMonoid(t)` (attribute)

true if `t` is a finitely presented monoid or a submonoid of a finitely presented monoid (generally speaking, such a semigroup can be constructed with `Monoid(gens)`, where *gens* is a list of elements of a finitely presented monoid).

A submonoid of a monoid has the same identity as the monoid.

52.1.3 IsFpSemigroup

▷ `IsFpSemigroup(s)` (property)

is a synonym for `IsSubsemigroupFpSemigroup(s)` and `IsWholeFamily(s)` (this is because a subsemigroup of a finitely presented semigroup is not necessarily finitely presented).

52.1.4 IsFpMonoid

▷ `IsFpMonoid(m)` (property)

is a synonym for `IsSubmonoidFpMonoid(m)` and `IsWholeFamily(m)` (this is because a submonoid of a finitely presented monoid is not necessarily finitely presented).

52.1.5 IsElementOfFpSemigroup

▷ `IsElementOfFpSemigroup(elm)` (Category)

returns true if *elm* is an element of a finitely presented semigroup.

52.1.6 IsElementOfFpMonoid

▷ `IsElementOfFpMonoid(elm)` (Category)

returns true if *elm* is an element of a finitely presented monoid.

52.1.7 FpGrpMonSmgOfFpGrpMonSmgElement

▷ FpGrpMonSmgOfFpGrpMonSmgElement(*elm*) (operation)

returns the finitely presented group, monoid or semigroup to which *elm* belongs

Example

```
gap> f := FreeSemigroup("a","b");;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> s := f / [ [ a^2 , a*b ] ];;
gap> IsFpSemigroup( s );
true
gap> t := Semigroup( [ GeneratorsOfSemigroup( s )[ 1 ] ]);
<semigroup with 1 generator>
gap> IsSubsemigroupFpSemigroup( t );
true
gap> IsElementOfFpSemigroup( GeneratorsOfSemigroup( t )[ 1 ] );
true
```

52.2 Creating Finitely Presented Semigroups

52.2.1 \/ (for a free semigroup and a list of pairs of elements)

▷ \/(*F*, *rels*) (method)

creates a finitely presented semigroup given by the presentation $\langle gens \mid rels \rangle$ where *gens* are the generators of the free semigroup *F*, and the relations *rels* are entered as pairs of words in the generators of the free semigroup.

The same result is obtained with the infix operator /, i.e., as *F* / *rels*.

Example

```
gap> f:=FreeSemigroup(3);;
gap> s:=GeneratorsOfSemigroup(f);;
gap> f/[ [s[1]*s[2]*s[1],s[1]] , [s[2]^4,s[1]] ];
<fp semigroup on the generators [ s1, s2, s3 ]>
```

52.2.2 FactorFreeSemigroupByRelations

▷ FactorFreeSemigroupByRelations(*f*, *rels*) (function)

for a free semigroup *f* and *rels* is a list of pairs of elements of *f*. Returns the finitely presented semigroup which is the quotient of *f* by the least congruence on *f* generated by the pairs in *rels*.

Example

```
gap> FactorFreeSemigroupByRelations(f,
>                           [[s[1]*s[2]*s[1],s[1]],[s[2]^4,s[1]]]);
<fp semigroup on the generators [ s1, s2, s3 ]>
```

52.2.3 IsomorphismFpSemigroup

▷ `IsomorphismFpSemigroup(s)` (attribute)

for a semigroup `s` returns an isomorphism from `s` to a finitely presented semigroup

Example

```
gap> f := FreeGroup(2);;
gap> g := f/[f.1^4,f.2^5];
<fp group on the generators [ f1, f2 ]>
gap> phi := IsomorphismFpSemigroup(g);
MappingByFunction( <fp group on the generators
[ f1, f2 ]>, <fp semigroup on the generators
[ <identity ...>, f1^-1, f1, f2^-1, f2
 ]>, function( x ) ... end, function( x ) ... end )
gap> s := Range(phi);
<fp semigroup on the generators [ <identity ...>, f1^-1, f1, f2^-1,
  f2 ]>
```

52.3 Comparison of Elements of Finitely Presented Semigroups

52.3.1 \= (for two elements in a f.p. semigroup)

▷ `\=(a, b)` (method)

Two elements `a`, `b` of a finitely presented semigroup are equal if they are equal in the semigroup. Nevertheless they may be represented as different words in the generators. Because of the fundamental problems mentioned in the introduction to this chapter such a test may take a very long time and cannot be guaranteed to finish (see 52.6).

52.4 Preimages in the Free Semigroup

Elements of a finitely presented semigroup are not words, but are represented using a word from the free semigroup as representative.

52.4.1 UnderlyingElement (fp semigroup elements)

▷ `UnderlyingElement(elm)` (operation)

for an element `elm` of a finitely presented semigroup, it returns the word from the free semigroup that is used as a representative for `elm`.

Example

```
gap> f := FreeSemigroup( "a" , "b" );;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> s := f / [ [ a^3 , a ] , [ b^3 , b ] , [ a*b , b*a ] ];
<fp semigroup on the generators [ a, b ]>
gap> w := GeneratorsOfSemigroup(s)[1] * GeneratorsOfSemigroup(s)[2];
a*b
gap> IsWord (w );
```

```
false
gap> ue := UnderlyingElement( w );
a*b
gap> IsWord( ue );
true
```

52.4.2 ElementOfFpSemigroup

▷ ElementOfFpSemigroup(*fam*, *w*) (operation)

for a family *fam* of elements of a finitely presented semigroup and a word *w* in the free generators underlying this finitely presented semigroup, this operation creates the element of the finitely presented semigroup with the representative *w* in the free semigroup.

Example

```
gap> fam := FamilyObj( GeneratorsOfSemigroup(s)[1] );;
gap> ge := ElementOfFpSemigroup( fam, a*b );
a*b
gap> ge in f;
false
gap> ge in s;
true
```

52.4.3 FreeSemigroupOfFpSemigroup

▷ FreeSemigroupOfFpSemigroup(*s*) (attribute)

returns the underlying free semigroup for the finitely presented semigroup *s*, ie, the free semigroup over which *s* is defined as a quotient (this is the free semigroup generated by the free generators provided by FreeGeneratorsOfFpSemigroup(*s*)).

52.4.4 FreeGeneratorsOfFpSemigroup

▷ FreeGeneratorsOfFpSemigroup(*s*) (attribute)

returns the underlying free generators corresponding to the generators of the finitely presented semigroup *s*.

52.4.5 RelationsOfFpSemigroup

▷ RelationsOfFpSemigroup(*s*) (attribute)

returns the relations of the finitely presented semigroup *s* as pairs of words in the free generators provided by FreeGeneratorsOfFpSemigroup(*s*).

Example

```
gap> f := FreeSemigroup( "a" , "b" );;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> s := f / [ [ a^3 , a ] , [ b^3 , b ] , [ a*b , b*a ] ];
<fp semigroup on the generators [ a, b ]>
```

```
gap> Size( s );
8
gap> fs := FreeSemigroupOfFpSemigroup( s );;
gap> f = fs;
true
gap> FreeGeneratorsOfFpSemigroup( s );
[ a, b ]
gap> RelationsOfFpSemigroup( s );
[ [ a^3, a ], [ b^3, b ], [ a*b, b*a ] ]
```

52.5 Finitely presented monoids

The functionality available for finitely presented monoids is essentially the same as that available for finitely presented semigroups, and thus the previous sections apply (with the obvious changes) to finitely presented monoids.

52.5.1 \/ (for a free monoid and a list of pairs of elements)

▷ `\/(F, rels)` (method)

creates a finitely presented monoid given by the monoid presentation $\langle gens \mid rels \rangle$ where *gens* are the generators of the free monoid *F*, and the relations *rels* are entered as pairs of words in both the identity and the generators of the free monoid.

The same result is obtained with the infix operator /, i.e., as *F*`/`*rels*.

Example

```
gap> f := FreeMonoid( 3 );
<free monoid on the generators [ m1, m2, m3 ]>
gap> x := GeneratorsOfMonoid( f );
[ m1, m2, m3 ]
gap> e:= Identity ( f );
<identity ...>
gap> m := f/[ [x[1]^3,e] , [x[1]*x[2],x[2] ]];
<fp monoid on the generators [ m1, m2, m3 ]>
```

52.6 Rewriting Systems and the Knuth-Bendix Procedure

If a finitely presented semigroup has a confluent rewriting system then it has a solvable word problem, that is, there is an algorithm to decide when two words in the free underlying semigroup represent the same element of the finitely presented semigroup. Indeed, once we have a confluent rewriting system, it is possible to successfully test that two words represent the same element in the semigroup, by reducing both words using the rewriting system rules. This is, at the moment, the method that GAP uses to check equality in finitely presented semigroups and monoids.

52.6.1 ReducedConfluentRewritingSystem

▷ `ReducedConfluentRewritingSystem(S[, ordering])` (attribute)

returns a reduced confluent rewriting system of the finitely presented semigroup or monoid *S* with respect to the reduction ordering *ordering* (see 34).

The default for *ordering* is the length plus lexicographic ordering on words, also called the shortlex ordering; for the definition see for example [Sim94].

Notice that this might not terminate. In particular, if the semigroup or monoid *S* does not have a solvable word problem then it this will certainly never end. Also, in this case, the object returned is an immutable rewriting system, because once we have a confluent rewriting system for a finitely presented semigroup or monoid we do not want to allow it to change (as it was most probably very time consuming to get it in the first place). Furthermore, this is also an attribute storing object (see 13.4).

Example

```
gap> f := FreeSemigroup( "a" , "b" );;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> g := f /  [ [ a^2 , a*b ] , [ a^4 , b] ];;
gap> rws := ReducedConfluentRewritingSystem(g);
Rewriting System for Semigroup( [ a, b ] ) with rules
[ [ a*b, a^2 ], [ a^4, b ], [ b*a, a^2 ], [ b^2, a^2 ] ]
```

The creation of a reduced confluent rewriting system for a semigroup or for a monoid, in GAP, uses the Knuth-Bendix procedure for strings, which manipulates a rewriting system of the semigroup or monoid and attempts to make it confluent (See 38. See also Sims [Sim94]). (Since the word problem for semigroups/monoids is not solvable in general, Knuth-Bendix procedure cannot always terminate).

In order to apply this procedure we will build a rewriting system for the semigroup or monoid, which we will call a *Knuth-Bendix Rewriting System* (we need to define this because we need the rewriting system to store some information needed for the implementation of the Knuth-Bendix procedure).

Actually, Knuth-Bendix Rewriting Systems do not only serve this purpose. Indeed these are objects which are mutable and which can be manipulated (see 38).

Note that the implemented version of the Knuth-Bendix procedure, in GAP returns, if it terminates, a confluent rewriting system which is reduced. Also, a reduction ordering has to be specified when building a rewriting system. If none is specified, the shortlex ordering is assumed (note that the procedure may terminate with a certain ordering and not with another one).

On Unix systems it is possible to replace the built-in Knuth-Bendix by other routines, for example the package kbmag offers such a possibility.

52.6.2 KB_REW

▷ KB_REW (global variable)

▷ GAPKB_REW (global variable)

KB_REW is a global record variable whose components contain functions used for Knuth-Bendix. By default KB_REW is assigned to GAPKB_REW, which contains the KB functions provided by the GAP library.

52.6.3 KnuthBendixRewritingSystem

▷ KnuthBendixRewritingSystem(*s*, *wordord*) (function)
▷ KnuthBendixRewritingSystem(*m*, *wordord*) (function)

in the first form, for a semigroup *s* and a reduction ordering for the underlying free semigroup, it returns the Knuth-Bendix rewriting system of the finitely presented semigroup *s* using the reduction ordering *wordord*. In the second form, for a monoid *m* and a reduction ordering for the underlying free monoid, it returns the Knuth-Bendix rewriting system of the finitely presented monoid *m* using the reduction ordering *wordord*.

52.6.4 SemigroupOfRewritingSystem

▷ SemigroupOfRewritingSystem(*rws*) (attribute)

returns the semigroup over which *rws* is a rewriting system

52.6.5 MonoidOfRewritingSystem

▷ MonoidOfRewritingSystem(*rws*) (attribute)

returns the monoid over which *rws* is a rewriting system

52.6.6 FreeSemigroupOfRewritingSystem

▷ FreeSemigroupOfRewritingSystem(*rws*) (attribute)

returns the free semigroup over which *rws* is a rewriting system

52.6.7 FreeMonoidOfRewritingSystem

▷ FreeMonoidOfRewritingSystem(*rws*) (attribute)

returns the free monoid over which *rws* is a rewriting system

Example

```
gap> f1 := FreeSemigroupOfRewritingSystem(rws);
<free semigroup on the generators [ a, b ]>
gap> f1=f;
true
gap> g1 := SemigroupOfRewritingSystem(rws);
<fp semigroup on the generators [ a, b ]>
gap> g1=g;
true
```

As mentioned before, having a confluent rewriting system, one can decide whether two words represent the same element of a finitely presented semigroup (or finitely presented monoid).

Example

```
gap> a := GeneratorsOfSemigroup( g )[ 1 ];
a
```

```
gap> b := GeneratorsOfSemigroup( g )[ 2 ];
b
gap> a*b*a=a^3;
true
gap> ReducedForm(rws,UnderlyingElement(a*b*a));
a^3
gap> ReducedForm(rws,UnderlyingElement(a^3));
a^3
```

52.7 Todd-Coxeter Procedure

This procedure gives a standard way of finding a transformation representation of a finitely presented semigroup. Usually one does not explicitly call this procedure but uses `IsomorphismTransformationSemigroup` (53.7.5).

52.7.1 CosetTableOfFpSemigroup

▷ `CosetTableOfFpSemigroup(r)` (attribute)

`r` is a right congruence of an fp-semigroup `S`. This attribute is the coset table of FP semigroup `S` on a right congruence `r`. Given a right congruence `r` we represent `S` as a set of transformations of the congruence classes of `r`.

The images of the cosets under the generators are compiled in a list `table` such that `table[i][s]` contains the image of coset `s` under generator `i`.

Chapter 53

Transformations

This chapter describes the functions in GAP for transformations.

A *transformation* in GAP is simply a function from the positive integers to the positive integers. Transformations are to semigroup theory what permutations are to group theory, in the sense that every semigroup can be realised as a semigroup of transformations. In GAP transformation semigroups are always finite, and so only finite semigroups can be realised in this way.

A transformation in GAP acts on the positive integers (up to some architecture dependent limit) on the right. The image of a point `i` under a transformation `f` is expressed as `i^f` in GAP. This action is also implemented by the function `OnPoints` (41.2.1). If `i^f` is different from `i`, then `i` is *moved* by *f* and otherwise it is *fixed* by `f`. Transformations in GAP are created using the operations described in Section 53.2.

The *degree* of a transformation `f` is usually defined as the largest positive integer where `f` is defined. In previous versions of GAP, transformations were only defined on positive integers less than their degree, it was only possible to multiply transformations of equal degree, and a transformation did not act on any point exceeding its degree. Starting with version 4.7 of GAP, transformations behave more like permutations, in that they fix unspecified points and it is possible to multiply arbitrary transformations; see Chapter 42. The definition of the degree of a transformation `f` in the current version of GAP is the largest value `n` such that `n^f<>n` or `i^f=n` for some `i<>n`. Equivalently, the degree of a transformation is the least value `n` such that `[n+1,n+2,...]` is fixed pointwise by `f`.

The transformations of a given degree belong to the full transformation semigroup of that degree; see `FullTransformationSemigroup` (53.7.3). Transformation semigroups are hence subsemigroups of the full transformation semigroup.

It is possible to use transformations in GAP without reference to the degree, much as it is possible to use permutations in this way. However, for backwards compatibility, and because it is sometimes useful, it is possible to access the degree of a transformation using `DegreeOfTransformation` (53.5.1). Certain attributes of transformations are also calculated with respect to the degree, such as the rank, image set, or kernel (these values can also be calculated with respect to any positive integer). So, it is possible to ignore the degree of a transformation if you prefer to think of transformations as acting on the positive integers in a similar way to permutations. For example, this approach is used in the FR package. It is also possible to think of transformations as only acting on the positive integers not exceeding their degree. For example, this was the approach formerly used in GAP and it is also useful in the Semigroups package.

Transformations are displayed, by default, using the list `[1^f..n^f]` where `n` is the degree of `f`. This behaviour differs from versions of GAP earlier than 4.7. See Section 53.6 for more information.

The *rank* of a transformation on the positive integers up to `n` is the number of distinct points in `[1^f..n^f]`. The *kernel* of a transformation `f` on `[1..n]` is the equivalence relation on `[1..n]` consisting of those `(i, j)` such that `i^f = j^f`. The kernel of a transformation is represented in two ways: as a partition of `[1..n]` or as the image list of a transformation `g` such that the kernel of `g` on `[1..n]` equals the kernel of `f` and `j^g=i` for all `j` in `i`th class. The latter is referred to as the flat kernel of `f`. For any given transformation and value `n`, there is a unique transformation with this property.

A *functional digraph* is a directed graph where every vertex has out-degree 1. A transformation `f` can be thought of as a functional digraph with vertices the positive integers and edges from `i` to `i^f` for every `i`. A *component* of a transformation is defined as a component and a *cycle* is just a cycle (or strongly connected component) of the corresponding functional digraph. More specifically, `i` and `j` are in the same component if and only if there are $i = v_0, v_1, \ldots, v_n = j$ such that either $v_{k+1} = v_k^f$ or $v_k = v_{k+1}^f$ for all k. A *cycle* of a transformation is defined as a cycle (or strongly connected component) of the corresponding functional digraph. More specifically, `i` belongs to a cycle of `f` if there are $i = v_0, v_1, \ldots, v_n = i$ such that either $v_{k+1} = v_k^f$ or $v_k = v_{k+1}^f$ for all k.

Internally, GAP stores a transformation `f` as a list consisting of the images `i^f` of the points in `i` less than some value, which is at least the degree of `f` and which is determined at the time of creation. When the degree of a transformation `f` is at most 65536, the images of points under `f` are stored as 16-bit integers, the kernel and image set are subobjects of `f` which are plain lists of GAP integers. When the degree of `f` is greater than 65536, the images of points under `f` are stored as 32-bit integers; the kernel and image set are stored in the same way as before. A transformation belongs to `IsTrans2Rep` if it is stored using 16-bit integers and to `IsTrans4Rep` if it is stored using 32-bit integers.

53.1 The family and categories of transformations

53.1.1 IsTransformation

▷ `IsTransformation(`*obj*`)` (Category)

Every transformation in GAP belongs to the category `IsTransformation`. Basic operations for transformations are `ImageListOfTransformation` (53.5.2), `ImageSetOfTransformation` (53.5.3), `KernelOfTransformation` (53.5.12), `FlatKernelOfTransformation` (53.5.11), `RankOfTransformation` (53.5.4), `DegreeOfTransformation` (53.5.1), multiplication of two transformations via `*`, and exponentiation with the first argument a positive integer `i` and second argument a transformation `f` where the result is the image `i^f` of the point `i` under `f`.

53.1.2 IsTransformationCollection

▷ `IsTransformationCollection(`*obj*`)` (Category)

Every collection of transformations belongs to the category `IsTransformationCollection`. For example, transformation semigroups belong to `IsTransformationCollection`.

53.1.3 TransformationFamily

▷ `TransformationFamily` (family)

The family of all transformations is TransformationFamily.

53.2 Creating transformations

There are several ways of creating transformations in GAP, which are described in this section.

53.2.1 Transformation (for an image list)

▷ Transformation(*list*) (operation)
▷ Transformation(*list, func*) (operation)
▷ TransformationList(*list*) (function)

Returns: A transformation or fail.

TransformationList returns the transformation f such that i^*f*=*list*[i] if i is between 1 and the length of *list* and i^*f*=i if i is larger than the length of *list*. TransformationList will return fail if *list* is not dense, if *list* contains an element which is not a positive integer, or if *list* contains an integer not in [1..Length(*list*)].

This is the analogue in the context of transformations of PermList (42.5.2). Transformation is a synonym of TransformationList when the argument is a list.

When the arguments are a list of positive integers *list* and a function *func*, Transformation returns the transformation f such that *list*[i]^f=*func*(*list*[i]) if i is in the range [1..Length(*list*)] and f fixes all other points.

Example

```
gap> SetUserPreference("NotationForTransformations", "input");
gap> f:=Transformation( [ 11, 10, 2, 11, 4, 4, 7, 6, 9, 10, 1, 11 ] );
Transformation( [ 11, 10, 2, 11, 4, 4, 7, 6, 9, 10, 1, 11 ] )
gap> f:=TransformationList( [ 2, 3, 3, 1 ] );
Transformation( [ 2, 3, 3, 1 ] )
gap> SetUserPreference("NotationForTransformations", "fr");
gap> f:=Transformation([10, 11], x-> x^2);
<transformation: 1,2,3,4,5,6,7,8,9,100,121>
gap> SetUserPreference("NotationForTransformations", "input");
```

53.2.2 Transformation (for a source and destination)

▷ Transformation(*src, dst*) (operation)
▷ TransformationListList(*src, dst*) (function)

Returns: A transformation or fail.

If *src* and *dst* are lists of positive integers of the same length, such that *src* contains no element twice, then TransformationListList(*src*, *dst*) returns a transformation f such that src[i]^*f*=dst[i]. The transformation *f* fixes all points larger than the maximum of the entries in *src* and *dst*.

This is the analogue in the context of transformations of MappingPermListList (42.5.3). Transformation is a synonym of TransformationListList when its arguments are two lists of positive integers.

Example

```
gap> Transformation( [ 10, 11 ],[ 11, 12 ] );
Transformation( [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 12 ] )
gap> TransformationListList( [ 1, 2, 3 ], [ 4, 5, 6 ] );
Transformation( [ 4, 5, 6, 4, 5, 6 ] )
```

53.2.3 TransformationByImageAndKernel (for an image and kernel)

▷ TransformationByImageAndKernel(*im, ker*) (operation)

Returns: A transformation or fail.

Transformation returns the transformation f i^f=*im*[*ker*[i]] for i in the range [1..Length(*ker*)]. This transformation has flat kernel equal to *ker* and image set equal to Set(*im*).

The argument *im* should be a duplicate free list of positive integers and *ker* should be the flat kernel of a transformation with rank equal to the length of *im*. If the arguments do not fulfil these conditions, then fail is returned.

Example

```
gap> TransformationByImageAndKernel([ 8, 1, 3, 4 ],
> [ 1, 2, 3, 1, 2, 1, 2, 4 ]);
Transformation( [ 8, 1, 3, 8, 1, 8, 1, 4 ] )
gap> TransformationByImageAndKernel([ 1, 3, 8, 4 ],
> [ 1, 2, 3, 1, 2, 1, 2, 4 ]);
Transformation( [ 1, 3, 8, 1, 3, 1, 3, 4 ] )
```

53.2.4 Idempotent

▷ Idempotent(*im, ker*) (function)

Returns: A transformation or fail.

Idempotent returns the idempotent transformation with image set *im* and flat kernel *ker* if such a transformation exists and fail if it does not.

More specifically, a transformation is returned when the argument *im* is a set of positive integers and *ker* is the flat kernel of a transformation with rank equal to the length of *im* and where *im* has one element in every class of the kernel corresponding to *ker*.

Note that this is function does not always return the same transformation as TransformationByImageAndKernel with the same arguments.

Example

```
gap> Idempotent([ 2, 4, 6, 7, 8, 10, 11 ],
> [ 1, 2, 1, 3, 3, 4, 5, 1, 6, 6, 7, 5 ] );
Transformation( [ 8, 2, 8, 4, 4, 6, 7, 8, 10, 10, 11, 7 ] )
gap> TransformationByImageAndKernel([ 2, 4, 6, 7, 8, 10, 11 ],
> [ 1, 2, 1, 3, 3, 4, 5, 1, 6, 6, 7, 5 ] );
Transformation( [ 2, 4, 2, 6, 6, 7, 8, 2, 10, 10, 11, 8 ] )
```

53.2.5 TransformationOp

▷ TransformationOp(*obj, list[, func]*) (function)

▷ TransformationOpNC(*obj, list[, func]*) (function)

Returns: A transformation or fail.

TransformationOp returns the transformation that corresponds to the action of the object *obj* on the domain or list *list* via the function *func*. If the optional third argument *func* is not specified, then the action OnPoints (41.2.1) is used by default. Note that the returned transformation refers to the positions in *list* even if *list* itself consists of integers.

This function is the analogue in the context of transformations of Permutation (**Reference: Permutation**).

If *obj* does not map elements of *list* into *list*, then fail is returned.

TransformationOpNC does not check that *obj* maps elements of *list* to elements of *list* or that a transformation is defined by the action of *obj* on *list* via *func*. This function should be used only with caution, and in situations where it is guaranteed that the arguments have the required properties.

Example

```
gap> f:=Transformation( [ 10, 2, 3, 10, 5, 10, 7, 2, 5, 6 ] );;
gap> TransformationOp(f, [ 2, 3 ] );
IdentityTransformation
gap> TransformationOp(f, [ 1, 2, 3 ] );
fail
gap> S:=SemigroupByMultiplicationTable( [ [ 1, 1, 1 ], [ 1, 1, 1 ],
> [ 1, 1, 2 ] ] );;
gap> TransformationOp(Elements(S)[1], S, OnRight);
Transformation( [ 1, 1, 1 ] )
gap> TransformationOp(Elements(S)[3], S, OnRight);
Transformation( [ 1, 1, 2 ] )
```

53.2.6 TransformationNumber

▷ TransformationNumber(*m, n*) (operation)

▷ NumberTransformation(*f[, n]*) (operation)

Returns: A transformation or a number.

These functions implement a bijection from the transformations with degree at most *n* to the numbers [1..*n*^*n*].

More precisely, if *m* and *n* are positive integers such that *m* is at most *n*^*n*, then TransformationNumber returns the *m*th transformation with degree at most *n*.

If *f* is a transformation and *n* is a positive integer, which is greater than or equal to the degree of *f*, then NumberTransformation returns the number in [1..*n*^*n*] that corresponds to *f*. If the optional second argument *n* is not specified, then the degree of *f* is used by default.

Example

```
gap> f:=Transformation( [ 3, 3, 5, 3, 3 ] );;
gap> NumberTransformation(f, 5);
1613
gap> NumberTransformation(f, 10);
2242256790
gap> TransformationNumber(2242256790, 10);
Transformation( [ 3, 3, 5, 3, 3 ] )
gap> TransformationNumber(1613, 5);
Transformation( [ 3, 3, 5, 3, 3 ] )
```

53.2.7 RandomTransformation

▷ RandomTransformation(*n*) (operation)

Returns: A random transformation.

If *n* is a positive integer, then RandomTransformation returns a random transformation with degree at most *n*.

Example

```
gap> RandomTransformation(6);
Transformation( [ 2, 1, 2, 1, 1, 2 ] )
```

53.2.8 IdentityTransformation

▷ IdentityTransformation (global variable)

Returns: The identity transformation.

Returns the identity transformation, which has degree 0.

Example

```
gap> f:=IdentityTransformation;
IdentityTransformation
```

53.2.9 ConstantTransformation

▷ ConstantTransformation(*m*, *n*) (operation)

Returns: A transformation.

This function returns a constant transformation f such that i^f=*n* for all i less than or equal to *m*, when *n* and *m* are positive integers.

Example

```
gap> ConstantTransformation(5, 1);
Transformation( [ 1, 1, 1, 1, 1 ] )
gap> ConstantTransformation(6, 4);
Transformation( [ 4, 4, 4, 4, 4, 4 ] )
```

53.3 Changing the representation of a transformation

It is possible that a transformation in GAP can be represented as another type of object, or that another type of GAP object can be represented as a transformation.

The operations AsPermutation (42.5.5) and AsPartialPerm (54.4.2) can be used to convert transformations into permutations or partial permutations, where appropriate. In this section we describe functions for converting other types of objects into transformations.

53.3.1 AsTransformation

▷ AsTransformation(*f*[, *n*]) (operation)

Returns: A transformation.

AsTransformation returns the permutation, transformation, partial permutation or binary relation *f* as a transformation.

for permutations

If *f* is a permutation and *n* is a non-negative integer, then AsTransformation(*f*, *n*) returns the transformation g such that i^g=i^f for all i in the range [1..*n*].

If no non-negative integer *n* is specified, then the largest moved point of *f* is used as the value for *n*; see LargestMovedPoint (42.3.2).

for transformations

If *f* is a transformation and *n* is a non-negative integer less than the degree of *f* such that *f* is a transformation of [1..*n*], then AsTransformation returns the restriction of *f* to [1..*n*].

If *f* is a transformation and *n* is not specified or equals a is greater than or equal to the degree of *f*, then *f* is returned.

for partial permutations

A partial permutation *f* can be converted into a transformation g as follows. The degree m of g is equal to the maximum of *n*, the largest moved point of *f* plus 1, and the largest image of a moved point plus 1. The transformation g agrees with *f* on the domain of *f* and maps the points in [1..m], which are not in the domain of *f* to n, i.e. i^g=i^*f* for all i in the domain of *f*, i^g=n for all i in [1..n], and i^g=i for all i greater than *n*. AsTransformation(*f*) returns the transformation g defined in the previous sentences.

If the optional argument *n* is not present, then the default value of the maximum of the largest moved point and the largest image of a moved point of *f* plus 1 is used.

for binary relations

In the case that *f* is a binary relation, which defines a transformation, then AsTransformation returns that transformation.

Example

```
gap> f:=Transformation( [ 3, 5, 3, 4, 1, 2 ] );;
gap> AsTransformation(f, 5);
Transformation( [ 3, 5, 3, 4, 1 ] )
gap> AsTransformation(f, 10);
Transformation( [ 3, 5, 3, 4, 1, 2 ] )
gap> AsTransformation((1, 3)(2, 4));
Transformation( [ 3, 4, 1, 2 ] )
gap> AsTransformation((1, 3)(2, 4), 10);
Transformation( [ 3, 4, 1, 2 ] )
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 6 ], [ 6, 7, 1, 4, 3, 2 ] );
[5,3,1,6,2,7](4)
gap> AsTransformation(f, 11);
Transformation( [ 6, 7, 1, 4, 3, 2, 11, 11, 11, 11, 11 ] )
gap> AsPartialPerm(last, DomainOfPartialPerm(f));
[5,3,1,6,2,7](4)
gap> AsTransformation(f, 14);
Transformation( [ 6, 7, 1, 4, 3, 2, 14, 14, 14, 14, 14, 14, 14, 14 ] )
gap> AsPartialPerm(last, DomainOfPartialPerm(f));
[5,3,1,6,2,7](4)
gap> AsTransformation(f);
Transformation( [ 6, 7, 1, 4, 3, 2, 8, 8 ] )
gap> AsTransformation(Transformation( [ 1, 1, 2 ] ), 0);
IdentityTransformation
```

53.3.2 RestrictedTransformation

▷ RestrictedTransformation(*f*, *list*) (operation)

▷ RestrictedTransformationNC(*f*, *list*) (operation)

Returns: A transformation.

RestrictedTransformation returns the new transformation g such that i^g=i^*f* for all i in *list* and such that i^g=i for all i not in *list*.

RestrictedTransformation checks that *list* is a duplicate free dense list consisting of positive integers, whereas RestrictedTransformationNC performs no checks.

Example

```
gap> f:=Transformation( [ 2, 10, 5, 9, 10, 9, 6, 3, 8, 4, 6, 5 ] );;
gap> RestrictedTransformation(f, [ 1, 2, 3, 10, 11, 12 ] );
```

```
Transformation( [ 2, 10, 5, 4, 5, 6, 7, 8, 9, 4, 6, 5 ] )
```

53.3.3 PermutationOfImage

▷ `PermutationOfImage(f)` (attribute)

Returns: A permutation or `fail`.

If the transformation `f` is a permutation of the points in its image, then `PermutationOfImage` returns this permutation. If `f` does not permute its image, then `fail` is returned.

If `f` happens to be a permutation, then `PermutationOfImage` with argument `f` returns the same value as `AsPermutation` with argument `f`.

Example

```
gap> f:=Transformation( [ 5, 8, 3, 5, 8, 6, 2, 2, 7, 8 ] );;
gap> PermutationOfImage(f);
fail
gap> f:=Transformation( [ 8, 2, 10, 2, 4, 4, 7, 6, 9, 10 ] );;
gap> PermutationOfImage(f);
fail
gap> f:=Transformation( [ 1, 3, 6, 6, 2, 10, 2, 3, 10, 5 ] );;
gap> PermutationOfImage(f);
(2,3,6,10,5)
gap> f:=Transformation( [ 5, 2, 8, 4, 1, 8, 10, 3, 5, 7 ] );;
gap> PermutationOfImage(f);
(1,5)(3,8)(7,10)
```

53.4 Operators for transformations

`i ^ f`

returns the image of the positive integer `i` under the transformation `f`.

`f ^ g`

returns `g^-1*f*g` when `f` is a transformation and `g` is a permutation `\^` (**Reference:** `^`). This operation requires essentially the same number of steps as multiplying a transformation by a permutation, which is approximately one third of the number required to first invert `g`, take the produce with `f`, and then the product with `g`.

`f * g`

returns the composition of `f` and `g` when `f` and `g` are transformations or permutations. The product of a permutation and a transformation is returned as a transformation.

`f / g`

returns `f*g^-1` when `f` is a transformation and `g` is a permutation. This operation requires essentially the same number of steps as multiplying a transformation by a permutation, which is approximately half the number required to first invert `g` and then take the produce with `f`.

`LQUO(g, f)`

returns `g^-1*f` when `f` is a transformation and `g` is a permutation. This operation uses essentially the same number of steps as multiplying a transformation by a permutation, which is approximately half the number required to first invert `g` and then take the produce with `f`.

`f < g`
: returns `true` if the image list of `f` is lexicographically less than the image list of `g` and `false` if it is not.

`f = g`
: returns `true` if the transformation `f` equals the transformation `g` and returns `false` if it does not.

53.4.1 PermLeftQuoTransformation

▷ `PermLeftQuoTransformation(f, g)` (operation)
▷ `PermLeftQuoTransformationNC(f, g)` (operation)

Returns: A permutation.

Returns the permutation on the image set of `f` induced by `f^-1*g` when the transformations `f` and `g` have equal kernel and image set.

`PermLeftQuoTransformation` verifies that `f` and `g` have equal kernels and image sets, and returns an error if they do not. `PermLeftQuoTransformationNC` does no checks.

Example

```
gap> f:=Transformation( [ 5, 6, 7, 1, 4, 3, 2, 7 ] );;
gap> g:=Transformation( [ 5, 7, 1, 6, 4, 3, 2, 1 ] );;
gap> PermLeftQuoTransformation(f, g);
(1,6,7)
gap> PermLeftQuoTransformation(g, f);
(1,7,6)
```

53.4.2 IsInjectiveListTrans

▷ `IsInjectiveListTrans(obj, list)` (operation)

Returns: `true` or `false`.

The argument *obj* should be a transformation or the list of images of a transformation and *list* should be a list of positive integers. `IsInjectiveListTrans` checks if *obj* is injective on *list*.

More precisely, if *obj* is a transformation, then we define `f:=`*obj* and if *obj* is the image list of a transformation we define `f:=Transformation(`*obj*`)`. `IsInjectiveListTrans` returns `true` if `f` is injective on *list* and `false` if it is not. If *list* is not duplicate free, then `false` is returned.

Example

```
gap> f:=Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 1, 5 ] );;
gap> IsInjectiveListTrans( [ 1, 5 ], f );
true
gap> IsInjectiveListTrans( [ 5, 1 ], f );
true
gap> IsInjectiveListTrans( [ 5, 1, 5, 1, 1, ], f );
false
gap> IsInjectiveListTrans( [ 5, 1, 2, 3 ], [ 1, 2, 3, 4, 5 ] );
true
```

53.4.3 ComponentTransformationInt

▷ `ComponentTransformationInt(f, n)` (attribute)

Returns: A list of positive integers.

If *f* is a transformation and *n* is a positive integer, then ComponentTransformationInt returns those elements i such that *n*^*f*^j=i for some positive integer j, i.e. the elements of the component of *f* containing *n* that can be obtained by applying powers of *f* to *n*.

Example

```
gap> f:=Transformation( [ 6, 2, 8, 4, 7, 5, 8, 3, 5, 8 ] );;
gap> ComponentTransformationInt(f, 1);
[ 1, 6, 5, 7, 8, 3 ]
gap> ComponentTransformationInt(f, 12);
[ 12 ]
gap> ComponentTransformationInt(f, 5);
[ 5, 7, 8, 3 ]
```

53.4.4 PreImagesOfTransformation

▷ PreImagesOfTransformation(*f*, *n*) (operation)

Returns: A set of positive integers.

Returns the preimages of the positive integer *n* under the transformation *f*, i.e. the positive integers i such that i^*f*=n.

Example

```
gap> f:=Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 1, 5 ] );;
gap> PreImagesOfTransformation(f, 1);
[ 8, 9 ]
gap> PreImagesOfTransformation(f, 3);
[  ]
gap> PreImagesOfTransformation(f, 100);
[ 100 ]
```

53.5 Attributes for transformations

In this section we describe the functions available in GAP for finding various properties and attributes of transformations.

53.5.1 DegreeOfTransformation

▷ DegreeOfTransformation(*f*) (attribute)

▷ DegreeOfTransformationCollection(*coll*) (attribute)

Returns: A positive integer.

The *degree* of a transformation *f* is the largest value such that n^*f*<>n or i^*f*=n for some i<>n. Equivalently, the degree of a transformation is the least value n such that [n+1,n+2,...] is fixed pointwise by *f*. The degree a collection of transformations *coll* is the maximum degree of any transformation in *coll*.

Example

```
gap> DegreeOfTransformation(IdentityTransformation);
0
gap> DegreeOfTransformationCollection([ Transformation( [ 1, 3, 4, 1 ] ),
> Transformation( [ 3, 1, 1, 3, 4 ]), Transformation( [ 2, 4, 1, 2 ] ) ]);
5
```

53.5.2 ImageListOfTransformation

▷ ImageListOfTransformation(*f[, n]*) (operation)
▷ ListTransformation(*f[, n]*) (operation)

Returns: The list of images of a transformation.

Returns the list of images of `[1..n]` under the transformation *f*, which is `[1^f..n^f]`. If the optional second argument *n* is not present, then the degree of *f* is used by default.

This is the analogue for transformations of `ListPerm` (42.5.1) for permutations.

Example

```
gap> f:=Transformation( [ 2 ,3, 4, 2, 4 ] );;
gap> ImageListOfTransformation(f);
[ 2, 3, 4, 2, 4 ]
gap> ImageListOfTransformation(f, 10);
[ 2, 3, 4, 2, 4, 6, 7, 8, 9, 10 ]
```

53.5.3 ImageSetOfTransformation

▷ ImageSetOfTransformation(*f[, n]*) (attribute)

Returns: The set of images of the transformation.

Returns the set of points in the list of images of `[1..n]` under *f*, i.e. the sorted list of images with duplicates removed. If the optional second argument *n* is not given, then the degree of *f* is used.

Example

```
gap> f:=Transformation( [ 5, 6, 7, 1, 4, 3, 2, 7 ] );;
gap> ImageSetOfTransformation(f);
[ 1, 2, 3, 4, 5, 6, 7 ]
gap> ImageSetOfTransformation(f, 10);
[ 1, 2, 3, 4, 5, 6, 7, 9, 10 ]
```

53.5.4 RankOfTransformation (for a transformation and a positive integer)

▷ RankOfTransformation(*f[, n]*) (attribute)
▷ RankOfTransformation(*f[, list]*) (attribute)

Returns: The rank of a transformation.

When the arguments are a transformation *f* and a positive integer *n*, `RankOfTransformation` returns the size of the set of images of the transformation *f* in the range `[1..n]`. If the optional second argument *n* is not specified, then the degree of *f* is used.

When the arguments are a transformation *f* and a list *list* of positive integers, this function returns the size of the set of images of the transformation *f* on *list*.

Example

```
gap> f:=Transformation( [ 8, 5, 8, 2, 2, 8, 4, 7, 3, 1 ] );;
gap> ImageSetOfTransformation(f);
[ 1, 2, 3, 4, 5, 7, 8 ]
gap> RankOfTransformation(f);
7
gap> RankOfTransformation(f, 100);
97
gap> RankOfTransformation(f, [ 2, 5, 8 ] );
3
```

53.5.5 MovedPoints (for a transformation)

▷ MovedPoints(*f*) (operation)
▷ MovedPoints(*coll*) (operation)

Returns: A set of positive integers.

When the argument is a transformation, MovedPoints returns the set of positive integers i such that i^*f*<>i. MovedPoints returns the set of points moved by some element of the collection of transformations *coll*.

Example

```
gap> f:=Transformation( [ 6, 10, 1, 4, 6, 5, 1, 2, 3, 3 ] );;
gap> MovedPoints(f);
[ 1, 2, 3, 5, 6, 7, 8, 9, 10 ]
gap> f:=IdentityTransformation;
IdentityTransformation
gap> MovedPoints(f);
[  ]
```

53.5.6 NrMovedPoints (for a transformation)

▷ NrMovedPoints(*f*) (operation)
▷ NrMovedPoints(*coll*) (operation)

Returns: A positive integer.

When the argument is a transformation,NrMovedPoints returns the number of positive integers i such that i^*f*<>i. MovedPoints returns the number of points which are moved by at least one element of the collection of transformations *coll*.

Example

```
gap> f:=Transformation( [ 7, 1, 4, 3, 2, 7, 7, 6, 6, 5 ] );;
gap> NrMovedPoints(f);
9
gap> NrMovedPoints(IdentityTransformation);
0
```

53.5.7 SmallestMovedPoint (for a transformation)

▷ SmallestMovedPoint(*f*) (operation)
▷ SmallestMovedPoint(*coll*) (operation)

Returns: A positive integer or infinity.

SmallestMovedPoint returns the smallest positive integer i such that i^*f*<>i if such an i exists. If *f* is the identity transformation, then infinity is returned.

If the argument is a collection of transformations *coll*, then the smallest point which is moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity transformations, then SmallestMovedPoint returns infinity.

Example

```
gap> S:=FullTransformationSemigroup(5);
<full transformation semigroup on 5 pts>
gap> SmallestMovedPoint(S);
1
gap> S:=Semigroup(IdentityTransformation);;
gap> SmallestMovedPoint(S);
infinity
```

```
gap> f:=Transformation( [ 1, 2, 3, 6, 6, 6 ] );;
gap> SmallestMovedPoint(f);
4
```

53.5.8 LargestMovedPoint (for a transformation)

▷ `LargestMovedPoint(`*f*`)` (operation)
▷ `LargestMovedPoint(`*coll*`)` (operation)

Returns: A positive integer.

`LargestMovedPoint` returns the largest positive integers `i` such that `i^`*f*`<>i` if such an `i` exists. If *f* is the identity transformation, then `0` is returned.

If the argument is a collection of transformations *coll*, then the largest point which is moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity transformations, then `LargestMovedPoint` returns `0`.

Example

```
gap> S:=FullTransformationSemigroup(5);
<full transformation semigroup on 5 pts>
gap> LargestMovedPoint(S);
5
gap> S:=Semigroup(IdentityTransformation);;
gap> LargestMovedPoint(S);
0
gap> f:=Transformation( [ 1, 2, 3, 6, 6, 6 ] );;
gap> LargestMovedPoint(f);
5
```

53.5.9 SmallestImageOfMovedPoint (for a transformation)

▷ `SmallestImageOfMovedPoint(`*f*`)` (operation)
▷ `SmallestImageOfMovedPoint(`*coll*`)` (operation)

Returns: A positive integer or `infinity`.

`SmallestImageOfMovedPoint` returns the smallest positive integer `i^`*f* such that `i^`*f*`<>i` if such an `i` exists. If *f* is the identity transformation, then `infinity` is returned.

If the argument is a collection of transformations *coll*, then the smallest integer which is the image a point moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity transformations, then `SmallestImageOfMovedPoint` returns `infinity`.

Example

```
gap> S:=FullTransformationSemigroup(5);
<full transformation semigroup on 5 pts>
gap> SmallestImageOfMovedPoint(S);
1
gap> S:=Semigroup(IdentityTransformation);;
gap> SmallestImageOfMovedPoint(S);
infinity
gap> f:=Transformation( [ 1, 2, 3, 6, 6, 6 ] );;
gap> SmallestImageOfMovedPoint(f);
6
```

53.5.10 LargestImageOfMovedPoint (for a transformation)

▷ `LargestImageOfMovedPoint(`*f*`)` (operation)
▷ `LargestImageOfMovedPoint(`*coll*`)` (operation)

Returns: A positive integer.

`LargestImageOfMovedPoint` returns the largest positive integer `i^f` such that `i^f<>i` if such an `i` exists. If *f* is the identity transformation, then `0` is returned.

If the argument is a collection of transformations *coll*, then the largest integer which is the image a point moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity transformations, then `LargestImageOfMovedPoint` returns `0`.

Example

```
gap> S:=FullTransformationSemigroup(5);
<full transformation semigroup on 5 pts>
gap> LargestImageOfMovedPoint(S);
5
gap> S:=Semigroup(IdentityTransformation);;
gap> LargestImageOfMovedPoint(S);
0
gap> f:=Transformation( [ 1, 2, 3, 6, 6, 6 ] );;
gap> LargestImageOfMovedPoint(f);
6
```

53.5.11 FlatKernelOfTransformation

▷ `FlatKernelOfTransformation(`*f[, n]*`)` (function)

Returns: The flat kernel of a transformation.

If the kernel classes of the transformation *f* on `[1..`*n*`]` are $K_1, \ldots, K_r$, then `FlatKernelOfTransformation` returns a list `L` such that `L[i]=j` for all `i` in K_j. For a given transformation and positive integer *n*, there is a unique such list.

If the optional second argument *n* is not present, then the degree of *f* is used by defualt.

Example

```
gap> f:=Transformation( [ 10, 3, 7, 10, 1, 5, 9, 2, 6, 10 ] );;
gap> FlatKernelOfTransformation(f);
[ 1, 2, 3, 1, 4, 5, 6, 7, 8, 1 ]
```

53.5.12 KernelOfTransformation

▷ `KernelOfTransformation(`*f[, n, bool]*`)` (attribute)

Returns: The kernel of a transformation.

When the arguments are a transformation *f*, a positive integer *n*, and `true`, `KernelOfTransformation` returns the kernel of the transformation *f* on `[1..`*n*`]` as a set of sets of positive integers. If the argument *bool* is `false`, then only the non-singleton classes are returned.

The second and third arguments are optional, the default values are the degree of *f* and `true`.

Example

```
gap> f:=Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 11, 1, 12, 5 ] );;
gap> KernelOfTransformation(f);
[ [ 1, 4 ], [ 2, 5 ], [ 3 ], [ 6, 7 ], [ 8, 10 ], [ 9 ], [ 11 ],
  [ 12 ] ]
```

```
gap> KernelOfTransformation(f, 5);
[ [ 1, 4 ], [ 2, 5 ], [ 3 ] ]
gap> KernelOfTransformation(f, 5, false);
[ [ 1, 4 ], [ 2, 5 ] ]
gap> KernelOfTransformation(f, 15);
[ [ 1, 4 ], [ 2, 5 ], [ 3 ], [ 6, 7 ], [ 8, 10 ], [ 9 ], [ 11 ],
  [ 12 ], [ 13 ], [ 14 ], [ 15 ] ]
gap> KernelOfTransformation(f, false);
[ [ 1, 4 ], [ 2, 5 ], [ 6, 7 ], [ 8, 10 ] ]
```

53.5.13 InverseOfTransformation

▷ `InverseOfTransformation(f)` (operation)

Returns: A transformation.

`InverseOfTransformation` returns a semigroup inverse of the transformation `f` in the full transformation semigroup. An *inverse* of `f` is any transformation g such that `f*g*f=f` and `g*f*g=g`. Every transformation has at least one inverse in a full transformation semigroup.

Example

```
gap> f:=Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 1, 5 ] );;
gap> g:=InverseOfTransformation(f);
Transformation( [ 8, 1, 1, 1, 10, 2, 3, 1, 6, 1 ] )
gap> f*g*f;
Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 1, 5 ] )
gap> g*f*g;
Transformation( [ 8, 1, 1, 1, 10, 2, 3, 1, 6, 1 ] )
```

53.5.14 Inverse (for a transformation)

▷ `Inverse(f)` (attribute)

Returns: A transformation.

If the transformation `f` is a bijection, then `Inverse` or `f^-1` returns the inverse of `f`. If `f` is not a bijection, then `fail` is returned.

Example

```
gap> Transformation( [ 3, 8, 12, 1, 11, 9, 9, 4, 10, 5, 10, 6 ] )^-1;
fail
gap> Transformation( [ 2, 3, 1 ] )^-1;
Transformation( [ 3, 1, 2 ] )
```

53.5.15 IndexPeriodOfTransformation

▷ `IndexPeriodOfTransformation(f)` (attribute)

Returns: A pair of positive integers.

Returns the least positive integers `m` and `r` such that `f^(m+r)=f^m`, which are known as the *index* and *period* of the transformation `f`.

Example

```
gap> f:=Transformation( [ 3, 4, 4, 6, 1, 3, 3, 7, 1 ] );;
gap> IndexPeriodOfTransformation(f);
[ 2, 3 ]
gap> f^2=f^5;
```

```
true
gap> IndexPeriodOfTransformation(IdentityTransformation);
[ 1, 1 ]
gap> IndexPeriodOfTransformation(Transformation([1,2,1]));
[ 1, 1 ]
gap> IndexPeriodOfTransformation(Transformation([1,2,3]));
[ 1, 1 ]
gap> IndexPeriodOfTransformation(Transformation([1,3,2]));
[ 1, 2 ]
```

53.5.16 SmallestIdempotentPower (for a transformation)

▷ `SmallestIdempotentPower(f)` (attribute)

Returns: A positive integer.

This function returns the least positive integer `n` such that the transformation `f^n` is an idempotent. The smallest idempotent power of `f` is the least multiple of the period of `f` that is greater than or equal to the index of `f`; see `IndexPeriodOfTransformation` (53.5.15).

Example

```
gap> f:=Transformation( [ 6, 7, 4, 1, 7, 4, 6, 1, 3, 4 ] );;
gap> SmallestIdempotentPower(f);
3
gap> f:=Transformation( [ 6, 6, 6, 2, 7, 1, 5, 3, 10, 6 ] );;
gap> SmallestIdempotentPower(f);
2
```

53.5.17 ComponentsOfTransformation

▷ `ComponentsOfTransformation(f)` (attribute)

Returns: A list of lists of positive integers.

`ComponentsOfTransformation` returns a list of the components of the transformation `f`. Each component is a subset of `[1..DegreeOfTransformation(f)]`, and the union of the components is `[1..DegreeOfTransformation(f)]`.

Example

```
gap> f:=Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] );
Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] )
gap> ComponentsOfTransformation(f);
[ [ 1, 4, 6, 9 ], [ 2, 3, 5, 7, 10, 11, 12 ], [ 8 ] ]
gap> f:=AsTransformation((1,8,2,4,11,5,10)(3,7)(9,12));
Transformation( [ 8, 4, 7, 11, 10, 6, 3, 2, 12, 1, 5, 9 ] )
gap> ComponentsOfTransformation(f);
[ [ 1, 2, 4, 5, 8, 10, 11 ], [ 3, 7 ], [ 6 ], [ 9, 12 ] ]
```

53.5.18 NrComponentsOfTransformation

▷ `NrComponentsOfTransformation(f)` (attribute)

Returns: A positive integer.

`NrComponentsOfTransformation` returns the number of components of the transformation `f` on the range `[1..DegreeOfTransformation(f)]`.

Example

```
gap> f:=Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] );
Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] )
gap> NrComponentsOfTransformation(f);
3
gap> f:=AsTransformation((1,8,2,4,11,5,10)(3,7)(9,12));
Transformation( [ 8, 4, 7, 11, 10, 6, 3, 2, 12, 1, 5, 9 ] )
gap> NrComponentsOfTransformation(f);
4
```

53.5.19 ComponentRepsOfTransformation

▷ `ComponentRepsOfTransformation(`*f*`)` (attribute)

Returns: A list of lists of positive integers.

`ComponentRepsOfTransformation` returns the representatives, in the following sense, of the components of the transformation *f*. For every `i` in `[1..DegreeOfTransformation(f)]` there exists a representative `j` and a positive integer `k` such that `i^(f^k)=j`. The representatives returned by `ComponentRepsOfTransformation` are partitioned according to the component they belong to. `ComponentRepsOfTransformation` returns the least number of representatives.

Example

```
gap> f:=Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] );
Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] )
gap> ComponentRepsOfTransformation(f);
[ [ 3, 10 ], [ 9 ], [ 8 ] ]
gap> f:=AsTransformation((1,8,2,4,11,5,10)(3,7)(9,12));
Transformation( [ 8, 4, 7, 11, 10, 6, 3, 2, 12, 1, 5, 9 ] )
gap> ComponentRepsOfTransformation(f);
[ [ 1 ], [ 3 ], [ 6 ], [ 9 ] ]
```

53.5.20 CyclesOfTransformation

▷ `CyclesOfTransformation(`*f[, list]*`)` (attribute)

Returns: A list of lists of positive integers.

When the arguments of this function are a transformation*f* and a list *list*, it returns a list of the cycles of the components of *f* containing any element of *list*.

If the optional second argument is not present, then the range `[1..DegreeOfTransformation(`*f*`)]` is used as the default value for *list*.

Example

```
gap> f:=Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] );
Transformation( [ 6, 12, 11, 1, 7, 6, 2, 8, 4, 7, 5, 12 ] )
gap> CyclesOfTransformation(f);
[ [ 6 ], [ 12 ], [ 8 ] ]
gap> CyclesOfTransformation(f, [ 1, 2, 4 ] );
[ [ 6 ], [ 12 ] ]
gap> CyclesOfTransformation(f, [ 1 .. 17 ]);
[ [ 6 ], [ 12 ], [ 8 ], [ 13 ], [ 14 ], [ 15 ], [ 16 ], [ 17 ] ]
```

53.5.21 CycleTransformationInt

▷ `CycleTransformationInt(f, n)` (attribute)

Returns: A list of positive integers.

If `f` is a transformation and `n` is a positive integer, then `CycleTransformationInt` returns the cycle of the component of `f` containing `n`.

Example

```
gap> f:=Transformation( [ 6, 2, 8, 4, 7, 5, 8, 3, 5, 8 ] );;
gap> CycleTransformationInt(f, 1);
[ 8, 3 ]
gap> CycleTransformationInt(f, 12);
[ 12 ]
gap> CycleTransformationInt(f, 5);
[ 8, 3 ]
```

53.5.22 LeftOne (for a transformation)

▷ `LeftOne(f)` (attribute)

▷ `RightOne(f)` (attribute)

Returns: A transformation.

`LeftOne` returns an idempotent transformation `e` such that the kernel (with respect to the degree of `f`) of `e` equals the kernel of the transformation `f` and `e*f=f`.

`RightOne` returns an idempotent transformation `e` such that the image set (with respect to the degree of `f`) of `e` equals the image set of `f` and `f*e=f`.

Example

```
gap> f:=Transformation( [ 11, 10, 2, 11, 4, 4, 7, 6, 9, 10, 1, 11 ] );;
gap> e:=RightOne(f);
Transformation( [ 1, 2, 2, 4, 4, 6, 7, 7, 9, 10, 11, 11 ] )
gap> IsIdempotent(e);
true
gap> f*e=f;
true
gap> e:=LeftOne(f);
Transformation( [ 1, 2, 3, 1, 5, 5, 7, 8, 9, 2, 11, 1 ] )
gap> e*f=f;
true
gap> IsIdempotent(e);
true
```

53.5.23 TrimTransformation

▷ `TrimTransformation(f[, n])` (function)

Returns: Nothing.

It can happen that the internal representation of a transformation uses more memory than necessary. For example, this can happen when composing transformations where it is possible that the resulting transformation `f` has belongs to `IsTrans4Rep` and has its images stored as 32-bit integers, while none of its moved points exceeds 65536. The purpose of `TrimTransformation` is to change the internal representation of such an `f` to remove the trailing fixed points.

If the optional second argument *n* is provided, then the internal representation of *f* is reduced to the images of the first *n* positive integers. Please note that it must be the case that i^*f*<=n for all i in the range [1..*n*] otherwise the resulting object will not define a transformation.

If the optional second argument is not included, then the degree of *f* is used by default.

The transformation *f* is changed in-place, and nothing is returned by this function.

Example

```
gap> f:=Transformation( [ 1 .. 2^16 ], x-> x+1 );
<transformation on 65537 pts with rank 65536>
gap> g:=Transformation( [ 1 .. 2^16+1 ], function(x)
> if x=1 or x=65537 then return x; else return x-1; fi; end);
<transformation on 65536 pts with rank 65535>
gap> h:=g*f;
Transformation( [ 2, 2 ] )
gap> DegreeOfTransformation(h); IsTrans4Rep(h); MemoryUsage(h);
65537
true
262188
gap> TrimTransformation(h); h;
Transformation( [ 2, 2 ] )
gap> DegreeOfTransformation(h); IsTrans4Rep(h); MemoryUsage(h);
2
false
44
```

53.6 Displaying transformations

It is possible to change the way that GAP displays transformations using the user preferences TransformationDisplayLimit and NotationForTransformations; see Section UserPreference (3.2.3) for more information about user preferences.

If f is a transformation where degree n exceeds the value of the user preference TransformationDisplayLimit, then f is displayed as:

Example

```
<transformation on n pts with rank r>
```

where r is the rank of f relative to n. The idea is to abbreviate the display of transformations defined on many points. The default value for the TransformationDisplayLimit is 100.

If the degree of f does not exceed the value of TransformationDisplayLimit, then how f is displayed depends on the value of the user preference NotationForTransformations.

There are two possible values for NotationForTransformations:

input
: With this option a transformation *f* is displayed in as: Transformation(ImageListOfTransformation(*f*, n) where n is the degree of *f*. The only exception is the identity transformation, which is displayed as: IdentityTransformation.

fr
: With this option a transformation *f* is displayed in as: <transformation: ImageListOfTransformation(*f*, n)> where n is the largest moved point of *f*. The only exception is the identity transformation, which is displayed as: <identity transformation>.

Example

```
gap> SetUserPreference("TransformationDisplayLimit", 12);
gap> f:=Transformation([ 3, 8, 12, 1, 11, 9, 9, 4, 10, 5, 10, 6 ]);
<transformation on 12 pts with rank 10>
gap> SetUserPreference("TransformationDisplayLimit", 100);
gap> f;
Transformation( [ 3, 8, 12, 1, 11, 9, 9, 4, 10, 5, 10, 6 ] )
gap> SetUserPreference("NotationForTransformations", "fr");
gap> f;
<transformation: 3,8,12,1,11,9,9,4,10,5,10,6>
```

53.7 Semigroups of transformations

As mentioned at the start of the chapter, every semigroup is isomorphic to a semigroup of transformations, and in this section we describe the functions in GAP specific to transformation semigroups. For more information about semigroups in general see Chapter 51.

The Semigroups package contains many additional functions and methods for computing with semigroups of transformations. In particular, Semigroups contains more efficient methods than those available in the GAP library (and in many cases more efficient than any other software) for creating semigroups of transformations, calculating their Green"s classes, size, elements, group of units, minimal ideal, small generating sets, testing membership, finding the inverses of a regular element, factorizing elements over the generators, and more. Since a transformation semigroup is also a transformation collection, there are special methods for `MovedPoints` (53.5.5), `NrMovedPoints` (53.5.6), `LargestMovedPoint` (53.5.8), `SmallestMovedPoint` (53.5.7), `LargestImageOfMovedPoint` (53.5.10), and `SmallestImageOfMovedPoint` (53.5.9), when applied to a transformation semigroup.

53.7.1 IsTransformationSemigroup

▷ `IsTransformationSemigroup(`*obj*`)` (property)
▷ `IsTransformationMonoid(`*obj*`)` (property)

Returns: `true` or `false`.

A *transformation semigroup* is simply a semigroup consisting of transformations. An object *obj* is a transformation semigroup in GAP if it satisfies `IsSemigroup` (51.1.1) and `IsTransformationCollection` (53.1.2).

A *transformation monoid* is a monoid consisting of transformations. An object *obj* is a transformation monoid in GAP if it satisfies `IsMonoid` (51.2.1) and `IsTransformationCollection` (53.1.2).

Note that it is possible for a transformation semigroup to have a multiplicative neutral element (i.e. an identity element) but not to satisfy `IsTransformationMonoid`. For example,

Example

```
gap> f:=Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 1, 5 ] );;
gap> S:=Semigroup(f, One(f));
<commutative transformation monoid on 10 pts with 1 generator>
gap> IsMonoid(S);
true
gap> IsTransformationMonoid(S);
true
```

```
gap> S:=Semigroup( Transformation( [ 3, 8, 1, 4, 5, 6, 7, 1, 10, 10 ] ),
> Transformation( [ 1, 2, 3, 4, 5, 6, 7, 8, 10, 10 ] ) );
<transformation semigroup on 10 pts with 2 generators>
gap> One(S);
fail
gap> MultiplicativeNeutralElement(S);
Transformation( [ 1, 2, 3, 4, 5, 6, 7, 8, 10, 10 ] )
gap> IsMonoid(S);
false
```

In this example S cannot be converted into a monoid using AsMonoid (51.2.5) since the One (31.10.2) of any element in S differs from the multiplicative neutral element.

For more details see IsMagmaWithOne (35.1.2).

53.7.2 DegreeOfTransformationSemigroup

▷ DegreeOfTransformationSemigroup(*S*) (attribute)

Returns: A non-negative integer.

The *degree* of a transformation semigroup *S* is just the maximum of the degrees of the elements of *S*.

Example

```
gap> S:=Semigroup(Transformation( [ 3, 8, 1, 4, 5, 6, 7, 1, 10, 10, 11 ] ),
> Transformation( [ 1, 2, 3, 4, 5, 6, 7, 8, 1, 1, 11 ] ) );
<transformation semigroup on 10 pts with 2 generators>
gap> DegreeOfTransformationSemigroup(S);
10
```

53.7.3 FullTransformationSemigroup

▷ FullTransformationSemigroup(*n*) (function)

▷ FullTransformationMonoid(*n*) (function)

Returns: The full transformation semigroup of degree *n*.

If *n* is a positive integer, then FullTransformationSemigroup returns the monoid consisting of all transformations with degree at most *n*, called the *full transformation semigroup*.

The full transformation semigroup is regular, has *n*^*n* elements, and is generated by any set containing transformations that generate the symmetric group on *n* points and any transformation of rank *n*-1.

FulTransformationMonoid is a synonym for FullTransformationSemigroup.

Example

```
gap> FullTransformationSemigroup(1234);
<full transformation semigroup on 1234 pts>
```

53.7.4 IsFullTransformationSemigroup

▷ IsFullTransformationSemigroup(*S*) (property)

▷ IsFullTransformationMonoid(*S*) (property)

Returns: true or false.

If the transformation semigroup *S* of degree n contains every transformation of degree at most n, then IsFullTransformationSemigroup return true and otherwise it returns false.

IsFullTransformationMonoid is a synonym of IsFullTransformationSemigroup. It is common in the literature for the full transformation monoid to be referred to as the full transformation semigroup.

Example

```
gap> S:=Semigroup(AsTransformation((1,3,4,2), 5), AsTransformation((1,3,5), 5),
> Transformation( [ 1, 1, 2, 3, 4 ] ));
<transformation semigroup on 5 pts with 3 generators>
gap> IsFullTransformationSemigroup(S);
true
gap> S;
<full transformation semigroup on 5 pts>
gap> IsFullTransformationMonoid(S);
true
gap> S:=FullTransformationSemigroup(5);;
gap> IsFullTransformationSemigroup(S);
true
```

53.7.5 IsomorphismTransformationSemigroup

▷ IsomorphismTransformationSemigroup(*S*) (attribute)
▷ IsomorphismTransformationMonoid(*S*) (attribute)

Returns: An isomorphism to a transformation semigroup or monoid.

Returns an isomorphism from the finite semigroup *S* to a transformation semigroup. For most types of objects in GAP the degree of this transformation semigroup will be equal to the size of *S* plus 1.

Let *S*^1 denote the monoid obtained from *S* by adjoining an identity element. Then *S* acts faithfully on *S*^1 by right multiplication, i.e. every element of *S* describes a transformation on 1,..,|S|+1. The isomorphism from *S* to the transformation semigroup described in this way is called the *right regular representation* of *S*. In most cases, IsomorphismTransformationSemigroup will return the right regular representation of *S*.

As exceptions, if *S* is a permutation group or a partial perm semigroup, then the elements of *S* act naturally and faithfully by transformations on the values from 1 to the largest moved point of *S*.

If *S* is a finitely presented semigroup, then the Todd-Coxeter approach will be attempted.

IsomorphismTransformationMonoid differs from IsomorphismTransformationSemigroup only in that its range is a transformation monoid, and not only a semigroup, when the semigroup *S* is a monoid.

Example

```
gap> gens:= [ [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
>  [ [ Z(3), Z(3)^0 ], [ Z(3), 0*Z(3) ] ],
>  [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3) ] ] ];;
gap> S:=Semigroup(gens);;
gap> Size(S);
81
gap> IsomorphismTransformationSemigroup(S);;
gap> S:=SymmetricInverseSemigroup(4);
<symmetric inverse semigroup on 4 pts>
gap> IsomorphismTransformationMonoid(S);
MappingByFunction( <symmetric inverse semigroup on 4 pts>,
<transformation monoid on 5 pts with 4 generators>
 , function( x ) ... end, <Operation "AsPartialPerm"> )
```

```
gap> G:=Group((1,2,3));
Group([ (1,2,3) ])
gap> IsomorphismTransformationMonoid(G);
MappingByFunction( Group([ (1,2,3) ]), <commutative transformation
 monoid on 3 pts with 1 generator>
 , function( x ) ... end, function( x ) ... end )
```

53.7.6 AntiIsomorphismTransformationSemigroup

▷ `AntiIsomorphismTransformationSemigroup(`*S*`)` (operation)

Returns: An anti-isomorphism.

If *S* is a semigroup, then `AntiIsomorphismTransformationSemigroup` returns an anti-isomorphism from *S* to a transformation semigroup. At present, the degree of the resulting transformation semigroup equals the size of *S* plus 1, and, consequently, this function is of limited use.

Example

```
gap> S:=Semigroup( Transformation( [ 5, 5, 1, 1, 3 ] ),
> Transformation( [ 2, 4, 1, 5, 5 ] ) );
<transformation semigroup on 5 pts with 2 generators>
gap> Size(S);
172
gap> AntiIsomorphismTransformationSemigroup(S);
MappingByFunction( <transformation semigroup of size 172,
 on 5 pts with 2 generators>, <transformation semigroup
 on 173 pts with 2 generators>
 , function( x ) ... end, function( x ) ... end )
```

Chapter 54

Partial permutations

This chapter describes the functions in GAP for partial permutations.

A *partial permutation* in GAP is simply an injective function from any finite set of positive integers to any other finite set of positive integers. The largest point on which a partial permutation can be defined, and the largest value that the image of such a point can have, are defined by certain architecture dependent limits.

Every inverse semigroup is isomorphic to an inverse semigroup of partial permutations and, as such, partial permutations are to inverse semigroup theory what permutations are to group theory and transformations are to semigroup theory. In this way, partial permutations are the elements of inverse partial permutation semigroups.

A partial permutations in GAP acts on a finite set of positive integers on the right. The image of a point `i` under a partial permutation `f` is expressed as `i^f` in GAP. This action is also implemented by the function `OnPoints` (41.2.1). The preimage of a point `i` under the partial permutation `f` can be computed using `i/f` without constructing the inverse of `f`. Partial permutations in GAP are created using the operations described in Section 54.2. Partial permutations are, by default, displayed in component notation, which is described in Section 54.6.

The fundamental attributes of a partial permutation are:

Domain

The *domain* of a partial permutation is just the set of positive integers where it is defined; see `DomainOfPartialPerm` (54.3.4). We will denote the domain of a partial permutation `f` by dom(`f`).

Degree

The *degree* of a partial permutation `f` is just the largest positive integer where `f` is defined. In other words, the degree of `f` is the largest element in the domain of `f`; see `DegreeOfPartialPerm` (54.3.1).

Image list

The *image list* of a partial permutation `f` is the list `[i_1^f, i_2^f, .. , i_n^f]` where the domain of `f` is `[i_1, i_2, .., i_n]` see `ImageListOfPartialPerm` (54.3.6). For example, the partial perm sending `1` to `5` and `2` to `4` has image list `[ 5, 4 ]`.

Image set

The *image set* of a partial permutation `f` is just the set of points in the image list (i.e. the image

list after it has been sorted into increasing order); see ImageSetOfPartialPerm (54.3.7). We will denote the image set of a partial permutation f by im(f).

Codegree

The *codegree* of a partial permutation f is just the largest positive integer of the form i^f for any i in the domain of f. In other words, the codegree of f is the largest element in the image of f; see CodegreeOfPartialPerm (54.3.2).

Rank

The *rank* of a partial permutation f is the size of its domain, or equivalently the size of its image set or image list; see RankOfPartialPerm (54.3.3).

A *functional digraph* is a directed graph where every vertex has out-degree 1. A partial permutation f can be thought of as a functional digraph with vertices [1..DegreeOfPartialPerm(f)] and edges from i to i^f for every i. A *component* of a partial permutation is defined as a component of the corresponding functional digraph. More specifically, i and j are in the same component if and only if there are $i = v_0, v_1, \ldots, v_n = j$ such that either $v_{k+1} = v_k^f$ or $v_k = v_{k+1}^f$ for all k.

If S is a semigroup and s is an element of S, then an element t in S is a *semigroup inverse* for s if s*t*s=s and t*s*t=t; see, for example, InverseOfTransformation (53.5.13). A semigroup in which every element has a unique semigroup inverse is called an *inverse semigroup*.

Every partial permutation belongs to a symmetric inverse monoid; see SymmetricInverseSemigroup (54.7.3). Inverse semigroups of partial permutations are hence inverse subsemigroups of the symmetric inverse monoids.

The inverse f^-1 of a partial permutation f is simply the partial permutation that maps i^f to i for all i in the image of f. It follows that the domain of f^-1 equals the image of f and that the image of f^-1 equals the domain of f. The inverse f^-1 is the unique partial permutation with the property that f*f^-1*f=f and f^-1*f*f^-1=f^-1. In other words, f^-1 is the unique semigroup inverse of f in the symmetric inverse monoid.

If f and g are partial permutations, then the domain and image of the product are:

$$\mathrm{dom}(fg) = (\mathrm{im}(f) \cap \mathrm{dom}(g))f^{-1} \text{ and } \mathrm{im}(fg) = (\mathrm{im}(f) \cap \mathrm{dom}(g))g$$

A partial permutation is an idempotent if and only if it is the identity function on its domain. The products f*f^-1 and f^-1*f are just the identity functions on the domain and image of f, respectively. It follows that f*f^-1 is a left identity for f and f^-1*f is a right identity. These products will be referred to here as the *left one* and *right one* of the partial permutation f; see LeftOne (54.3.21). The *one* of a partial permutation is just the identity on the union of its domain and its image, and the *zero* of a partial permutation is just the empty partial permutation; see One (54.3.22) and Zero (54.3.23).

If S is an arbitrary inverse semigroup, the *natural partial order* on S is defined as follows: for elements x and y of S we say x≤y if there exists an idempotent element e in S such that x=ey. In the context of the symmetric inverse monoid, a partial permutation f is less than or equal to a partial permutation g in the natural partial order if and only if f is a restriction of g. The natural partial order is a meet semilattice, in other words, every pair of elements has a greatest lower bound; see MeetOfPartialPerms (54.2.5).

Note that unlike permutations, partial permutations do not fix unspecified points but are simply undefined on such points; see Chapter 42. Similar to permutations, and unlike transformations, it is possible to multiply any two partial permutations in GAP.

Internally, GAP stores a partial permutation f as a list consisting of the codegree of f and the images i^f of the points i that are less than or equal to the degree of f; the value 0 is stored where i^f

is undefined. The domain and image set of `f` are also stored after either of these values is computed. When the codegree of a partial permutation `f` is less than 65536, the codegree and images `i^f` are stored as 16-bit integers, the domain and image set are subobjects of `f` which are immutable plain lists of GAP integers. When the codegree of `f` is greater than or equal to 65536, the codegree and images are stored as 32-bit integers; the domain and image set are stored in the same way as before. A partial permutation belongs to `IsPPerm2Rep` if it is stored using 16-bit integers and to `IsPPerm4Rep` otherwise.

In the names of the GAP functions that deal with partial permutations, the word "Permutation" is usually abbreviated to "Perm", to save typing. For example, the category test function for partial permutations is `IsPartialPerm` (54.1.1).

54.1 The family and categories of partial permutations

54.1.1 IsPartialPerm

▷ `IsPartialPerm(`*obj*`)` (Category)

Returns: `true` or `false`.

Every partial permutation in GAP belongs to the category `IsPartialPerm`. Basic operations for partial permutations are `DomainOfPartialPerm` (54.3.4), `ImageListOfPartialPerm` (54.3.6), `ImageSetOfPartialPerm` (54.3.7), `RankOfPartialPerm` (54.3.3), `DegreeOfPartialPerm` (54.3.1), multiplication of two partial permutations is via `*`, and exponentiation with the first argument a positive integer `i` and second argument a partial permutation `f` where the result is the image `i^f` of the point `i` under `f`. The inverse of a partial permutation `f` can be obtains using `f^-1`.

54.1.2 IsPartialPermCollection

▷ `IsPartialPermCollection(`*obj*`)` (Category)

Every collection of partial permutations belongs to the category `IsPartialPermCollection`. For example, a semigroup of partial permutations belongs in `IsPartialPermCollection`.

54.1.3 PartialPermFamily

▷ `PartialPermFamily` (family)

The family of all partial permutations is `PartialPermFamily`

54.2 Creating partial permutations

There are several ways of creating partial permutations in GAP, which are described in this section.

54.2.1 PartialPerm (for a domain and image)

▷ `PartialPerm(`*dom, img*`)` (function)

▷ `PartialPerm(`*list*`)` (function)

Returns: A partial permutation.

Partial permutations can be created in two ways: by giving the domain and the image, or the dense image list.

Domain and image

The partial permutation defined by a domain *dom* and image *img*, where *dom* is a set of positive integers and *img* is a duplicate free list of positive integers, maps *dom*[i] to *img*[i]. For example, the partial permutation mapping 1 and 5 to 20 and 2 can be created using:

Example

```
PartialPerm([1,5],[20,2]);
```

In this setting, PartialPerm is the analogue in the context of partial permutations of MappingPermListList (42.5.3).

Dense image list

The partial permutation defined by a dense image list *list*, maps the positive integer i to *list*[i] if *list*[i]<>0 and is undefined at i if *list*[i]=0. For example, the partial permutation mapping 1 and 5 to 20 and 2 can be created using:

Example

```
PartialPerm([20,0,0,0,2]);
```

In this setting, PartialPerm is the analogue in the context of partial permutations of PermList (42.5.2).

Regardless of which of these two methods are used to create a partial permutation in GAP the internal representation is the same.

If the largest point in the domain is larger than the rank of the partial permutation, then using the dense image list to define the partial permutation will require less typing; otherwise using the domain and the image will require less typing. For example, the partial permutation mapping 10000 to 1 can be defined using:

Example

```
PartialPerm([10000], [1]);
```

but using the dense image list would require a list with 9999 entries equal to 0 and the final entry equal to 1. On the other hand, the identity on [1,2,3,4,6] can be defined using:

Example

```
PartialPerm([1,2,3,4,0,6]);
```

Please note that a partial permutation in GAP is never a permutation nor is a permutation ever a partial permutation. For example, the permutation (1,4,2) fixes 3 but the partial permutation PartialPerm([4,1,0,2]); is not defined on 3.

54.2.2 PartialPermOp

▷ PartialPermOp(*obj, list[, func]*) (function)

▷ PartialPermOpNC(*obj, list[, func]*) (function)

Returns: A partial permutation or fail.

PartialPermOp returns the partial permutation that corresponds to the action of the object *obj* on the domain or list *list* via the function *func*. If the optional third argument *func* is not specified,

then the action OnPoints (41.2.1) is used by default. Note that the returned partial permutation refers to the positions in *list* even if *list* itself consists of integers.

This function is the analogue in the context of partial permutations of Permutation (**Reference: Permutation**) or TransformationOp (53.2.5).

If *obj* does not map the elements of *list* injectively, then fail is returned.

PartialPermOpNC does not check that *obj* maps elements of *list* injectively or that a partial permutation is defined by the action of *obj* on *list* via *func*. This function should be used only with caution, in situations where it is guaranteed that the arguments have the required properties.

Example

```
gap> f:=Transformation( [ 9, 10, 4, 2, 10, 5, 9, 10, 9, 6 ] );;
gap> PartialPermOp(f, [ 6 .. 8 ], OnPoints);
[1,4][2,5][3,6]
```

54.2.3 RestrictedPartialPerm

▷ RestrictedPartialPerm(*f, set*) (operation)

Returns: A partial permutation.

RestrictedPartialPerm returns a new partial permutation that acts on the points in the set of positive integers *set* in the same way as the partial permutation *f*, and that is undefined on those points that are not in *set*.

Example

```
gap> f:=PartialPerm( [ 1, 3, 4, 7, 8, 9 ], [ 9, 4, 1, 6, 2, 8 ] );;
gap> RestrictedPartialPerm(f, [ 2, 3, 6, 10 ] );
[3,4]
```

54.2.4 JoinOfPartialPerms

▷ JoinOfPartialPerms(*arg*) (function)

▷ JoinOfIdempotentPartialPermsNC(*arg*) (function)

Returns: A partial permutation or fail.

The join of partial permutations *f* and *g* is just the join, or supremum, of *f* and *g* under the natural partial ordering of partial permutations.

JoinOfPartialPerms returns the union of the partial permutations in its argument if this defines a partial permutation, and fail if it is not. The argument *arg* can be a partial permutation collection or a number of partial permutations.

The function JoinOfIdempotentPartialPermsNC returns the join of its argument which is assumed to be a collection of idempotent partial permutations or a number of idempotent partial permutations. It is not checked that the arguments are idempotents. The performance of this function is higher than JoinOfPartialPerms when it is known *a priori* that the argument consists of idempotents.

The union of *f* and *g* is a partial permutation if and only if *f* and *g* agree on the intersection $\mathrm{dom}(f) \cap \mathrm{dom}(g)$ of their domains and the images of $\mathrm{dom}(f) \setminus \mathrm{dom}(g)$ and $\mathrm{dom}(g) \setminus \mathrm{dom}(f)$ under *f* and *g*, respectively, are disjoint.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] );
[3,7][8,1,2,6,9][10,5]
gap> g:=PartialPerm( [ 11, 12, 14, 16, 18, 19 ],
> [ 17, 20, 11, 19, 14, 12 ] );
```

```
[16,19,12,20][18,14,11,17]
gap> JoinOfPartialPerms(f, g);
[3,7][8,1,2,6,9][10,5][16,19,12,20][18,14,11,17]
gap> f:=PartialPerm( [ 1, 4, 5, 6, 7 ], [ 5, 7, 3, 1, 4 ] );
[6,1,5,3](4,7)
gap> g:=PartialPerm( [ 100 ], [ 1 ] );
[100,1]
gap> JoinOfPartialPerms(f, g);
fail
gap> f:=PartialPerm( [ 1, 3, 4 ], [ 3, 2, 4 ] );
[1,3,2](4)
gap> g:=PartialPerm( [ 1, 2, 4 ], [ 2, 3, 4 ] );
[1,2,3](4)
gap> JoinOfPartialPerms(f, g);
fail
gap> f:=PartialPerm( [ 1 ], [ 2 ] );
[1,2]
gap> JoinOfPartialPerms(f, f^-1);
(1,2)
```

54.2.5 MeetOfPartialPerms

▷ `MeetOfPartialPerms(`*arg*`)` (operation)

Returns: A partial permutation.

The meet of partial permutations *f* and *g* is just the meet, or infimum, of *f* and *g* under the natural partial ordering of partial permutations. In other words, the meet is the greatest partial permuation which is a restriction of both *f* and *g*.

Note that unlike the join of partial permutations, the meet always exists.

`MeetOfPartialPerms` returns the meet of the partial permutations in its argument. The argument *arg* can be a partial permutation collection or a number of partial permutations.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 6, 100000 ], [ 2, 6, 7, 1, 5 ] );
[3,7][100000,5](1,2,6)
gap> g:=PartialPerm( [ 1, 2, 3, 4, 6 ], [ 2, 4, 6, 1, 5 ] );
[3,6,5](1,2,4)
gap> MeetOfPartialPerms(f, g);
[1,2]
gap> g:=PartialPerm( [ 1, 2, 3, 5, 6, 7, 9, 10 ],
> [ 4, 10, 5, 6, 7, 1, 3, 2 ] );
[9,3,5,6,7,1,4](2,10)
gap> MeetOfPartialPerms(f, g);
<empty partial perm>
```

54.2.6 EmptyPartialPerm

▷ `EmptyPartialPerm()` (function)

Returns: The empty partial permutation.

The empty partial permutation is returned by this function when it is called with no arguments. This is just short hand for `PartialPerm([]);`.

Example

```
gap> EmptyPartialPerm();
<empty partial perm>
```

54.2.7 RandomPartialPerm

▷ RandomPartialPerm(*n*) (operation)
▷ RandomPartialPerm(*set*) (operation)
▷ RandomPartialPerm(*dom, img*) (operation)

Returns: A random partial permutation.

In its first form, RandomPartialPerm returns a randomly chosen partial permutation where points in the domain and image are bounded above by the positive integer *n*.

Example

```
gap> RandomPartialPerm(10);
[2,9][4,1,6,5][7,3](8)
```

In its second form, RandomPartialPerm returns a randomly chosen partial permutation with points in the domain and image contained in the set of positive integers *set*.

Example

```
gap> RandomPartialPerm([1,2,3,1000]);
[2,3,1000](1)
```

In its third form, RandomPartialPerm creates a randomly chosen partial permutation with domain contained in the set of positive integers *dom* and image contained in the set of positive integers *img*. The arguments *dom* and *img* do not have to have equal length.

Note that it is not guarenteed in either of these cases that partial permutations are chosen with a uniform distribution.

54.3 Attributes for partial permutations

In this section we describe the functions available in GAP for finding various attributes of partial permutations.

54.3.1 DegreeOfPartialPerm

▷ DegreeOfPartialPerm(*f*) (attribute)
▷ DegreeOfPartialPermCollection(*coll*) (attribute)

Returns: A non-negative integer.

The *degree* of a partial permutation *f* is the largest positive integer where it is defined, i.e. the maximum element in the domain of *f*.

The degree a collection of partial permutations *coll* is the largest degree of any partial permutation in *coll*.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] );
[3,7][8,1,2,6,9][10,5]
gap> DegreeOfPartialPerm(f);
10
```

54.3.2 CodegreeOfPartialPerm

▷ `CodegreeOfPartialPerm(f)` (attribute)
▷ `CodegreeOfPartialPermCollection(coll)` (attribute)

Returns: A non-negative integer.

The *codegree* of a partial permutation `f` is the largest positive integer in its image.

The codegree a collection of partial permutations `coll` is the largest codegree of any partial permutation in `coll`.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 8, 10 ], [ 7, 1, 4, 3, 2, 6, 5 ] );
[8,6][10,5,2,1,7](3,4)
gap> CodegreeOfPartialPerm(f);
7
```

54.3.3 RankOfPartialPerm

▷ `RankOfPartialPerm(f)` (attribute)
▷ `RankOfPartialPermCollection(coll)` (attribute)

Returns: A non-negative integer.

The *rank* of a partial permutation `f` is the size of its domain, or equivalently the size of its image set or image list.

The rank of a partial permutation collection `coll` is the size of the union of the domains of the elements of `coll`, or equivalently, the total number of points on which the elements of `coll` act. Note that this is value may not the same as the size of the union of the images of the elements in `coll`.

Example

```
gap> f:=PartialPerm( [ 1, 2, 4, 6, 8, 9 ], [ 7, 10, 1, 9, 4, 2 ] );
[6,9,2,10][8,4,1,7]
gap> RankOfPartialPerm(f);
6
```

54.3.4 DomainOfPartialPerm

▷ `DomainOfPartialPerm(f)` (operation)
▷ `DomainOfPartialPermCollection(f)` (operation)

Returns: A set of positive integers (maybe empty).

The *domain* of a partial permutation `f` is the set of positive integers where `f` is defined.

The domain of a partial permutation collection `coll` is the union of the domains of its elements.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] );
[3,7][8,1,2,6,9][10,5]
gap> DomainOfPartialPerm(f);
[ 1, 2, 3, 6, 8, 10 ]
```

54.3.5 ImageOfPartialPermCollection

▷ `ImageOfPartialPermCollection(coll)` (operation)

Returns: A set of positive integers (maybe empty).

The *image* of a partial permutation collection `coll` is the union of the images of its elements.

Example
```
gap> S:=SymmetricInverseSemigroup(5);
<symmetric inverse semigroup on 5 pts>
gap> ImageOfPartialPermCollection(GeneratorsOfInverseSemigroup(S));
[ 1, 2, 3, 4, 5 ]
```

54.3.6 ImageListOfPartialPerm

▷ `ImageListOfPartialPerm(`*f*`)` (operation)

Returns: The list of images of a partial permutation.

The *image list* of a partial permutation `f` is the list of images of the elements of the domain `f` where `ImageListOfPartialPerm(`*f*`)[i]=DomainOfPartialPerm(`*f*`)[i]^`*f* for any `i` in the range from `1` to the rank of *f*.

Example
```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 8, 10 ], [ 7, 1, 4, 3, 2, 6, 5 ] );
[8,6][10,5,2,1,7](3,4)
gap> ImageListOfPartialPerm(f);
[ 7, 1, 4, 3, 2, 6, 5 ]
```

54.3.7 ImageSetOfPartialPerm

▷ `ImageSetOfPartialPerm(`*f*`)` (attribute)

Returns: The image set of a partial permutation.

The *image set* of a partial permutation `f` is just the set of points in the image list (i.e. the image list after it has been sorted into increasing order).

Example
```
gap> f:=PartialPerm( [ 1, 2, 3, 5, 7, 10 ], [ 10, 2, 3, 5, 7, 6 ] );
[1,10,6](2)(3)(5)(7)
gap> ImageSetOfPartialPerm(f);
[ 2, 3, 5, 6, 7, 10 ]
```

54.3.8 FixedPointsOfPartialPerm (for a partial perm)

▷ `FixedPointsOfPartialPerm(`*f*`)` (operation)

▷ `FixedPointsOfPartialPerm(`*coll*`)` (operation)

Returns: A set of positive integers.

`FixedPointsOfPartialPerm` returns the set of points `i` in the domain of the partial permutation *f* such that `i^`*f*`=i`.

When the argument is a collection of partial permutations *coll*, `FixedPointsOfPartialPerm` returns the set of points fixed by every element of the collection of partial permutations *coll*.

Example
```
gap> f:=PartialPerm(
> [ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ],
> [ 30, 20, 29, 24, 9, 14, 26, 5, 25, 15, 11, 6, 35, 2, 10, 19, 23 ] );;
gap> FixedPointsOfPartialPerm(f);
[  ]
gap> f:=PartialPerm([1..10]);;
gap> FixedPointsOfPartialPerm(f);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

54.3.9 MovedPoints (for a partial perm)

▷ MovedPoints(*f*) (operation)
▷ MovedPoints(*coll*) (operation)

Returns: A set of positive integers.

MovedPoints returns the set of points i in the domain of the partial permutation *f* such that i^*f*<>i.

When the argument is a collection of partial permutations *coll*, MovedPoints returns the set of points moved by some element of the collection of partial permutations *coll*.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ],
> [ 30, 20, 29, 24, 9, 14, 26, 5, 25, 15, 11, 6, 35, 2, 10, 19, 23 ] );;
gap> FixedPointsOfPartialPerm(f);
[  ]
gap> MovedPoints(f);
[ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ]
gap> f:=PartialPerm([1..10]);;
gap> FixedPointsOfPartialPerm(f);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> MovedPoints(f);
[  ]
```

54.3.10 NrFixedPoints (for a partial perm)

▷ NrFixedPoints(*f*) (operation)
▷ NrFixedPoints(*coll*) (operation)

Returns: A positive integer.

NrFixedPoints returns the number of points i in the domain of the partial permutation *f* such that i^*f*=i.

When the argument is a collection of partial permutations *coll*, NrFixedPoints returns the number of points fixed by every element of the collection of partial permutations *coll*.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ],
> [ 30, 20, 29, 24, 9, 14, 26, 5, 25, 15, 11, 6, 35, 2, 10, 19, 23 ] );;
gap> NrFixedPoints(f);
0
gap> f:=PartialPerm([1..10]);;
gap> NrFixedPoints(f);
10
```

54.3.11 NrMovedPoints (for a partial perm)

▷ NrMovedPoints(*f*) (operation)
▷ NrMovedPoints(*coll*) (operation)

Returns: A positive integer.

NrMovedPoints returns the number of points i in the domain of the partial permutation *f* such that i^*f*<>i.

When the argument is a collection of partial permutations *coll*, NrMovedPoints returns the number of points moved by some element of the collection of partial permutations *coll*.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ],
> [ 30, 20, 29, 24, 9, 14, 26, 5, 25, 15, 11, 6, 35, 2, 10, 19, 23 ] );;
gap> NrMovedPoints(f);
17
gap> f:=PartialPerm([1..10]);;
gap> NrMovedPoints(f);
0
```

54.3.12 SmallestMovedPoint (for a partial perm)

▷ SmallestMovedPoint(*f*) (operation)
▷ SmallestMovedPoint(*coll*) (operation)

Returns: A positive integer or infinity.

SmallestMovedPoint returns the smallest positive integer i such that i^*f*<>i if such an i exists. If *f* is an identity partial permutation, then infinity is returned.

If the argument is a collection of partial permutations *coll*, then the smallest point which is moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity partial permutations, then SmallestMovedPoint returns infinity.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ],
> [ 30, 20, 29, 24, 9, 14, 26, 5, 25, 15, 11, 6, 35, 2, 10, 19, 23 ] );
[1,30][3,29][8,14,25,10,26,19][17,15][21,6,9][22,35][32,23]
(2,20,11,5,24)
gap> SmallestMovedPoint(f);
1
gap> f:=PartialPerm([1..10]);;
gap> SmallestMovedPoint(f);
infinity
```

54.3.13 LargestMovedPoint (for a partial perm)

▷ LargestMovedPoint(*f*) (operation)
▷ LargestMovedPoint(*coll*) (operation)

Returns: A positive integer or infinity.

LargestMovedPoint returns the largest positive integers i such that i^*f*<>i if such an i exists. If *f* is the identity partial permutation, then 0 is returned.

If the argument is a collection of partial permutations *coll*, then the largest point which is moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity partial permutations, then LargestMovedPoint returns 0.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 5, 6, 8, 10, 11, 14, 17, 20, 21, 22, 24, 25, 26, 32 ],
> [ 30, 20, 29, 24, 9, 14, 26, 5, 25, 15, 11, 6, 35, 2, 10, 19, 23 ] );
[1,30][3,29][8,14,25,10,26,19][17,15][21,6,9][22,35][32,23]
(2,20,11,5,24)
```

```
gap> LargestMovedPoint(f);
32
gap> f:=PartialPerm([1..10]);;
gap> LargestMovedPoint(f);
0
```

54.3.14 SmallestImageOfMovedPoint (for a partial permutation)

▷ `SmallestImageOfMovedPoint(`*f*`)` (operation)
▷ `SmallestImageOfMovedPoint(`*coll*`)` (operation)

Returns: A positive integer or `infinity`.

`SmallestImageOfMovedPoint` returns the smallest positive integer `i^`*f* such that `i^`*f*`<>i` if such an `i` exists. If *f* is the identity partial permutation, then `infinity` is returned.

If the argument is a collection of partial permutations *coll*, then the smallest integer which is the image a point moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity partial permutations, then `SmallestImageOfMovedPoint` returns `infinity`.

Example

```
gap> S:=SymmetricInverseSemigroup(5);
<symmetric inverse semigroup on 5 pts>
gap> SmallestImageOfMovedPoint(S);
1
gap> S:=Semigroup(PartialPerm([10..100], [10..100]));;
gap> SmallestImageOfMovedPoint(S);
infinity
gap> f:=PartialPerm( [ 1, 2, 3, 6 ] );;
gap> SmallestImageOfMovedPoint(f);
6
```

54.3.15 LargestImageOfMovedPoint (for a partial permutation)

▷ `LargestImageOfMovedPoint(`*f*`)` (operation)
▷ `LargestImageOfMovedPoint(`*coll*`)` (operation)

Returns: A positive integer.

`LargestImageOfMovedPoint` returns the largest positive integer `i^`*f* such that `i^`*f*`<>i` if such an `i` exists. If *f* is an identity partial permutation, then `0` is returned.

If the argument is a collection of partial permutations *coll*, then the largest integer which is the image of a point moved by at least one element of *coll* is returned, if such a point exists. If *coll* only contains identity partial permutations, then `LargestImageOfMovedPoint` returns `0`.

Example

```
gap> S:=SymmetricInverseSemigroup(5);
<symmetric inverse semigroup on 5 pts>
gap> LargestImageOfMovedPoint(S);
5
gap> S:=Semigroup(PartialPerm([10..100], [10..100]));;
gap> LargestImageOfMovedPoint(S);
0
gap> f:=PartialPerm( [ 1, 2, 3, 6 ] );;
gap> LargestImageOfMovedPoint(f);
6
```

54.3.16 IndexPeriodOfPartialPerm

▷ `IndexPeriodOfPartialPerm(f)` (attribute)

Returns: A pair of positive integers.

Returns the least positive integers `m, r` such that `f^(m+r)=f^m`, which are known as the *index* and *period* of the partial permutation `f`.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19 ],
> [ 9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1 ] );
[2,18][3,20][6,5,11,19,1,9][7,16,13][12,14](8)
gap> IndexPeriodOfPartialPerm(f);
[ 6, 1 ]
gap> f^6=f^7;
true
```

54.3.17 SmallestIdempotentPower (for a partial perm)

▷ `SmallestIdempotentPower(f)` (attribute)

Returns: A positive integer.

This function returns the least positive integer `n` such that the partial permutation `f^n` is an idempotent. The smallest idempotent power of `f` is the least multiple of the period of `f` that is greater than or equal to the index of `f`; see `IndexPeriodOfPartialPerm` (54.3.16).

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 7, 8, 10, 11, 13, 18, 19, 20 ],
> [ 5, 1, 7, 3, 10, 2, 12, 14, 11, 16, 6, 9, 15 ] );
[4,3,7,2,1,5,10,14][8,12][13,16][18,6][19,9][20,15](11)
gap> SmallestIdempotentPower(f);
8
gap> f^8;
<identity partial perm on [ 11 ]>
```

54.3.18 ComponentsOfPartialPerm

▷ `ComponentsOfPartialPerm(f)` (attribute)

Returns: A list of lists of positive integer.

`ComponentsOfPartialPerm` returns a list of the components of the partial permutation `f`. Each component is a subset of the domain of `f`, and the union of the components equals the domain.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 19 ],
> [ 20, 4, 6, 19, 9, 14, 3, 12, 17, 5, 15, 13 ] );
[1,20][2,4,19,13,15][7,14][8,3,6][10,12,5,9][11,17]
gap> ComponentsOfPartialPerm(f);
[ [ 1, 20 ], [ 2, 4, 19, 13, 15 ], [ 7, 14 ], [ 8, 3, 6 ],
  [ 10, 12, 5, 9 ], [ 11, 17 ] ]
```

54.3.19 NrComponentsOfPartialPerm

▷ `NrComponentsOfPartialPerm(f)` (attribute)

Returns: A positive integer.

NrComponentsOfPartialPerm returns the number of components of the partial permutation *f* on its domain.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 19 ],
> [ 20, 4, 6, 19, 9, 14, 3, 12, 17, 5, 15, 13 ] );
[1,20][2,4,19,13,15][7,14][8,3,6][10,12,5,9][11,17]
gap> NrComponentsOfPartialPerm(f);
6
```

54.3.20 ComponentRepsOfPartialPerm

▷ ComponentRepsOfPartialPerm(*f*) (attribute)

ComponentRepsOfPartialPerm returns the representatives, in the following sense, of the components of the partial permutation *f*. For every i in the domain of *f* there exists a representative j and a positive integer k such that i^(*f*^k)=j. Unlike transformations, there is exactly one representative for every component of *f*. ComponentRepsOfPartialPerm returns the least number of representatives.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 19 ],
> [ 20, 4, 6, 19, 9, 14, 3, 12, 17, 5, 15, 13 ] );
[1,20][2,4,19,13,15][7,14][8,3,6][10,12,5,9][11,17]
gap> ComponentRepsOfPartialPerm(f);
[ 1, 2, 7, 8, 10, 11 ]
```

54.3.21 LeftOne (for a partial perm)

▷ LeftOne(*f*) (attribute)

▷ RightOne(*f*) (attribute)

Returns: A partial permutation.

LeftOne returns the identity partial permutation e such that the domain and image of e equal the domain of the partial permutation *f* and such that e**f*=f.

RightOne returns the identity partial permutation e such that the domain and image of e equal the image of *f* and such that *f*e=f.

Example

```
gap> f:=PartialPerm( [ 1, 2, 4, 5, 6, 7 ], [ 10, 1, 6, 5, 8, 7 ] );
[2,1,10][4,6,8](5)(7)
gap> RightOne(f);
<identity partial perm on [ 1, 5, 6, 7, 8, 10 ]>
gap> LeftOne(f);
<identity partial perm on [ 1, 2, 4, 5, 6, 7 ]>
```

54.3.22 One (for a partial perm)

▷ One(*f*) (method)

Returns: A partial permutation.

As described in OneImmutable (**Reference: OneImmutable**), One returns the multiplicative neutral element of the partial permutation *f*, which is the identity partial permutation on the union of the domain and image of *f*. Equivalently, the one of *f* is the join of the right one and left one of *f*.

Example
```
gap> f:=PartialPerm([ 1, 2, 3, 4, 5, 7, 10 ], [ 3, 7, 9, 6, 1, 10, 2 ]);;
gap> One(f);
<identity partial perm on [ 1, 2, 3, 4, 5, 6, 7, 9, 10 ]>
```

54.3.23 Zero (for a partial perm)

▷ `Zero(`*f*`)` (method)

Returns: The empty partial permutation.

As described in `ZeroImmutable` (**Reference: ZeroImmutable**), `Zero` returns the multiplicative zero element of the partial permutation *f*, which is the empty partial permutation.

Example
```
gap> f:=PartialPerm([ 1, 2, 3, 4, 5, 7, 10 ], [ 3, 7, 9, 6, 1, 10, 2 ]);;
gap> Zero(f);
<empty partial perm>
```

54.4 Changing the representation of a partial permutation

It is possible that a partial permutation in GAP can be represented by other types of objects, or that other types of GAP objects can be represented by partial permutations. Partial permutations which are mathematically permutations can be converted into permutations in GAP using the function `AsPermutation` (42.5.5). Similarly, a partial permutation can be converted into a transformation using `AsTransformation` (53.3.1).

In this section we describe functions for converting other types of objects in GAP into partial permutations.

54.4.1 AsPartialPerm (for a permutation and a set of positive integers)

▷ `AsPartialPerm(`*f, set*`)` (function)
▷ `AsPartialPerm(`*f*`)` (function)
▷ `AsPartialPerm(`*f, n*`)` (function)

Returns: A partial permutation.

A permutation *f* defines a partial permutation when it is restricted to any finite set of positive integers. `AsPartialPerm` can be used to obtain this partial permutation.

There are several possible arguments for `AsPartialPerm`:

for a permutation and set of positive integers

`AsPartialPerm` returns the partial permutation that equals *f* on the set of positive integers *set* and that is undefined on every other positive integer.

Note that as explained in `PartialPerm` (54.2.1) *a permutation is never a partial permutation* in GAP, please keep this in mind when using `AsPartialPerm`.

for a permutation

`AsPartialPerm` returns the partial permutation that agrees with *f* on `[1..LargestMovedPoint(`*f*`)]` and that is not defined on any other positive integer.

for a permutation and a positive integer

AsPartialPerm returns the partial permutation that agrees with *f* on [1..*n*], when *n* is a positive integer, and that is not defined on any other positive integer.

The operation PartialPermOp (54.2.2) can also be used to convert permutations into partial permutations.

Example

```
gap> f:=(2,8,19,9,14,10,20,17,4,13,12,3,5,7,18,16);;
gap> AsPartialPerm(f);
(1)(2,8,19,9,14,10,20,17,4,13,12,3,5,7,18,16)(6)(11)(15)
gap> AsPartialPerm(f, [ 1, 2, 3 ] );
[2,8][3,5](1)
```

54.4.2 AsPartialPerm (for a transformation and a set of positive integer)

▷ AsPartialPerm(*f*, *set*) (function)

▷ AsPartialPerm(*f*, *n*) (function)

▷ AsPartialPerm(*f*) (function)

Returns: A partial permutation or fail.

A transformation *f* defines a partial permutation when it is restricted to a set of positive integers where it is injective. AsPartialPerm can be used to obtain this partial permutation.

There are several possible arguments for AsPartialPerm:

for a transformation and set of positive integers

AsPartialPerm returns the partial permutation obtained by restricting *f* to the set of positive integers *set* when:

- *set* contains no elements exceeding the degree of *f*;
- *f* is injective on *set*.

for a transformation and a positive integer

AsPartialPerm returns the partial permutation that agrees with *f* on [1..*n*] when *A* is a positive integer and this set satisfies the conditions given above.

for a transformation

Let n denote the degree of *f*. If n^*f*=n and *f* is injective on those i such that i^*f*<>n, then AsPartialPerm returns the partial permutation obtained by restricting *f* to those i such that i^*f*<>n.

AsPartialPerm returns fail if the arguments do not describe a partial permutation.

The operation PartialPermOp (54.2.2) can also be used to convert transformations into partial permutations.

Example

```
gap> f:=Transformation( [ 8, 3, 5, 9, 6, 2, 9, 7, 9 ] );;
gap> AsPartialPerm(f);
[1,8,7](2,3,5,6)
gap> AsPartialPerm(f, 3);
[1,8][2,3,5]
gap> AsPartialPerm(f, [ 2 .. 4 ] );
[2,3,5][4,9]
```

```
gap> f:=Transformation( [ 2, 10, 2, 4, 4, 7, 6, 9, 10, 1 ] );;
gap> AsPartialPerm(f);
fail
```

54.5 Operators and operations for partial permutations

`f ^ -1`
: returns the inverse of the partial permutation `f`.

`i ^ f`
: returns the image of the positive integer `i` under the partial permutation `f` if it is defined and `0` if it is not.

`i / f`
: returns the preimage of the positive integer `i` under the partial permutation `f` if it is defined and `0` if it is not. Note that the inverse of `f` is not calculated to find the preimage of `i`.

`f ^ g`
: returns `g^-1*f*g` when `f` is a partial permutation and `g` is a permutation or partial permutation; see `\^` (31.12.1). This operation requires essentially the same number of steps as multiplying partial permutations, which is around one third as many as inverting and multiplying twice.

`f * g`
: returns the composition of `f` and `g` when `f` and `g` are partial permutations or permutations. The product of a permutation and a partial permutation is returned as a partial permutation.

`f / g`
: returns `f*g^-1` when `f` is a partial permutation and `g` is a permutation or partial permutation. This operation requires essentially the same number of steps as multiplying partial permutations, which is approximately half that required to first invert `g` and then take the produce with `f`.

`LQUO(g, f)`
: returns `g^-1*f` when `f` is a partial permutation and `g` is a permutation or partial permutation. This operation requires essentially the same number of steps as multiplying partial permutations, which is approximately half that required to first invert `g` and then take the produce with `f`.

`f < g`
: returns `true` if the image of `f` on the range from 1 to the degree of `f` is lexicographically less than the corresponding image for `g` and `false` if it is not. See `NaturalLeqPartialPerm` (54.5.4) and `ShortLexLeqPartialPerm` (54.5.5) for additional orders for partial permutations.

`f = g`
: returns `true` if the partial permutation `f` equals the partial permutation `g` and returns `false` if it does not.

54.5.1 PermLeftQuoPartialPerm

▷ PermLeftQuoPartialPerm(*f*, *g*) (operation)
▷ PermLeftQuoPartialPermNC(*f*, *g*) (operation)

Returns: A permutation.

Returns the permutation on the image set of *f* induced by *f*^-1**g* when the partial permutations *f* and *g* have equal domain and image set.

PermLeftQuoPartialPerm verifies that *f* and *g* have equal domains and image sets, and returns an error if they do not. PermLeftQuoPartialPermNC does no checks.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 7 ], [ 7, 9, 10, 4, 2, 5 ] );
[1,7,5,2,9][3,10](4)
gap> g:=PartialPerm( [ 1, 2, 3, 4, 5, 7 ], [ 7, 4, 9, 2, 5, 10 ] );
[1,7,10][3,9](2,4)(5)
gap> PermLeftQuoPartialPerm(f, g);
(2,5,10,9,4)
```

54.5.2 PreImagePartialPerm

▷ PreImagePartialPerm(*f*, *i*) (operation)

Returns: A positive integer or fail.

PreImagePartialPerm returns the preimage of the positive integer *i* under the partial permutation *f* if *i* belongs to the image of *f*. If *i* does not belong to the image of *f*, then fail is returned.

The same result can be obtained by using *i*/*f* as described in Section 54.5.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 5, 9, 10 ], [ 5, 10, 7, 8, 9, 1 ] );
[2,10,1,5,8][3,7](9)
gap> PreImagePartialPerm(f, 8);
5
gap> PreImagePartialPerm(f, 5);
1
gap> PreImagePartialPerm(f, 1);
10
gap> PreImagePartialPerm(f, 10);
2
gap> PreImagePartialPerm(f, 2);
fail
```

54.5.3 ComponentPartialPermInt

▷ ComponentPartialPermInt(*f*, *i*) (operation)

Returns: A set of positive integers.

ComponentPartialPermInt returns the elements of the component of *f* containing *i* that can be obtained by repeatedly applying *f* to *i*.

Example

```
gap> f:=PartialPerm( [ 1, 2, 4, 5, 6, 7, 8, 10, 14, 15, 16, 17, 18 ],
> [ 11, 4, 14, 16, 15, 3, 20, 8, 17, 19, 1, 6, 12 ] );
[2,4,14,17,6,15,19][5,16,1,11][7,3][10,8,20][18,12]
gap> ComponentPartialPermInt(f, 4);
[ 4, 14, 17, 6, 15, 19 ]
```

```
gap> ComponentPartialPermInt(f, 3);
[  ]
gap> ComponentPartialPermInt(f, 10);
[ 10, 8, 20 ]
gap> ComponentPartialPermInt(f, 100);
[  ]
```

54.5.4 NaturalLeqPartialPerm

▷ NaturalLeqPartialPerm(*f*, *g*) (function)

Returns: true or false.

The *natural partial order* $\leq$ on an inverse semigroup S is defined by s$\leq$t if there exists an idempotent e in S such that s=et. Hence if *f* and *g* are partial permutations, then *f*$\leq$*g* if and only if *f* is a restriction of *g*; see RestrictedPartialPerm (54.2.3).

NaturalLeqPartialPerm returns true if *f* is a restriction of *g* and false if it is not. Note that since this is a partial order and not a total order, it is possible that *f* and *g* are incomparable with respect to the natural partial order.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 16, 17, 18, 19 ],
> [ 3, 12, 14, 4, 11, 18, 17, 2, 9, 5, 15, 8, 20, 10, 19 ] );;
gap> g:=RestrictedPartialPerm(f, [ 1, 2, 3, 9, 13, 20 ] );
[1,3,14][2,12]
gap> NaturalLeqPartialPerm(g,f);
true
gap> NaturalLeqPartialPerm(f,g);
false
gap> g:=PartialPerm( [ 1, 2, 3, 4, 5, 8, 10 ],
> [ 7, 1, 4, 3, 2, 6, 5 ] );;
gap> NaturalLeqPartialPerm(f, g);
false
gap> NaturalLeqPartialPerm(g, f);
false
```

54.5.5 ShortLexLeqPartialPerm

▷ ShortLexLeqPartialPerm(*f*, *g*) (function)

Returns: true or false.

ShortLexLeqPartialPerm returns true if the concatenation of the domain and image list of *f* is short-lex less than the corresponding concatenation for *g* and false otherwise.

Note that this is not the natural partial order on partial permutation or the same as comparing *f* and *g* using \<.

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 4, 6, 7, 8, 10 ],
> [ 3, 8, 1, 9, 4, 10, 5, 6 ] );
[2,8,5][7,10,6,4,9](1,3)
gap> g:=PartialPerm( [ 1, 2, 3, 4, 5, 8, 10 ],
> [ 7, 1, 4, 3, 2, 6, 5 ] );
[8,6][10,5,2,1,7](3,4)
gap> f<g;
```

```
true
gap> g<f;
false
gap> ShortLexLeqPartialPerm(f, g);
false
gap> ShortLexLeqPartialPerm(g, f);
true
gap> NaturalLeqPartialPerm(f, g);
false
gap> NaturalLeqPartialPerm(g, f);
false
```

54.5.6 TrimPartialPerm

▷ `TrimPartialPerm(f)` (function)

Returns: Nothing.

It can happen that the internal representation of a partial permutation uses more memory than necessary. For example, by composing a partial permutation with codegree less than 65536 with a partial permutation with codegree greater than 65535. It is possible that the resulting partial permutation `f` has its codegree and images stored as 32-bit integers, while none of its image points exceeds 65536. The purpose of this function is to change the internal representation of such an `f` from using 32-bit to using 16-bit integers.

Note that the partial permutation `f` is changed in-place, and nothing is returned by this function.

Example

```
gap> f:=PartialPerm( [ 1, 2 ], [ 3, 4 ] )
> *PartialPerm( [ 3, 5 ], [ 3, 100000 ] );
[1,3]
gap> IsPPerm4Rep(f);
true
gap> TrimPartialPerm(f); f;
[1,3]
gap> IsPPerm4Rep(f);
false
```

54.6 Displaying partial permutations

It is possible to change the way that GAP displays partial permutations using the user preferences `PartialPermDisplayLimit` and `NotationForPartialPerms`; see Section `UserPreference` (3.2.3) for more information about user preferences.

If `f` is a partial permutation of rank `r` exceeding the value of the user preference `PartialPermDisplayLimit`, then `f` is displayed as:

Example

```
<partial perm on r pts with degree m, codegree n>
```

where the degree and codegree are `m` and `n`, respectively. The idea is to abbreviated the display of partial permutations defined on many points. The default value for the `PartialPermDisplayLimit` is 100.

If the rank of f does not exceed the value of PartialPermDisplayLimit, then how f is displayed depends on the value of the user preference NotationForPartialPerms except in the case that f is the empty partial permutation or an identity partial permutation.

There are three possible values for NotationForPartialPerms user preference, which are described below.

component

Similar to permutations, and unlike transformations, partial permutations can be expressed as products of disjoint permutations and chains. A *chain* is a list c of some length n such that:

- c[1] is an element of the domain of *f* but not the image
- c[i]^*f*=c[i+1] for all i in the range from 1 to n-1.
- c[n] is in the image of *f* but not the domain.

In the display, permutations are displayed as they usually are in GAP, except that fixed points are displayed enclosed in round brackets, and chains are displayed enclosed in square brackets.

Example

```
gap> f:=PartialPerm(
> [ 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 16, 17, 18, 19 ],
> [ 3, 12, 14, 4, 11, 18, 17, 2, 9, 5, 15, 8, 20, 10, 19 ]);;
gap> PrettyPrintPP(f);
[1,3,14][16,8,2,12,15](4)(5,11)[6,18,10,9][7,17,20](19)
```

This option is the most compact way to display a partial permutation and is the default value of the user preference NotationForPartialPerms.

domainimage

With this option a partial permutation f is displayed in the format: DomainOfPartialPerm(*f*)-> ImageListOfPartialPerm(*f*).

Example

```
gap> f:=PartialPerm( [ 1, 2, 4, 5, 6, 7 ], [ 10, 1, 6, 5, 8, 7 ]);
[ 1, 2, 4, 5, 6, 7 ] -> [ 10, 1, 6, 5, 8, 7 ]
```

input

With this option a partial permutation *f* is displayed as: PartialPerm(DomainOfPartialPerm(*f*), ImageListOfPartialPerm(*f*)) which corresponds to the input (of the first type described in PartialPerm (54.2.1)).

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 5, 6, 9, 10 ],
> [ 4, 7, 3, 8, 2, 1, 6 ] );
PartialPerm( [ 1, 2, 3, 5, 6, 9, 10 ], [ 4, 7, 3, 8, 2, 1, 6 ] )
```

Example

```
gap> SetUserPreference("PartialPermDisplayLimit", 12);
gap> UserPreference("PartialPermDisplayLimit");
12
gap> f:=PartialPerm([1,2,3,4,5,6], [6,7,1,4,3,2]);
[5,3,1,6,2,7](4)
gap> f:=PartialPerm(
> [ 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 16, 17, 18, 19 ],
> [ 3, 12, 14, 4, 11, 18, 17, 2, 9, 5, 15, 8, 20, 10, 19 ] );
```

```
<partial perm on 15 pts with degree 19, codegree 20>
gap> SetUserPreference("PartialPermDisplayLimit", 100);
gap> f;
[1,3,14][6,18,10,9][7,17,20][16,8,2,12,15](4)(5,11)(19)
gap> UserPreference("NotationForPartialPerms");
"component"
gap> SetUserPreference("NotationForPartialPerms", "domainimage");
gap> f;
[ 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 16, 17, 18, 19 ] ->
[ 3, 12, 14, 4, 11, 18, 17, 2, 9, 5, 15, 8, 20, 10, 19 ]
gap> SetUserPreference("NotationForPartialPerms", "input");
gap> f;
PartialPerm(
[ 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 16, 17, 18, 19 ],
[ 3, 12, 14, 4, 11, 18, 17, 2, 9, 5, 15, 8, 20, 10, 19 ] )
```

54.7 Semigroups and inverse semigroups of partial permutations

As mentioned at the start of the chapter, every inverse semigroup is isomorphic to a semigroup of partial permutations, and in this section we describe the functions in GAP specific to partial perms semigroups. For more information about semigroups and inverse semigroups see Chapter 51.

The Semigroups package contains many additional functions and methods for computing with semigroups of partial permutations. In particular, Semigroups contains more efficient methods than those available in the GAP library (and in many cases more efficient than any other software) for creating semigroups of transformations, calculating their Green's classes, size, elements, group of units, minimal ideal, small generating sets, testing membership, finding the inverses of a regular element, factorizing elements over the generators, and more.

Since a partial permutation semigroup is also a partial permutation collection, there are special methods for `DomainOfPartialPermCollection` (54.3.4), `ImageOfPartialPermCollection` (54.3.5), `FixedPointsOfPartialPerm` (54.3.8), `MovedPoints` (54.3.9), `NrFixedPoints` (54.3.10), `NrMovedPoints` (54.3.11), `LargestMovedPoint` (54.3.13), and `SmallestMovedPoint` (54.3.12) when applied to a partial permutation semigroup.

54.7.1 IsPartialPermSemigroup

▷ `IsPartialPermSemigroup(`*obj*`)` (property)
▷ `IsPartialPermMonoid(`*obj*`)` (property)

Returns: `true` or `false`.

A *partial perm semigroup* is simply a semigroup consisting of partial permutations, which may or may not be an inverse semigroup. An object *obj* in GAP is a partial perm semigroup if and only if it satisfies `IsSemigroup` (51.1.1) and `IsPartialPermCollection` (54.1.2).

A *partial perm monoid* is a monoid consisting of partial permutations. An object in GAP is a partial perm monoid if it satisfies `IsMonoid` (51.2.1) and `IsPartialPermCollection` (54.1.2).

Note that it is possible for a partial perm semigroup to have a multiplicative neutral element (i.e. an identity element) but not to satisfy `IsPartialPermMonoid`. For example,

Example

```
gap> f:=PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] );;
gap> S:=Semigroup(f, One(f));
```

```
<commutative partial perm monoid on 9 pts with 1 generator>
gap> IsMonoid(S);
true
gap> IsPartialPermMonoid(S);
true
```

Note that unlike transformation semigroups, the `One` (31.10.2) of a partial permutation semigroup must coincide with the multiplicative neutral element, if either exists.

For more details see `IsMagmaWithOne` (35.1.2).

54.7.2 DegreeOfPartialPermSemigroup

▷ `DegreeOfPartialPermSemigroup(`*S*`)` (attribute)
▷ `CodegreeOfPartialPermSemigroup(`*S*`)` (attribute)
▷ `RankOfPartialPermSemigroup(`*S*`)` (attribute)

Returns: A non-negative integer.

The *degree* of a partial permutation semigroup *S* is the largest degree of any partial permutation in *S*.

The *codegree* of a partial permutation semigroup *S* is the largest positive integer in its image.

The *rank* of a partial permutation semigroup *S* is the number of points on which it acts.

Example

```
gap> S:=Semigroup( PartialPerm( [ 1, 2 ], [ 10000, 3 ] ) );
<commutative partial perm semigroup on 2 pts with 1 generator>
gap> DegreeOfPartialPermSemigroup(S);
2
gap> CodegreeOfPartialPermSemigroup(S);
10000
gap> RankOfPartialPermSemigroup(S);
2
```

54.7.3 SymmetricInverseSemigroup

▷ `SymmetricInverseSemigroup(`*n*`)` (operation)
▷ `SymmetricInverseMonoid(`*n*`)` (operation)

Returns: The symmetric inverse semigroup of degree *n*.

If *n* is a non-negative integer, then `SymmetricInverseSemigroup` returns the inverse semigroup consisting of all partial permutations with degree and codegree at most *n*. Note that *n* must be non-negative, but in particular, can equal 0.

The symmetric inverse semigroup has $\sum_{r=0}^{n} \binom{n}{r}^2 \cdot r!$ elements and is generated by any set that of partial permutations that generate the symmetric group on *n* points and any partial permutation of rank *n*-1.

`SymmetricInverseMonoid` is a synonym for `SymmetricInverseSemigroup`.

Example

```
gap> S:=SymmetricInverseSemigroup(5);
<symmetric inverse semigroup on 5 pts>
gap> Size(S);
1546
```

54.7.4 IsSymmetricInverseSemigroup

▷ IsSymmetricInverseSemigroup(*S*) (property)
▷ IsSymmetricInverseMonoid(*S*) (property)

Returns: true or false.

If the partial perm semigroup *S* of degree and codegree *n* equals the symmetric inverse semigroup on *n* points, then IsSymmetricInverseSemigroup return true and otherwise it returns false.

IsSymmetricInverseMonoid is a synonym of IsSymmetricInverseSemigroup. It is common in the literature for the symmetric inverse monoid to be referred to as the symmetric inverse semigroup.

Example

```
gap> S:=Semigroup( AsPartialPerm((1,3,4,2), 5), AsPartialPerm((1,3,5), 5),
> PartialPerm( [ 1, 2, 3, 4 ] ) );
<partial perm semigroup on 5 pts with 3 generators>
gap> IsSymmetricInverseSemigroup(S);
true
gap> S;
<symmetric inverse semigroup on 5 pts>
```

54.7.5 NaturalPartialOrder

▷ NaturalPartialOrder(*S*) (attribute)
▷ ReverseNaturalPartialOrder(*S*) (attribute)

Returns: The natural partial order on an inverse semigroup.

The *natural partial order* $\leq$ on an inverse semigroup *S* is defined by s$\leq$t if there exists an idempotent e in *S* such that s=et. Hence if f and g are partial permutations, then f$\leq$g if and only if f is a restriction of g; see RestrictedPartialPerm (54.2.3).

NaturalPartialOrder returns the natural partial order on the inverse semigroup of partial permutations *S* as a list of sets of positive integers where entry i in NaturalPartialOrder(*S*) is the set of positions in Elements(*S*) of elements which are less than Elements(*S*)[i]. See also NaturalLeqPartialPerm (54.5.4).

ReverseNaturalPartialOrder returns the reverse of the natural partial order on the inverse semigroup of partial permutations *S* as a list of sets of positive integers where entry i in ReverseNaturalPartialOrder(*S*) is the set of positions in Elements(*S*) of elements which are greater than Elements(*S*)[i]. See also NaturalLeqPartialPerm (54.5.4).

Example

```
gap> S:=InverseSemigroup([ PartialPerm( [ 1, 3 ], [ 1, 3 ] ),
> PartialPerm( [ 1, 2 ], [ 3, 2 ] ) ] );
<inverse partial perm semigroup on 3 pts with 2 generators>
gap> Size(S);
11
gap> NaturalPartialOrder(S);
[ [  ], [ 1 ], [ 1 ], [ 1 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 1 ], [ 1 ],
  [ 1, 4, 7 ], [ 1, 4, 8 ], [ 1, 2, 8 ] ]
gap> NaturalLeqPartialPerm(Elements(S)[4], Elements(S)[10]);
true
gap> NaturalLeqPartialPerm(Elements(S)[4], Elements(S)[1]);
false
```

54.7.6 IsomorphismPartialPermMonoid

▷ `IsomorphismPartialPermMonoid(S)` (operation)
▷ `IsomorphismPartialPermSemigroup(S)` (operation)

Returns: An isomorphism.

`IsomorphismPartialPermSemigroup(S)` returns an isomorphism from the inverse semigroup or group S to an inverse semigroup of partial permutations.

`IsomorphismPartialPermMonoid(S)` returns an isomorphism from the inverse monoid or group S to an inverse monoid of partial permutations.

We only describe `IsomorphismPartialPermMonoid`, the corresponding statements for `IsomorphismPartialPermSemigroup` also hold.

Partial permutation semigroups

If S is a partial permutation semigroup that does not satisfy `IsMonoid` (**Reference: IsMonoid**) but where `MultiplicativeNeutralElement(S)<>fail`, then `IsomorphismPartialPermMonoid(S)` returns an isomorphism from S to an inverse monoid of partial permutations.

Permutation groups

If S is a permutation group, then `IsomorphismPartialPermMonoid` returns an isomorphism from S to an inverse monoid of partial permutations on the set `MovedPoints(S)` obtained using `AsPartialPerm` (54.4.1). The inverse of this isomorphism is obtained using `AsPermutation` (42.5.5).

Transformation semigroups

If S is a transformation semigroup satisfying `IsInverseMonoid` (51.4.8), then `IsomorphismPartialPermMonoid` returns an isomorphism from S to an inverse monoid of partial permutations on a subset of `[1..DegreeOfTransformationSemigroup(S)]`.

Example

```
gap> S:=InverseSemigroup(
> PartialPerm( [ 1, 2, 3, 4, 5 ], [ 4, 2, 3, 1, 5 ] ),
> PartialPerm( [ 1, 2, 4, 5 ], [ 3, 1, 4, 2 ] ) );;
gap> IsMonoid(S);
false
gap> iso:=IsomorphismPartialPermMonoid(S);
MappingByFunction( <inverse partial perm semigroup on 5 pts
 with 2 generators>, <inverse partial perm monoid on 5 pts
 with 2 generators>, function( object ) ... end, function( object ) ..\
. end )
gap> Size(S);
508
gap> Size(Range(iso));
508
gap> G:=Group((1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8), (1,4)(2,6)(3,7)(5,8));;
gap> IsomorphismPartialPermSemigroup(G);
MappingByFunction( Group([ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)
(2,5,6,8), (1,4)(2,6)(3,7)(5,8) ]), <inverse partial perm semigroup
on 8 pts
 with 3 generators>, function( p ) ... end, function( f ) ... end )
gap> S:=Semigroup(Transformation( [ 2, 5, 1, 7, 3, 7, 7 ] ),
> Transformation( [ 3, 6, 5, 7, 2, 1, 7 ] ) );;
```

```
gap> iso:=IsomorphismPartialPermMonoid(S);;
gap> MultiplicativeNeutralElement(S)^iso;
<identity partial perm on [ 1, 2, 3, 5, 6, 7 ]>
gap> One(Range(iso));
<identity partial perm on [ 1, 2, 3, 5, 6, 7 ]>
gap> MovedPoints(Range(iso));
[ 1, 2, 3, 5, 6 ]
```

Chapter 55

Additive Magmas

This chapter deals with domains that are closed under addition +, which are called *near-additive magmas* in GAP. Together with the domains closed under multiplication * (see 35), they are the basic algebraic structures. In many cases, the addition is commutative (see `IsAdditivelyCommutative` (55.3.1)), the domain is called an *additive magma* then. Every module (see 57), vector space (see 61), ring (see 56), or field (see 58) is an additive magma. In the cases of all *(near-)additive magma-with-zero* or *(near-)additive magma-with-inverses*, additional additive structure is present (see 55.1).

55.1 (Near-)Additive Magma Categories

55.1.1 IsNearAdditiveMagma

▷ `IsNearAdditiveMagma(`*obj*`)` (Category)

A *near-additive magma* in GAP is a domain A with an associative but not necessarily commutative addition +: $A \times A \to A$.

55.1.2 IsNearAdditiveMagmaWithZero

▷ `IsNearAdditiveMagmaWithZero(`*obj*`)` (Category)

A *near-additive magma-with-zero* in GAP is a near-additive magma A with an operation `0*` (or `Zero` (31.10.3)) that yields the zero element of A.

So a near-additive magma-with-zero `A` does always contain a unique additively neutral element z, i.e., $z+a = a = a+z$ holds for all $a \in A$ (see `AdditiveNeutralElement` (55.3.5)). This zero element z can be computed with the operation `Zero` (31.10.3), by applying this function to A or to any element a in A. The zero element can be computed also as `0 * a`, for any a in A.

Note that it may happen that a near-additive magma containing a zero does *not* lie in the category `IsNearAdditiveMagmaWithZero` (see 31.6).

55.1.3 IsNearAdditiveGroup

▷ `IsNearAdditiveGroup(`*obj*`)` (Category)
▷ `IsNearAdditiveMagmaWithInverses(`*obj*`)` (Category)

A *near-additive group* in GAP is a near-additive magma-with-zero A with an operation `-1*`: $A \to A$ that maps each element a of A to its additive inverse `-1*`a (or `AdditiveInverse( a )`, see `AdditiveInverse` (31.10.9)).

The addition `+` of A is assumed to be associative, so a near-additive group is not more than a *near-additive magma-with-inverses*. `IsNearAdditiveMagmaWithInverses` is just a synonym for `IsNearAdditiveGroup`, and can be used alternatively in all function names involving the string `"NearAdditiveGroup"`.

Note that not every trivial near-additive magma is a near-additive magma-with-zero, but every trivial near-additive magma-with-zero is a near-additive group.

55.1.4 IsAdditiveMagma

▷ `IsAdditiveMagma(`*obj*`)` (Category)

An *additive magma* in GAP is a domain A with an associative and commutative addition `+`: $A \times A \to A$, see `IsNearAdditiveMagma` (55.1.1) and `IsAdditivelyCommutative` (55.3.1).

55.1.5 IsAdditiveMagmaWithZero

▷ `IsAdditiveMagmaWithZero(`*obj*`)` (Category)

An *additive magma-with-zero* in GAP is an additive magma A (see `IsAdditiveMagma` (55.1.4) with an operation `0*` (or `Zero` (31.10.3)) that yields the zero of A.

So an additive magma-with-zero A does always contain a unique additively neutral element z, i.e., $z+a = a = a+z$ holds for all $a \in A$ (see `AdditiveNeutralElement` (55.3.5)). This element z can be computed with the operation `Zero` (31.10.3) as `Zero( A )`, and z is also equal to `Zero( a )` and to `0*`a for each element a in A.

Note that it may happen that an additive magma containing a zero does *not* lie in the category `IsAdditiveMagmaWithZero` (see 31.6).

55.1.6 IsAdditiveGroup

▷ `IsAdditiveGroup(`*obj*`)` (Category)
▷ `IsAdditiveMagmaWithInverses(`*obj*`)` (Category)

An *additive group* in GAP is an additive magma-with-zero A with an operation `-1*`: $A \to A$ that maps each element a of A to its additive inverse `-1*`a (or `AdditiveInverse( a )`, see `AdditiveInverse` (31.10.9)).

The addition `+` of A is assumed to be commutative and associative, so an additive group is not more than an *additive magma-with-inverses*. `IsAdditiveMagmaWithInverses` is just a synonym for `IsAdditiveGroup`, and can be used alternatively in all function names involving the string `"AdditiveGroup"`.

Note that not every trivial additive magma is an additive magma-with-zero, but every trivial additive magma-with-zero is an additive group.

55.2 (Near-)Additive Magma Generation

This section describes functions that create additive magmas from generators (see NearAdditiveMagma (55.2.1), NearAdditiveMagmaWithZero (55.2.2), NearAdditiveGroup (55.2.3)), the underlying operations for which methods can be installed (see NearAdditiveMagmaByGenerators (55.2.4), NearAdditiveMagmaWithZeroByGenerators (55.2.5), NearAdditiveGroupByGenerators (55.2.6)) and functions for forming additive submagmas (see SubnearAdditiveMagma (55.2.7), SubnearAdditiveMagmaWithZero (55.2.8), SubnearAdditiveGroup (55.2.9)).

55.2.1 NearAdditiveMagma

▷ NearAdditiveMagma([*Fam*,]*gens*) (function)

returns the (near-)additive magma *A* that is generated by the elements in the list *gens*, that is, the closure of *gens* under addition +. The family *Fam* of *A* can be entered as first argument; this is obligatory if *gens* is empty (and hence also *A* is empty).

55.2.2 NearAdditiveMagmaWithZero

▷ NearAdditiveMagmaWithZero([*Fam*,]*gens*) (function)

returns the (near-)additive magma-with-zero *A* that is generated by the elements in the list *gens*, that is, the closure of *gens* under addition + and Zero (31.10.3). The family *Fam* of *A* can be entered as first argument; this is obligatory if *gens* is empty (and hence *A* is trivial).

55.2.3 NearAdditiveGroup

▷ NearAdditiveGroup([*Fam*,]*gens*) (function)

returns the (near-)additive group *A* that is generated by the elements in the list *gens*, that is, the closure of *gens* under addition +, Zero (31.10.3), and AdditiveInverse (31.10.9). The family *Fam* of *A* can be entered as first argument; this is obligatory if *gens* is empty (and hence *A* is trivial).

55.2.4 NearAdditiveMagmaByGenerators

▷ NearAdditiveMagmaByGenerators([*Fam*,]*gens*) (operation)

An underlying operation for NearAdditiveMagma (55.2.1).

55.2.5 NearAdditiveMagmaWithZeroByGenerators

▷ NearAdditiveMagmaWithZeroByGenerators([*Fam*,]*gens*) (operation)

An underlying operation for NearAdditiveMagmaWithZero (55.2.2).

55.2.6 NearAdditiveGroupByGenerators

▷ NearAdditiveGroupByGenerators([Fam,]gens) (operation)

An underlying operation for NearAdditiveGroup (55.2.3).

55.2.7 SubnearAdditiveMagma

▷ SubnearAdditiveMagma(D, gens) (function)
▷ SubadditiveMagma(D, gens) (function)
▷ SubnearAdditiveMagmaNC(D, gens) (function)
▷ SubadditiveMagmaNC(D, gens) (function)

SubnearAdditiveMagma returns the near-additive magma generated by the elements in the list *gens*, with parent the domain *D*. SubnearAdditiveMagmaNC does the same, except that it does not check whether the elements of *gens* lie in *D*.

SubadditiveMagma and SubadditiveMagmaNC are just synonyms of these functions.

55.2.8 SubnearAdditiveMagmaWithZero

▷ SubnearAdditiveMagmaWithZero(D, gens) (function)
▷ SubadditiveMagmaWithZero(D, gens) (function)
▷ SubnearAdditiveMagmaWithZeroNC(D, gens) (function)
▷ SubadditiveMagmaWithZeroNC(D, gens) (function)

SubnearAdditiveMagmaWithZero returns the near-additive magma-with-zero generated by the elements in the list *gens*, with parent the domain *D*. SubnearAdditiveMagmaWithZeroNC does the same, except that it does not check whether the elements of *gens* lie in *D*.

SubadditiveMagmaWithZero and SubadditiveMagmaWithZeroNC are just synonyms of these functions.

55.2.9 SubnearAdditiveGroup

▷ SubnearAdditiveGroup(D, gens) (function)
▷ SubadditiveGroup(D, gens) (function)
▷ SubnearAdditiveGroupNC(D, gens) (function)
▷ SubadditiveGroupNC(D, gens) (function)

SubnearAdditiveGroup returns the near-additive group generated by the elements in the list *gens*, with parent the domain *D*. SubadditiveGroupNC does the same, except that it does not check whether the elements of *gens* lie in *D*.

SubadditiveGroup and SubadditiveGroupNC are just synonyms of these functions.

55.3 Attributes and Properties for (Near-)Additive Magmas

55.3.1 IsAdditivelyCommutative

▷ `IsAdditivelyCommutative(A)` (property)

A near-additive magma `A` in GAP is *additively commutative* if for all elements $a, b \in$ `A` the equality $a+b = b+a$ holds.

Note that the commutativity of the *multiplication* `*` in a multiplicative structure can be tested with `IsCommutative` (35.4.9).

55.3.2 GeneratorsOfNearAdditiveMagma

▷ `GeneratorsOfNearAdditiveMagma(A)` (attribute)
▷ `GeneratorsOfAdditiveMagma(A)` (attribute)

is a list of elements of the near-additive magma `A` that generates `A` as a near-additive magma, that is, the closure of this list under addition is `A`.

55.3.3 GeneratorsOfNearAdditiveMagmaWithZero

▷ `GeneratorsOfNearAdditiveMagmaWithZero(A)` (attribute)
▷ `GeneratorsOfAdditiveMagmaWithZero(A)` (attribute)

is a list of elements of the near-additive magma-with-zero `A` that generates `A` as a near-additive magma-with-zero, that is, the closure of this list under addition and `Zero` (31.10.3) is `A`.

55.3.4 GeneratorsOfNearAdditiveGroup

▷ `GeneratorsOfNearAdditiveGroup(A)` (attribute)
▷ `GeneratorsOfAdditiveGroup(A)` (attribute)

is a list of elements of the near-additive group `A` that generates `A` as a near-additive group, that is, the closure of this list under addition, taking the zero element, and taking additive inverses (see `AdditiveInverse` (31.10.9)) is `A`.

55.3.5 AdditiveNeutralElement

▷ `AdditiveNeutralElement(A)` (attribute)

returns the element z in the near-additive magma `A` with the property that $z+a = a = a+z$ holds for all $a \in$ `A`, if such an element exists. Otherwise `fail` is returned.

A near-additive magma that is not a near-additive magma-with-zero can have an additive neutral element z; in this case, z *cannot* be obtained as `Zero( A )` or as `0*`a for an element a in `A`, see `Zero` (31.10.3).

55.3.6 TrivialSubnearAdditiveMagmaWithZero

▷ `TrivialSubnearAdditiveMagmaWithZero(A)` (attribute)

is the additive magma-with-zero that has the zero of the near-additive magma-with-zero `A` as its only element.

55.4 Operations for (Near-)Additive Magmas

55.4.1 ClosureNearAdditiveGroup

▷ `ClosureNearAdditiveGroup(A, a)` (operation)
▷ `ClosureNearAdditiveGroup(A, B)` (operation)

returns the closure of the near-additive magma `A` with the element `a` or with the near-additive magma `B`, w.r.t. addition, taking the zero element, and taking additive inverses.

55.4.2 ShowAdditionTable

▷ `ShowAdditionTable(R)` (function)
▷ `ShowMultiplicationTable(M)` (function)

For a structure `R` with an addition given by `+`, respectively a structure `M` with a multiplication given by `*`, this command displays the addition (multiplication) table of the structure in a pretty way.

Example

```
gap> ShowAdditionTable(GF(4));
+        | 0*Z(2)   Z(2)^0   Z(2^2)   Z(2^2)^2
---------+------------------------------------
0*Z(2)   | 0*Z(2)   Z(2)^0   Z(2^2)   Z(2^2)^2
Z(2)^0   | Z(2)^0   0*Z(2)   Z(2^2)^2 Z(2^2)
Z(2^2)   | Z(2^2)   Z(2^2)^2 0*Z(2)   Z(2)^0
Z(2^2)^2 | Z(2^2)^2 Z(2^2)   Z(2)^0   0*Z(2)
```

Chapter 56

Rings

This chapter deals with domains that are additive groups (see IsAdditiveGroup (55.1.6) closed under multiplication *. Such a domain, if * and + are distributive, is called a *ring* in GAP. Each division ring, field (see 58), or algebra (see 62) is a ring. Important examples of rings are the integers (see 14) and matrix rings.

In the case of a *ring-with-one*, additional multiplicative structure is present, see IsRingWithOne (56.3.1). There is a little support in GAP for rings that have no additional structure: it is possible to perform some computations for small finite rings; infinite rings are handled by GAP in an acceptable way in the case that they are algebras.

Also, the SONATA package provides support for near-rings, and a related functionality for multiplicative semigroups of near-rings is available in the Smallsemi package.

Several functions for ring elements, such as IsPrime (56.5.8) and Factors (56.5.9), are defined only relative to a ring *R*, which can be entered as an optional argument; if *R* is omitted then a *default ring* is formed from the ring elements given as arguments, see DefaultRing (56.1.3).

56.1 Generating Rings

56.1.1 IsRing

▷ IsRing(*R*) (property)

A *ring* in GAP is an additive group (see IsAdditiveGroup (55.1.6)) that is also a magma (see IsMagma (35.1.1)), such that addition + and multiplication * are distributive, see IsDistributive (56.4.5).

The multiplication need *not* be associative (see IsAssociative (35.4.7)). For example, a Lie algebra (see 64) is regarded as a ring in GAP.

56.1.2 Ring

▷ Ring(*r*, *s*, ...) (function)
▷ Ring(*coll*) (function)

In the first form Ring returns the smallest ring that contains all the elements *r*, *s*, ... In the second form Ring returns the smallest ring that contains all the elements in the collection *coll*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

Ring differs from DefaultRing (56.1.3) in that it returns the smallest ring in which the elements lie, while DefaultRing (56.1.3) may return a larger ring if that makes sense.

Example

```
gap> Ring( 2, E(4) );
<ring with 2 generators>
```

56.1.3 DefaultRing

▷ DefaultRing(*r, s, ...*) (function)
▷ DefaultRing(*coll*) (function)

In the first form DefaultRing returns a ring that contains all the elements *r*, *s*, ... etc. In the second form DefaultRing returns a ring that contains all the elements in the collection *coll*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

The ring returned by DefaultRing need not be the smallest ring in which the elements lie. For example for elements from cyclotomic fields, DefaultRing may return the ring of integers of the smallest cyclotomic field in which the elements lie, which need not be the smallest ring overall, because the elements may in fact lie in a smaller number field which is itself not a cyclotomic field.

(For the exact definition of the default ring of a certain type of elements, look at the corresponding method installation.)

DefaultRing is used by ring functions such as Quotient (56.1.9), IsPrime (56.5.8), Factors (56.5.9), or Gcd (56.7.1) if no explicit ring is given.

Ring (56.1.2) differs from DefaultRing in that it returns the smallest ring in which the elements lie, while DefaultRing may return a larger ring if that makes sense.

Example

```
gap> DefaultRing( 2, E(4) );
GaussianIntegers
```

56.1.4 RingByGenerators

▷ RingByGenerators(*C*) (operation)

RingByGenerators returns the ring generated by the elements in the collection *C*, i. e., the closure of *C* under addition, multiplication, and taking additive inverses.

Example

```
gap> RingByGenerators([ 2, E(4) ]);
<ring with 2 generators>
```

56.1.5 DefaultRingByGenerators

▷ DefaultRingByGenerators(*coll*) (operation)

For a collection *coll*, returns a default ring in which *coll* is contained.

Example

```
gap> DefaultRingByGenerators([ 2, E(4) ]);
GaussianIntegers
```

56.1.6 GeneratorsOfRing

▷ GeneratorsOfRing(*R*) (attribute)

GeneratorsOfRing returns a list of elements such that the ring *R* is the closure of these elements under addition, multiplication, and taking additive inverses.

Example

```
gap> R:=Ring( 2, 1/2 );
<ring with 2 generators>
gap> GeneratorsOfRing( R );
[ 2, 1/2 ]
```

56.1.7 Subring

▷ Subring(*R, gens*) (function)
▷ SubringNC(*R, gens*) (function)

returns the ring with parent *R* generated by the elements in *gens*. When the second form, SubringNC is used, it is *not* checked whether all elements in *gens* lie in *R*.

Example

```
gap> R:= Integers;
Integers
gap> S:= Subring( R, [ 4, 6 ] );
<ring with 1 generators>
gap> Parent( S );
Integers
```

56.1.8 ClosureRing

▷ ClosureRing(*R, r*) (operation)
▷ ClosureRing(*R, S*) (operation)

For a ring *R* and either an element *r* of its elements family or a ring *S*, ClosureRing returns the ring generated by both arguments.

Example

```
gap> ClosureRing( Integers, E(4) );
<ring-with-one, with 2 generators>
```

56.1.9 Quotient

▷ Quotient(*[R,]r, s*) (operation)

Quotient returns the quotient of the two ring elements *r* and *s* in the ring *R*, if given, and otherwise in their default ring (see DefaultRing (56.1.3)). It returns fail if the quotient does not exist in the respective ring.

(To perform the division in the quotient field of a ring, use the quotient operator /.)

Example

```
gap> Quotient( 2, 3 );
fail
gap> Quotient( 6, 3 );
2
```

56.2 Ideals in Rings

A *left ideal* in a ring R is a subring of R that is closed under multiplication with elements of R from the left.

A *right ideal* in a ring R is a subring of R that is closed under multiplication with elements of R from the right.

A *two-sided ideal* or simply *ideal* in a ring R is both a left ideal and a right ideal in R.

So being a (left/right/two-sided) ideal is not a property of a domain but refers to the acting ring(s). Hence we must ask, e. g., `IsIdeal( `R,I `)` if we want to know whether the ring I is an ideal in the ring R. The property `IsTwoSidedIdealInParent` (56.2.3) can be used to store whether a ring is an ideal in its parent.

(Whenever the term `"Ideal"` occurs in an identifier without a specifying prefix `"Left"` or `"Right"`, this means the same as `"TwoSidedIdeal"`. Conversely, any occurrence of `"TwoSidedIdeal"` can be substituted by `"Ideal"`.)

For any of the above kinds of ideals, there is a notion of generators, namely `GeneratorsOfLeftIdeal` (56.2.8), `GeneratorsOfRightIdeal` (56.2.9), and `GeneratorsOfTwoSidedIdeal` (56.2.7). The acting rings can be accessed as `LeftActingRingOfIdeal` (56.2.10) and `RightActingRingOfIdeal` (56.2.10), respectively. Note that ideals are detected from known values of these attributes, especially it is assumed that whenever a domain has both a left and a right acting ring then these two are equal.

Note that we cannot use `LeftActingDomain` (57.1.11) and `RightActingDomain` here, since ideals in algebras are themselves vector spaces, and such a space can of course also be a module for an action from the right. In order to make the usual vector space functionality automatically available for ideals, we have to distinguish the left and right module structure from the additional closure properties of the ideal.

Further note that the attributes denoting ideal generators and acting ring are used to create ideals if this is explicitly wanted, but the ideal relation in the sense of `IsTwoSidedIdeal` (56.2.3) is of course independent of the presence of the attribute values.

Ideals are constructed with `LeftIdeal` (56.2.1), `RightIdeal` (56.2.1), `TwoSidedIdeal` (56.2.1). Principal ideals of the form $x*R$, $R*x$, $R*x*R$ can also be constructed with a simple multiplication.

Currently many methods for dealing with ideals need linear algebra to work, so they are mainly applicable to ideals in algebras.

56.2.1 TwoSidedIdeal

▷ `TwoSidedIdeal(`*R, gens[, "basis"]*`)` (function)
▷ `Ideal(`*R, gens[, "basis"]*`)` (function)
▷ `LeftIdeal(`*R, gens[, "basis"]*`)` (function)
▷ `RightIdeal(`*R, gens[, "basis"]*`)` (function)

Let *R* be a ring, and *gens* a list of collection of elements in *R*. TwoSidedIdeal, LeftIdeal, and RightIdeal return the two-sided, left, or right ideal, respectively, *I* in *R* that is generated by *gens*. The ring *R* can be accessed as LeftActingRingOfIdeal (56.2.10) or RightActingRingOfIdeal (56.2.10) (or both) of *I*.

If *R* is a left *F*-module then also *I* is a left *F*-module, in particular the LeftActingDomain (57.1.11) values of *R* and *I* are equal.

If the optional argument "basis" is given then *gens* are assumed to be a list of basis vectors of *I* viewed as a free *F*-module. (This is mainly applicable to ideals in algebras.) In this case, it is *not* checked whether *gens* really is linearly independent and whether *gens* is a subset of *R*.

Ideal is simply a synonym of TwoSidedIdeal.

Example

```
gap> R:= Integers;;
gap> I:= Ideal( R, [ 2 ] );
<two-sided ideal in Integers, (1 generators)>
```

56.2.2 TwoSidedIdealNC

▷ TwoSidedIdealNC(*R, gens[, "basis"]*) (function)
▷ IdealNC(*R, gens[, "basis"]*) (function)
▷ LeftIdealNC(*R, gens[, "basis"]*) (function)
▷ RightIdealNC(*R, gens[, "basis"]*) (function)

The effects of TwoSidedIdealNC, LeftIdealNC, and RightIdealNC are the same as TwoSidedIdeal (56.2.1), LeftIdeal (56.2.1), and RightIdeal (56.2.1), respectively, but they do not check whether all entries of *gens* lie in *R*.

56.2.3 IsTwoSidedIdeal

▷ IsTwoSidedIdeal(*R, I*) (operation)
▷ IsLeftIdeal(*R, I*) (operation)
▷ IsRightIdeal(*R, I*) (operation)
▷ IsTwoSidedIdealInParent(*I*) (property)
▷ IsLeftIdealInParent(*I*) (property)
▷ IsRightIdealInParent(*I*) (property)

The properties IsTwoSidedIdealInParent etc., are attributes of the ideal, and once known they are stored in the ideal.

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );
( Rationals^[ 3, 3 ] )
gap> I:= Ideal( A, [ Random( A ) ] );
<two-sided ideal in ( Rationals^[ 3, 3 ] ), (1 generators)>
gap> IsTwoSidedIdeal( A, I );
true
```

56.2.4 TwoSidedIdealByGenerators

▷ TwoSidedIdealByGenerators(*R, gens*) (operation)
▷ IdealByGenerators(*R, gens*) (operation)

TwoSidedIdealByGenerators returns the ring that is generated by the elements of the collection *gens* under addition, multiplication, and multiplication with elements of the ring *R* from the left and from the right.

R can be accessed by LeftActingRingOfIdeal (56.2.10) or RightActingRingOfIdeal (56.2.10), *gens* can be accessed by GeneratorsOfTwoSidedIdeal (56.2.7).

56.2.5 LeftIdealByGenerators

▷ LeftIdealByGenerators(*R, gens*) (operation)

LeftIdealByGenerators returns the ring that is generated by the elements of the collection *gens* under addition, multiplication, and multiplication with elements of the ring *R* from the left.

R can be accessed by LeftActingRingOfIdeal (56.2.10), *gens* can be accessed by GeneratorsOfLeftIdeal (56.2.8).

56.2.6 RightIdealByGenerators

▷ RightIdealByGenerators(*R, gens*) (operation)

RightIdealByGenerators returns the ring that is generated by the elements of the collection *gens* under addition, multiplication, and multiplication with elements of the ring *R* from the right.

R can be accessed by RightActingRingOfIdeal (56.2.10), *gens* can be accessed by GeneratorsOfRightIdeal (56.2.9).

56.2.7 GeneratorsOfTwoSidedIdeal

▷ GeneratorsOfTwoSidedIdeal(*I*) (attribute)
▷ GeneratorsOfIdeal(*I*) (attribute)

is a list of generators for the ideal *I*, with respect to the action of the rings that are stored as the values of LeftActingRingOfIdeal (56.2.10) and RightActingRingOfIdeal (56.2.10), from the left and from the right, respectively.

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> I:= Ideal( A, [ One( A ) ] );;
gap> GeneratorsOfIdeal( I );
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] ]
```

56.2.8 GeneratorsOfLeftIdeal

▷ GeneratorsOfLeftIdeal(*I*) (attribute)

is a list of generators for the left ideal *I*, with respect to the action from the left of the ring that is stored as the value of LeftActingRingOfIdeal (56.2.10).

56.2.9 GeneratorsOfRightIdeal

▷ `GeneratorsOfRightIdeal(I)` (attribute)

is a list of generators for the right ideal `I`, with respect to the action from the right of the ring that is stored as the value of `RightActingRingOfIdeal` (56.2.10).

56.2.10 LeftActingRingOfIdeal

▷ `LeftActingRingOfIdeal(I)` (attribute)
▷ `RightActingRingOfIdeal(I)` (attribute)

returns the left (resp. right) acting ring of an ideal `I`.

56.2.11 AsLeftIdeal

▷ `AsLeftIdeal(R, S)` (operation)
▷ `AsRightIdeal(R, S)` (operation)
▷ `AsTwoSidedIdeal(R, S)` (operation)

Let `S` be a subring of the ring `R`.

If `S` is a left ideal in `R` then `AsLeftIdeal` returns this left ideal, otherwise `fail` is returned.

If `S` is a right ideal in `R` then `AsRightIdeal` returns this right ideal, otherwise `fail` is returned.

If `S` is a two-sided ideal in `R` then `AsTwoSidedIdeal` returns this two-sided ideal, otherwise `fail` is returned.

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> B:= DirectSumOfAlgebras( A, A );
<algebra over Rationals, with 6 generators>
gap> C:= Subalgebra( B, Basis( B ){[1..9]} );
<algebra over Rationals, with 9 generators>
gap> I:= AsTwoSidedIdeal( B, C );
<two-sided ideal in <algebra of dimension 18 over Rationals>,
  (9 generators)>
```

56.3 Rings With One

56.3.1 IsRingWithOne

▷ `IsRingWithOne(R)` (property)

A *ring-with-one* in GAP is a ring (see `IsRing` (56.1.1)) that is also a magma-with-one (see `IsMagmaWithOne` (35.1.2)).

Note that the identity and the zero of a ring-with-one need *not* be distinct. This means that a ring that consists only of its zero element can be regarded as a ring-with-one.

This is especially useful in the case of finitely presented rings, in the sense that each factor of a ring-with-one is again a ring-with-one.

56.3.2 RingWithOne

▷ RingWithOne(*r, s, ...*) (function)
▷ RingWithOne(*coll*) (function)

In the first form RingWithOne returns the smallest ring with one that contains all the elements *r*, *s*, ... In the second form RingWithOne returns the smallest ring with one that contains all the elements in the collection *C*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

Example

```
gap> RingWithOne( [ 4, 6 ] );
Integers
```

56.3.3 RingWithOneByGenerators

▷ RingWithOneByGenerators(*coll*) (operation)

RingWithOneByGenerators returns the ring-with-one generated by the elements in the collection *coll*, i. e., the closure of *coll* under addition, multiplication, taking additive inverses, and taking the identity of an element.

56.3.4 GeneratorsOfRingWithOne

▷ GeneratorsOfRingWithOne(*R*) (attribute)

GeneratorsOfRingWithOne returns a list of elements such that the ring *R* is the closure of these elements under addition, multiplication, taking additive inverses, and taking the identity element One(*R*).

R itself need *not* be known to be a ring-with-one.

Example

```
gap> R:= RingWithOne( [ 4, 6 ] );
Integers
gap> GeneratorsOfRingWithOne( R );
[ 1 ]
```

56.3.5 SubringWithOne

▷ SubringWithOne(*R, gens*) (function)
▷ SubringWithOneNC(*R, gens*) (function)

returns the ring with one with parent *R* generated by the elements in *gens*. When the second form, SubringWithOneNC is used, it is *not* checked whether all elements in *gens* lie in *R*.

Example

```
gap> R:= SubringWithOne( Integers, [ 4, 6 ] );
Integers
gap> Parent( R );
Integers
```

56.4 Properties of Rings

56.4.1 IsIntegralRing

▷ `IsIntegralRing(R)` (property)

A ring-with-one `R` is integral if it is commutative, contains no nontrivial zero divisors, and if its identity is distinct from its zero.

Example

```
gap> IsIntegralRing( Integers );
true
```

56.4.2 IsUniqueFactorizationRing

▷ `IsUniqueFactorizationRing(R)` (Category)

A ring `R` is called a *unique factorization ring* if it is an integral ring (see `IsIntegralRing` (56.4.1)), and every nonzero element has a unique factorization into irreducible elements, i.e., a unique representation as product of irreducibles (see `IsIrreducibleRingElement` (56.5.7)). Unique in this context means unique up to permutations of the factors and up to multiplication of the factors by units (see `Units` (56.5.2)).

Mathematically, a field should therefore also be a unique factorization ring, since every nonzero element is a unit. In GAP, however, at least at present fields do not lie in the filter `IsUniqueFactorizationRing`, since operations such as `Factors` (56.5.9), `Gcd` (56.7.1), `StandardAssociate` (56.5.5) and so on do not apply to fields (the results would be trivial, and not especially useful) and methods which require their arguments to lie in `IsUniqueFactorizationRing` expect these operations to work.

(Note that we cannot install a subset maintained method for this filter since the factorization of an element needs not exist in a subring. As an example, consider the subring $4\mathbb{N}+1$ of the ring $4\mathbb{Z}+1$; in the subring, the element $3\cdot 3\cdot 11\cdot 7$ has the two factorizations $33\cdot 21 = 9\cdot 77$, but in the large ring there is the unique factorization $(-3)\cdot(-3)\cdot(-11)\cdot(-7)$, and it is easy to see that every element in $4\mathbb{Z}+1$ has a unique factorization.)

Example

```
gap> IsUniqueFactorizationRing( PolynomialRing( Rationals, 1 ) );
true
```

56.4.3 IsLDistributive

▷ `IsLDistributive(C)` (property)

is `true` if the relation $a*(b+c) = (a*b)+(a*c)$ holds for all elements a, b, c in the collection `C`, and `false` otherwise.

56.4.4 IsRDistributive

▷ `IsRDistributive(C)` (property)

is true if the relation $(a+b)*c = (a*c)+(b*c)$ holds for all elements a, b, c in the collection C, and false otherwise.

56.4.5 IsDistributive

▷ IsDistributive(C) (property)

is true if the collection C is both left and right distributive (see IsLDistributive (56.4.3), IsRDistributive (56.4.4)), and false otherwise.

Example

```
gap> IsDistributive( Integers );
true
```

56.4.6 IsAnticommutative

▷ IsAnticommutative(R) (property)

is true if the relation $a*b = -b*a$ holds for all elements a, b in the ring R, and false otherwise.

56.4.7 IsZeroSquaredRing

▷ IsZeroSquaredRing(R) (property)

is true if $a*a$ is the zero element of the ring R for all a in R, and false otherwise.

56.4.8 IsJacobianRing

▷ IsJacobianRing(R) (property)

is true if the Jacobi identity holds in the ring R, and false otherwise. The Jacobi identity means that $x*(y*z)+z*(x*y)+y*(z*x)$ is the zero element of R, for all elements x, y, z in R.

Example

```
gap> L:= FullMatrixLieAlgebra( GF( 5 ), 7 );
<Lie algebra over GF(5), with 13 generators>
gap> IsJacobianRing( L );
true
```

56.5 Units and Factorizations

56.5.1 IsUnit

▷ IsUnit([R,]r) (operation)

IsUnit returns true if r is a unit in the ring R, if given, and otherwise in its default ring (see DefaultRing (56.1.3)). If r is not a unit then false is returned.

An element r is called a *unit* in a ring R, if r has an inverse in R.

IsUnit may call Quotient (56.1.9).

56.5.2 Units

▷ `Units(R)` (attribute)

`Units` returns the group of units of the ring `R`. This may either be returned as a list or as a group.

An element r is called a *unit* of a ring R if r has an inverse in R. It is easy to see that the set of units forms a multiplicative group.

Example

```
gap> Units( GaussianIntegers );
[ -1, 1, -E(4), E(4) ]
gap> Units( GF( 16 ) );
<group with 1 generators>
```

56.5.3 IsAssociated

▷ `IsAssociated([R, ]r, s)` (operation)

`IsAssociated` returns `true` if the two ring elements `r` and `s` are associated in the ring `R`, if given, and otherwise in their default ring (see `DefaultRing` (56.1.3)). If the two elements are not associated then `false` is returned.

Two elements `r` and `s` of a ring `R` are called *associated* if there is a unit u of `R` such that $r\,u = s$.

56.5.4 Associates

▷ `Associates([R, ]r)` (operation)

`Associates` returns the set of associates of `r` in the ring `R`, if given, and otherwise in its default ring (see `DefaultRing` (56.1.3)).

Two elements `r` and s of a ring R are called *associated* if there is a unit u of R such that $ru = s$.

Example

```
gap> Associates( Integers, 2 );
[ -2, 2 ]
gap> Associates( GaussianIntegers, 2 );
[ -2, 2, -2*E(4), 2*E(4) ]
```

56.5.5 StandardAssociate

▷ `StandardAssociate([R, ]r)` (operation)

`StandardAssociate` returns the standard associate of the ring element `r` in the ring `R`, if given, and otherwise in its default ring (see `DefaultRing` (56.1.3)).

The *standard associate* of a ring element `r` of `R` is an associated element of `r` which is, in a ring dependent way, distinguished among the set of associates of `r`. For example, in the ring of integers the standard associate is the absolute value.

Example

```
gap> x:= Indeterminate( Rationals, "x" );;
gap> StandardAssociate( -x^2-x+1 );
x^2+x-1
```

56.5.6 StandardAssociateUnit

▷ StandardAssociateUnit([R,]r) (operation)

StandardAssociateUnit returns a unit in the ring R such that the ring element r times this unit equals the standard associate of r in R.

If R is not given, the default ring of r is used instead. (see DefaultRing (56.1.3)).

Example

```
gap> y:= Indeterminate( Rationals, "y" );;
gap> r:= -y^2-y+1;
-y^2-y+1
gap> StandardAssociateUnit( r );
-1
gap> StandardAssociateUnit( r ) * r = StandardAssociate( r );
true
```

56.5.7 IsIrreducibleRingElement

▷ IsIrreducibleRingElement([R,]r) (operation)

IsIrreducibleRingElement returns true if the ring element r is irreducible in the ring R, if given, and otherwise in its default ring (see DefaultRing (56.1.3)). If r is not irreducible then false is returned.

An element r of a ring R is called *irreducible* if r is not a unit in R and if there is no nontrivial factorization of r in R, i.e., if there is no representation of r as product *st* such that neither *s* nor *t* is a unit (see IsUnit (56.5.1)). Each prime element (see IsPrime (56.5.8)) is irreducible.

Example

```
gap> IsIrreducibleRingElement( Integers, 2 );
true
```

56.5.8 IsPrime

▷ IsPrime([R,]r) (operation)

IsPrime returns true if the ring element r is a prime in the ring R, if given, and otherwise in its default ring (see DefaultRing (56.1.3)). If r is not a prime then false is returned.

An element r of a ring R is called *prime* if for each pair *s* and *t* such that r divides *st* the element r divides either *s* or *t*. Note that there are rings where not every irreducible element (see IsIrreducibleRingElement (56.5.7)) is a prime.

56.5.9 Factors

▷ Factors([R,]r) (operation)

Factors returns the factorization of the ring element r in the ring R, if given, and otherwise in in its default ring (see DefaultRing (56.1.3)). The factorization is returned as a list of primes (see IsPrime (56.5.8)). Each element in the list is a standard associate (see StandardAssociate (56.5.5)) except the first one, which is multiplied by a unit as necessary to have Product(Factors(

`R, r ) ) = r`. This list is usually also sorted, thus smallest prime factors come first. If `r` is a unit or zero, `Factors( R, r ) = [ r ]`.

Example

```
gap> x:= Indeterminate( GF(2), "x" );;
gap> pol:= x^2+x+1;
x^2+x+Z(2)^0
gap> Factors( pol );
[ x^2+x+Z(2)^0 ]
gap> Factors( PolynomialRing( GF(4) ), pol );
[ x+Z(2^2), x+Z(2^2)^2 ]
```

56.5.10 PadicValuation

▷ `PadicValuation(r, p)` (operation)

`PadicValuation` is the operation to compute the p-adic valuation of a ring element `r`.

56.6 Euclidean Rings

56.6.1 IsEuclideanRing

▷ `IsEuclideanRing(R)` (Category)

A ring R is called a Euclidean ring if it is an integral ring and there exists a function δ, called the Euclidean degree, from $R-\{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R-\{0_R\}$ there exists an element q such that either $r-qs=0_R$ or $\delta(r-qs)<\delta(s)$. In GAP the Euclidean degree δ is implicitly built into a ring and cannot be changed. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisor of two elements, which in turn implies that R is a unique factorization ring.

Example

```
gap> IsEuclideanRing( GaussianIntegers );
true
```

56.6.2 EuclideanDegree

▷ `EuclideanDegree([R, ]r)` (operation)

`EuclideanDegree` returns the Euclidean degree of the ring element `r` in the ring `R`, if given, and otherwise in its default ring (see `DefaultRing` (56.1.3)).

The ring `R` must be a Euclidean ring (see `IsEuclideanRing` (56.6.1)).

Example

```
gap> EuclideanDegree( GaussianIntegers, 3 );
9
```

56.6.3 EuclideanQuotient

▷ EuclideanQuotient([R,]r, m) (operation)

EuclideanQuotient returns the Euclidean quotient of the ring elements *r* and *m* in the ring *R*, if given, and otherwise in their default ring (see DefaultRing (56.1.3)).

The ring *R* must be a Euclidean ring (see IsEuclideanRing (56.6.1)), otherwise an error is signalled.

Example

```
gap> EuclideanQuotient( 8, 3 );
2
```

56.6.4 EuclideanRemainder

▷ EuclideanRemainder([R,]r, m) (operation)

EuclideanRemainder returns the Euclidean remainder of the ring element *r* modulo the ring element *m* in the ring *R*, if given, and otherwise in their default ring (see DefaultRing (56.1.3)).

The ring *R* must be a Euclidean ring (see IsEuclideanRing (56.6.1)), otherwise an error is signalled.

Example

```
gap> EuclideanRemainder( 8, 3 );
2
```

56.6.5 QuotientRemainder

▷ QuotientRemainder([R,]r, m) (operation)

QuotientRemainder returns the Euclidean quotient and the Euclidean remainder of the ring elements *r* and *m* in the ring *R*, if given, and otherwise in their default ring (see DefaultRing (56.1.3)). The result is a pair of ring elements.

The ring *R* must be a Euclidean ring (see IsEuclideanRing (56.6.1)), otherwise an error is signalled.

Example

```
gap> QuotientRemainder( GaussianIntegers, 8, 3 );
[ 3, -1 ]
```

56.7 Gcd and Lcm

56.7.1 Gcd

▷ Gcd([R,]r1, r2, ...) (function)
▷ Gcd([R,]list) (function)

Gcd returns the greatest common divisor of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in the ring *R*, if given, and otherwise in their default ring, see DefaultRing (56.1.3).

Gcd returns the standard associate (see StandardAssociate (56.5.5)) of the greatest common divisors.

A divisor of an element r in the ring R is an element $d \in R$ such that r is a multiple of d. A common divisor of the elements $r_1, r_2, \ldots$ in the ring R is an element $d \in R$ which is a divisor of each $r_1, r_2, \ldots$. A greatest common divisor d in addition has the property that every other common divisor of $r_1, r_2, \ldots$ is a divisor of d.

Note that this in particular implies the following: For the zero element z of `R`, we have `Gcd( r, z ) = Gcd( z, r ) = StandardAssociate( r )` and `Gcd( z, z ) = z`.

Example

```
gap> Gcd( Integers, [ 10, 15 ] );
5
```

56.7.2 GcdOp

▷ `GcdOp(`*`[R, ]r, s`*`)` (operation)

`GcdOp` is the operation to compute the greatest common divisor of two ring elements `r`, `s` in the ring `R` or in their default ring.

56.7.3 GcdRepresentation

▷ `GcdRepresentation(`*`[R, ]r1, r2, ...`*`)` (function)
▷ `GcdRepresentation(`*`[R, ]list`*`)` (function)

`GcdRepresentation` returns a representation of the greatest common divisor of the ring elements `r1`, `r2`, ... resp. of the ring elements in the list `list` in the Euclidean ring `R`, if given, and otherwise in their default ring, see `DefaultRing` (56.1.3).

A representation of the gcd g of the elements $r_1, r_2, \ldots$ of a ring R is a list of ring elements $s_1, s_2, \ldots$ of R, such that $g = s_1 r_1 + s_2 r_2 + \cdots$. Such representations do not exist in all rings, but they do exist in Euclidean rings (see `IsEuclideanRing` (56.6.1)), which can be shown using the Euclidean algorithm, which in fact can compute those coefficients.

Example

```
gap> a:= Indeterminate( Rationals, "a" );;
gap> GcdRepresentation( a^2+1, a^3+1 );
[ -1/2*a^2-1/2*a+1/2, 1/2*a+1/2 ]
```

`Gcdex` (14.3.5) provides similar functionality over the integers.

56.7.4 GcdRepresentationOp

▷ `GcdRepresentationOp(`*`[R, ]r, s`*`)` (operation)

`GcdRepresentationOp` is the operation to compute the representation of the greatest common divisor of two ring elements `r`, `s` in the Euclidean ring `R` or in their default ring, respectively.

56.7.5 ShowGcd

▷ `ShowGcd(a, b)` (function)

This function takes two elements `a` and `b` of an Euclidean ring and returns their greatest common divisor. It will print out the steps performed by the Euclidean algorithm, as well as the rearrangement of these steps to express the gcd as a ring combination of `a` and `b`.

Example

```
gap> ShowGcd(192,42);
192=4*42 + 24
42=1*24 + 18
24=1*18 + 6
18=3*6 + 0
The Gcd is 6
 = 1*24 -1*18
 = -1*42 + 2*24
 = 2*192 -9*42
6
```

56.7.6 Lcm

▷ `Lcm(`*`[R, ]r1, r2, ...`*`)` (function)
▷ `Lcm(`*`[R, ]list`*`)` (function)

`Lcm` returns the least common multiple of the ring elements *`r1`*, *`r2`*, ... resp. of the ring elements in the list *`list`* in the ring *`R`*, if given, and otherwise in their default ring, see `DefaultRing` (56.1.3).

`Lcm` returns the standard associate (see `StandardAssociate` (56.5.5)) of the least common multiples.

A least common multiple of the elements $r_1, r_2, \ldots$ of the ring R is an element m that is a multiple of $r_1, r_2, \ldots$, and every other multiple of these elements is a multiple of m.

Note that this in particular implies the following: For the zero element z of *`R`*, we have `Lcm( `*`r`*`, `*`z`*` ) = Lcm( `*`z`*`, `*`r`*` ) = StandardAssociate( `*`r`*` )` and `Lcm( `*`z`*`, `*`z`*` ) = `*`z`*.

56.7.7 LcmOp

▷ `LcmOp(`*`[R, ]r, s`*`)` (operation)

`LcmOp` is the operation to compute the least common multiple of two ring elements *`r`*, *`s`* in the ring *`R`* or in their default ring, respectively.

The default methods for this uses the equality $lcm(m,n) = m*n/gcd(m,n)$ (see `GcdOp` (56.7.2)).

56.7.8 QuotientMod

▷ `QuotientMod(`*`[R, ]r, s, m`*`)` (operation)

`QuotientMod` returns the quotient of the ring elements *`r`* and *`s`* modulo the ring element *`m`* in the ring *`R`*, if given, and otherwise in their default ring, see `DefaultRing` (56.1.3).

`R` must be a Euclidean ring (see `IsEuclideanRing` (56.6.1)) so that `EuclideanRemainder` (56.6.4) can be applied. If the modular quotient does not exist (i.e. when *`s`* and *`m`* are not coprime), `fail` is returned.

The quotient q of *`r`* and *`s`* modulo *`m`* is an element of *`R`* such that $qs = r$ modulo m, i.e., such that $qs - r$ is divisible by *`m`* in *`R`* and that q is either zero (if *`r`* is divisible by *`m`*) or the Euclidean degree of q is strictly smaller than the Euclidean degree of *`m`*.

Example

```
gap> QuotientMod( 7, 2, 3 );
2
```

56.7.9 PowerMod

▷ PowerMod([R,]r, e, m) (operation)

PowerMod returns the e-th power of the ring element r modulo the ring element m in the ring R, if given, and otherwise in their default ring, see DefaultRing (56.1.3). e must be an integer.

R must be a Euclidean ring (see IsEuclideanRing (56.6.1)) so that EuclideanRemainder (56.6.4) can be applied to its elements.

If e is positive the result is r^e modulo m. If e is negative then PowerMod first tries to find the inverse of r modulo m, i.e., i such that $ir = 1$ modulo m. If the inverse does not exist an error is signalled. If the inverse does exist PowerMod returns PowerMod(R, i, -e, m).

PowerMod reduces the intermediate values modulo m, improving performance drastically when e is large and m small.

Example

```
gap> PowerMod( 12, 100000, 7 );
2
```

56.7.10 InterpolatedPolynomial

▷ InterpolatedPolynomial(R, x, y) (operation)

InterpolatedPolynomial returns, for given lists x, y of elements in a ring R of the same length n, say, the unique polynomial of degree less than n which has value $y[i]$ at $x[i]$, for all $i \in \{1,\ldots,n\}$. Note that the elements in x must be distinct.

Example

```
gap> InterpolatedPolynomial( Integers, [ 1, 2, 3 ], [ 5, 7, 0 ] );
-9/2*x^2+31/2*x-6
```

56.8 Homomorphisms of Rings

A *ring homomorphism* is a mapping between two rings that respects addition and multiplication.

Currently GAP supports ring homomorphisms between finite rings (using straightforward methods) and ring homomorphisms with additional structures, where source and range are in fact algebras and where also the linear structure is respected, see 62.10.

56.8.1 RingGeneralMappingByImages

▷ RingGeneralMappingByImages(R, S, gens, imgs) (operation)

is a general mapping from the ring A to the ring S. This general mapping is defined by mapping the entries in the list gens (elements of R) to the entries in the list imgs (elements of S), and taking the additive and multiplicative closure.

gens need not generate R as a ring, and if the specification does not define an additive and multiplicative mapping then the result will be multivalued. Hence, in general it is not a mapping.

56.8.2 RingHomomorphismByImages

▷ RingHomomorphismByImages(*R*, *S*, *gens*, *imgs*) (function)

RingHomomorphismByImages returns the ring homomorphism with source *R* and range *S* that is defined by mapping the list *gens* of generators of *R* to the list *imgs* of images in *S*.

If *gens* does not generate *R* or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then fail is returned.

One can avoid the checks by calling RingHomomorphismByImagesNC (56.8.3), and one can construct multi-valued mappings with RingGeneralMappingByImages (56.8.1).

56.8.3 RingHomomorphismByImagesNC

▷ RingHomomorphismByImagesNC(*R*, *S*, *gens*, *imgs*) (operation)

RingHomomorphismByImagesNC is the operation that is called by the function RingHomomorphismByImages (56.8.2). Its methods may assume that *gens* generates *R* as a ring and that the mapping of *gens* to *imgs* defines a ring homomorphism. Results are unpredictable if these conditions do not hold.

For creating a possibly multi-valued mapping from *R* to *S* that respects addition and multiplication, RingGeneralMappingByImages (56.8.1) can be used.

56.8.4 NaturalHomomorphismByIdeal

▷ NaturalHomomorphismByIdeal(*R*, *I*) (operation)

is the homomorphism of rings provided by the natural projection map of *R* onto the quotient ring *R*/*I*. This map can be used to take pre-images in the original ring from elements in the quotient.

Chapter 57

Modules

57.1 Generating modules

57.1.1 IsLeftOperatorAdditiveGroup

▷ `IsLeftOperatorAdditiveGroup(D)` (Category)

A domain `D` lies in `IsLeftOperatorAdditiveGroup` if it is an additive group that is closed under scalar multiplication from the left, and such that $\lambda * (x+y) = \lambda * x + \lambda * y$ for all scalars λ and elements $x, y \in D$ (here and below by scalars we mean elements of a domain acting on `D` from left or right as appropriate).

57.1.2 IsLeftModule

▷ `IsLeftModule(M)` (Category)

A domain `M` lies in `IsLeftModule` if it lies in `IsLeftOperatorAdditiveGroup`, *and* the set of scalars forms a ring, *and* $(\lambda+\mu)*x = \lambda*x+\mu*x$ for scalars λ, μ and $x \in M$, *and* scalar multiplication satisfies $\lambda * (\mu * x) = (\lambda * \mu) * x$ for scalars λ, μ and $x \in M$.

Example

```
gap> V:= FullRowSpace( Rationals, 3 );
( Rationals^3 )
gap> IsLeftModule( V );
true
```

57.1.3 GeneratorsOfLeftOperatorAdditiveGroup

▷ `GeneratorsOfLeftOperatorAdditiveGroup(D)` (attribute)

returns a list of elements of `D` that generates `D` as a left operator additive group.

57.1.4 GeneratorsOfLeftModule

▷ `GeneratorsOfLeftModule(M)` (attribute)

returns a list of elements of `M` that generate `M` as a left module.

Example

```
gap> V:= FullRowSpace( Rationals, 3 );;
gap> GeneratorsOfLeftModule( V );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
```

57.1.5 AsLeftModule

▷ AsLeftModule(*R*, *D*) (operation)

if the domain *D* forms an additive group and is closed under left multiplication by the elements of *R*, then AsLeftModule(*R*, *D*) returns the domain *D* viewed as a left module.

Example

```
gap> coll:= [[0*Z(2),0*Z(2)], [Z(2),0*Z(2)], [0*Z(2),Z(2)], [Z(2),Z(2)]];
[ [ 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, Z(2)^0 ] ]
gap> AsLeftModule( GF(2), coll );
<vector space of dimension 2 over GF(2)>
```

57.1.6 IsRightOperatorAdditiveGroup

▷ IsRightOperatorAdditiveGroup(*D*) (Category)

A domain *D* lies in IsRightOperatorAdditiveGroup if it is an additive group that is closed under scalar multiplication from the right, and such that $(x+y)*\lambda = x*\lambda + y*\lambda$ for all scalars λ and elements $x, y \in D$.

57.1.7 IsRightModule

▷ IsRightModule(*M*) (Category)

A domain *M* lies in IsRightModule if it lies in IsRightOperatorAdditiveGroup, *and* the set of scalars forms a ring, *and* $x*(\lambda+\mu) = x*\lambda + x*\mu$ for scalars λ, μ and $x \in M$, *and* scalar multiplication satisfies $(x*\mu)*\lambda = x*(\mu*\lambda)$ for scalars λ, μ and $x \in M$.

57.1.8 GeneratorsOfRightOperatorAdditiveGroup

▷ GeneratorsOfRightOperatorAdditiveGroup(*D*) (attribute)

returns a list of elements of *D* that generates *D* as a right operator additive group.

57.1.9 GeneratorsOfRightModule

▷ GeneratorsOfRightModule(*M*) (attribute)

returns a list of elements of *M* that generate *M* as a left module.

57.1.10 LeftModuleByGenerators

▷ LeftModuleByGenerators(*R, gens[, zero]*) (operation)

returns the left module over *R* generated by *gens*.

Example

```
gap> coll:= [ [Z(2),0*Z(2)], [0*Z(2),Z(2)], [Z(2),Z(2)] ];;
gap> V:= LeftModuleByGenerators( GF(16), coll );
<vector space over GF(2^4), with 3 generators>
```

57.1.11 LeftActingDomain

▷ LeftActingDomain(*D*) (attribute)

Let *D* be an external left set, that is, *D* is closed under the action of a domain *L* by multiplication from the left. Then *L* can be accessed as value of LeftActingDomain for *D*.

57.2 Submodules

57.2.1 Submodule

▷ Submodule(*M, gens[, "basis"]*) (function)

is the left module generated by the collection *gens*, with parent module *M*. If the string "basis" is entered as the third argument then the submodule of *M* is created for which the list *gens* is known to be a list of basis vectors; in this case, it is *not* checked whether *gens* really is linearly independent and whether all in *gens* lie in *M*.

Example

```
gap> coll:= [ [Z(2),0*Z(2)], [0*Z(2),Z(2)], [Z(2),Z(2)] ];;
gap> V:= LeftModuleByGenerators( GF(16), coll );;
gap> W:= Submodule( V, [ coll[1], coll[2] ] );
<vector space over GF(2^4), with 2 generators>
gap> Parent( W ) = V;
true
```

57.2.2 SubmoduleNC

▷ SubmoduleNC(*M, gens[, "basis"]*) (function)

SubmoduleNC does the same as Submodule (57.2.1), except that it does not check whether all in *gens* lie in *M*.

57.2.3 ClosureLeftModule

▷ ClosureLeftModule(*M, m*) (operation)

is the left module generated by the left module generators of *M* and the element *m*.

Example
```
gap> V:= LeftModuleByGenerators(Rationals, [ [ 1, 0, 0 ], [ 0, 1, 0 ] ]);
<vector space over Rationals, with 2 generators>
gap> ClosureLeftModule( V, [ 1, 1, 1 ] );
<vector space over Rationals, with 3 generators>
```

57.2.4 TrivialSubmodule

▷ `TrivialSubmodule(`*M*`)` (attribute)

returns the zero submodule of *M*.

Example
```
gap> V:= LeftModuleByGenerators(Rationals, [[ 1, 0, 0 ], [ 0, 1, 0 ]]);;
gap> TrivialSubmodule( V );
<vector space over Rationals, with 0 generators>
```

57.3 Free Modules

57.3.1 IsFreeLeftModule

▷ `IsFreeLeftModule(`*M*`)` (Category)

A left module is free as module if it is isomorphic to a direct sum of copies of its left acting domain.

Free left modules can have bases.

The characteristic (see `Characteristic` (31.10.1)) of a free left module is defined as the characteristic of its left acting domain (see `LeftActingDomain` (57.1.11)).

57.3.2 FreeLeftModule

▷ `FreeLeftModule(`*R, gens[, zero][, "basis"]*`)` (function)

`FreeLeftModule(` *R*, *gens* `)` is the free left module over the ring *R*, generated by the vectors in the collection *gens*.

If there are three arguments, a ring *R* and a collection *gens* and an element *zero*, then `FreeLeftModule(` *R*, *gens*, *zero* `)` is the *R*-free left module generated by *gens*, with zero element *zero*.

If the last argument is the string `"basis"` then the vectors in *gens* are known to form a basis of the free module.

It should be noted that the generators *gens* must be vectors, that is, they must support an addition and a scalar action of *R* via left multiplication. (See also Section 31.3 for the general meaning of "generators" in GAP.) In particular, `FreeLeftModule` is *not* an equivalent of commands such as `FreeGroup` (37.2.1) in the sense of a constructor of a free group on abstract generators. Such a construction seems to be unnecessary for vector spaces, for that one can use for example row spaces (see `FullRowSpace` (61.9.4)) in the finite dimensional case and polynomial rings (see `PolynomialRing` (66.15.1)) in the infinite dimensional case. Moreover, the definition of a "natural" addition for elements of a given magma (for example a permutation group) is possible via the construction of magma rings (see Chapter 65).

Example
```
gap> V:= FreeLeftModule(Rationals, [[ 1, 0, 0 ], [ 0, 1, 0 ]], "basis");
<vector space of dimension 2 over Rationals>
```

57.3.3 Dimension

▷ `Dimension(`*M*`)` (attribute)

A free left module has dimension *n* if it is isomorphic to a direct sum of *n* copies of its left acting domain.

(We do *not* mark `Dimension` as invariant under isomorphisms since we want to call `UseIsomorphismRelation` (31.13.3) also for free left modules over different left acting domains.)

Example
```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ] );;
gap> Dimension( V );
2
```

57.3.4 IsFiniteDimensional

▷ `IsFiniteDimensional(`*M*`)` (property)

is `true` if *M* is a free left module that is finite dimensional over its left acting domain, and `false` otherwise.

Example
```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ] );;
gap> IsFiniteDimensional( V );
true
```

57.3.5 UseBasis

▷ `UseBasis(`*V, gens*`)` (operation)

The vectors in the list *gens* are known to form a basis of the free left module *V*. `UseBasis` stores information in *V* that can be derived form this fact, namely

- *gens* are stored as left module generators if no such generators were bound (this is useful especially if *V* is an algebra),
- the dimension of *V* is stored.

Example
```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ] );;
gap> UseBasis( V, [ [ 1, 0 ], [ 1, 1 ] ] );
gap> V;  # now V knows its dimension
<vector space of dimension 2 over Rationals>
```

57.3.6 IsRowModule

▷ `IsRowModule(`*V*`)` (property)

A *row module* is a free left module whose elements are row vectors.

57.3.7 IsMatrixModule

▷ IsMatrixModule(*V*) (property)

A *matrix module* is a free left module whose elements are matrices.

57.3.8 IsFullRowModule

▷ IsFullRowModule(*M*) (property)

A *full row module* is a module R^n, for a ring R and a nonnegative integer n.

More precisely, a full row module is a free left module over a ring R such that the elements are row vectors of the same length n and with entries in R and such that the dimension is equal to n.

Several functions delegate their tasks to full row modules, for example `Iterator` (30.8.1) and `Enumerator` (30.3.2).

57.3.9 FullRowModule

▷ FullRowModule(*R, n*) (function)

is the row module `R^n`, for a ring `R` and a nonnegative integer `n`.

Example

```
gap> V:= FullRowModule( Integers, 5 );
( Integers^5 )
```

57.3.10 IsFullMatrixModule

▷ IsFullMatrixModule(*M*) (property)

A *full matrix module* is a module $R^{[m,n]}$, for a ring R and two nonnegative integers m, n.

More precisely, a full matrix module is a free left module over a ring R such that the elements are m by n matrices with entries in R and such that the dimension is equal to mn.

57.3.11 FullMatrixModule

▷ FullMatrixModule(*R, m, n*) (function)

is the matrix module `R^[m,n]`, for a ring `R` and nonnegative integers `m` and `n`.

Example

```
gap> FullMatrixModule( GaussianIntegers, 3, 6 );
( GaussianIntegers^[ 3, 6 ] )
```

Chapter 58

Fields and Division Rings

A *division ring* is a ring (see Chapter 56) in which every non-zero element has an inverse. The most important class of division rings are the commutative ones, which are called *fields*.

GAP supports finite fields (see Chapter 59) and abelian number fields (see Chapter 60), in particular the field of rationals (see Chapter 17).

This chapter describes the general GAP functions for fields and division rings.

If a field `F` is a subfield of a commutative ring `C`, `C` can be considered as a vector space over the (left) acting domain `F` (see Chapter 61). In this situation, we call `F` the *field of definition* of `C`.

Each field in GAP is represented as a vector space over a subfield (see `IsField` (58.1.2)), thus each field is in fact a field extension in a natural way, which is used by functions such as `Norm` (58.3.4) and `Trace` (58.3.5) (see 58.3).

58.1 Generating Fields

58.1.1 IsDivisionRing

▷ `IsDivisionRing(D)` (Category)

A *division ring* in GAP is a nontrivial associative algebra `D` with a multiplicative inverse for each nonzero element. In GAP every division ring is a vector space over a division ring (possibly over itself). Note that being a division ring is thus not a property that a ring can get, because a ring is usually not represented as a vector space.

The field of coefficients is stored as the value of the attribute `LeftActingDomain` (57.1.11) of `D`.

58.1.2 IsField

▷ `IsField(D)` (property)

A *field* is a commutative division ring (see `IsDivisionRing` (58.1.1) and `IsCommutative` (35.4.9)).

Example

```
gap> IsField( GaloisField(16) );           # the field with 16 elements
true
gap> IsField( Rationals );                 # the field of rationals
true
gap> q:= QuaternionAlgebra( Rationals );;  # noncommutative division ring
```

```
gap> IsField( q );  IsDivisionRing( q );
false
true
gap> mat:= [ [ 1 ] ];;  a:= Algebra( Rationals, [ mat ] );;
gap> IsDivisionRing( a );   # algebra not constructed as a division ring
false
```

58.1.3 Field (for several generators)

▷ `Field(`*z, ...*`)` (function)
▷ `Field(`*[F,]list*`)` (function)

`Field` returns the smallest field K that contains all the elements *z*, ..., or the smallest field K that contains all elements in the list *list*. If no subfield *F* is given, K is constructed as a field over itself, i.e. the left acting domain of K is K. Called with a field *F* and a list *list*, `Field` constructs the field generated by *F* and the elements in *list*, as a vector space over *F*.

58.1.4 DefaultField (for several generators)

▷ `DefaultField(`*z, ...*`)` (function)
▷ `DefaultField(`*list*`)` (function)

`DefaultField` returns a field K that contains all the elements *z*, ..., or a field K that contains all elements in the list *list*.

This field need not be the smallest field in which the elements lie, cf. `Field` (58.1.3). For example, for elements from cyclotomic fields `DefaultField` returns the smallest cyclotomic field in which the elements lie, but the elements may lie in a smaller number field which is not a cyclotomic field.

Example

```
gap> Field( Z(4) );  Field( [ Z(4), Z(8) ] );  # finite fields
GF(2^2)
GF(2^6)
gap> Field( E(9) );  Field( CF(4), [ E(9) ] ); # abelian number fields
CF(9)
AsField( GaussianRationals, CF(36) )
gap> f1:= Field( EB(5) );  f2:= DefaultField( EB(5) );
NF(5,[ 1, 4 ])
CF(5)
gap> f1 = f2;  IsSubset( f2, f1 );
false
true
```

58.1.5 DefaultFieldByGenerators

▷ `DefaultFieldByGenerators(`*[z, ...]*`)` (operation)

returns the default field containing the elements *z*, ... This field may be bigger than the smallest field containing these elements.

58.1.6 GeneratorsOfDivisionRing

▷ GeneratorsOfDivisionRing(*D*) (attribute)

generators with respect to addition, multiplication, and taking inverses (the identity cannot be omitted ...)

58.1.7 GeneratorsOfField

▷ GeneratorsOfField(*F*) (attribute)

generators with respect to addition, multiplication, and taking inverses. This attribute is the same as GeneratorsOfDivisionRing (58.1.6).

58.1.8 DivisionRingByGenerators

▷ DivisionRingByGenerators(*[F,]gens*) (operation)
▷ FieldByGenerators(*[F,]gens*) (operation)

Called with a field *F* and a list *gens* of scalars, DivisionRingByGenerators returns the division ring over *F* generated by *gens*. The unary version returns the division ring as vector space over FieldOverItselfByGenerators(*gens*).

FieldByGenerators is just a synonym for DivisionRingByGenerators.

58.1.9 AsDivisionRing

▷ AsDivisionRing(*[F,]C*) (operation)
▷ AsField(*[F,]C*) (operation)

If the collection *C* can be regarded as a division ring then AsDivisionRing(*C*) is the division ring that consists of the elements of *C*, viewed as a vector space over its prime field; otherwise fail is returned.

In the second form, if *F* is a division ring contained in *C* then the returned division ring is viewed as a vector space over *F*.

AsField is just a synonym for AsDivisionRing.

58.2 Subfields of Fields

58.2.1 Subfield

▷ Subfield(*F, gens*) (function)
▷ SubfieldNC(*F, gens*) (function)

Constructs the subfield of *F* generated by *gens*.

58.2.2 FieldOverItselfByGenerators

▷ `FieldOverItselfByGenerators(`*`[z, ...]`*`)` (operation)

This operation is needed for the call of `Field` (58.1.3) or `FieldByGenerators` (58.1.8) without explicitly given subfield, in order to construct a left acting domain for such a field.

58.2.3 PrimitiveElement

▷ `PrimitiveElement(`*`D`*`)` (attribute)

is an element of *D* that generates *D* as a division ring together with the left acting domain.

58.2.4 PrimeField

▷ `PrimeField(`*`D`*`)` (attribute)

The *prime field* of a division ring *D* is the smallest field which is contained in *D*. For example, the prime field of any field in characteristic zero is isomorphic to the field of rational numbers.

58.2.5 IsPrimeField

▷ `IsPrimeField(`*`D`*`)` (property)

A division ring is a prime field if it is equal to its prime field (see `PrimeField` (58.2.4)).

58.2.6 DegreeOverPrimeField

▷ `DegreeOverPrimeField(`*`F`*`)` (attribute)

is the degree of the field *F* over its prime field (see `PrimeField` (58.2.4)).

58.2.7 DefiningPolynomial

▷ `DefiningPolynomial(`*`F`*`)` (attribute)

is the defining polynomial of the field *F* as a field extension over the left acting domain of *F*. A root of the defining polynomial can be computed with `RootOfDefiningPolynomial` (58.2.8).

58.2.8 RootOfDefiningPolynomial

▷ `RootOfDefiningPolynomial(`*`F`*`)` (attribute)

is a root in the field *F* of its defining polynomial as a field extension over the left acting domain of *F*. The defining polynomial can be computed with `DefiningPolynomial` (58.2.7).

58.2.9 FieldExtension

▷ `FieldExtension(F, poly)` (operation)

is the field obtained on adjoining a root of the irreducible polynomial `poly` to the field `F`.

58.2.10 Subfields

▷ `Subfields(F)` (attribute)

is the set of all subfields of the field `F`.

58.3 Galois Action

Let $L > K$ be a field extension of finite degree. Then to each element $\alpha \in L$, we can associate a K-linear mapping φ_α on L, and for a fixed K-basis of L, we can associate to α the matrix M_α (over K) of this mapping.

The *norm* of α is defined as the determinant of M_α, the *trace* of α is defined as the trace of M_α, the *minimal polynomial* μ_α and the *trace polynomial* χ_α of α are defined as the minimal polynomial (see 66.8.1) and the characteristic polynomial (see `CharacteristicPolynomial` (24.13.1) and `TracePolynomial` (58.3.3)) of M_α. (Note that μ_α depends only on K whereas χ_α depends on both L and K.)

Thus norm and trace of α are elements of K, and μ_α and χ_α are polynomials over K, χ_α being a power of μ_α, and the degree of χ_α equals the degree of the field extension $L > K$.

The *conjugates* of α in L are those roots of χ_α (with multiplicity) that lie in L; note that if only L is given, there is in general no way to access the roots outside L.

Analogously, the *Galois group* of the extension $L > K$ is defined as the group of all those field automorphisms of L that fix K pointwise.

If $L > K$ is a Galois extension then the conjugates of α are all roots of χ_α (with multiplicity), the set of conjugates equals the roots of μ_α, the norm of α equals the product and the trace of α equals the sum of the conjugates of α, and the Galois group in the sense of the above definition equals the usual Galois group,

Note that `MinimalPolynomial( F, z )` is a polynomial *over* `F`, whereas `Norm( F, z )` is the norm of the element `z` *in* `F` w.r.t. the field extension `F > LeftActingDomain( F )`.

The default methods for field elements are as follows. `MinimalPolynomial` (66.8.1) solves a system of linear equations, `TracePolynomial` (58.3.3) computes the appropriate power of the minimal polynomial, `Norm` (58.3.4) and `Trace` (58.3.5) values are obtained as coefficients of the characteristic polynomial, and `Conjugates` (58.3.6) uses the factorization of the characteristic polynomial.

For elements in finite fields and cyclotomic fields, one wants to do the computations in a different way since the field extensions in question are Galois extensions, and the Galois groups are well-known in these cases. More general, if a field is in the category `IsFieldControlledByGaloisGroup` then the default methods are the following. `Conjugates` (58.3.6) returns the sorted list of images (with multiplicity) of the element under the Galois group, `Norm` (58.3.4) computes the product of the conjugates, `Trace` (58.3.5) computes the sum of the conjugates, `TracePolynomial` (58.3.3) and `MinimalPolynomial` (66.8.1) compute the product of linear factors $x - c$ with c ranging over the conjugates and the set of conjugates, respectively.

58.3.1 GaloisGroup (of field)

▷ GaloisGroup(*F*) (attribute)

The *Galois group* of a field *F* is the group of all field automorphisms of *F* that fix the subfield *K* =LeftActingDomain(*F*) pointwise.

Note that the field extension $F > K$ need *not* be a Galois extension.

Example

```
gap> g:= GaloisGroup( AsField( GF(2^2), GF(2^12) ) );;
gap> Size( g );  IsCyclic( g );
6
true
gap> h:= GaloisGroup( CF(60) );;
gap> Size( h );  IsAbelian( h );
16
true
```

58.3.2 MinimalPolynomial (over a field)

▷ MinimalPolynomial(*F, z[, ind]*) (function)

returns the minimal polynomial of *z* over the field *F*. This is a generator of the ideal in $F[x]$ of all polynomials which vanish on *z*. (This definition is consistent with the general definition of MinimalPolynomial (66.8.1) for rings.)

Example

```
gap> MinimalPolynomial( Rationals, E(8) );
x_1^4+1
gap> MinimalPolynomial( CF(4), E(8) );
x_1^2+(-E(4))
gap> MinimalPolynomial( CF(8), E(8) );
x_1+(-E(8))
```

58.3.3 TracePolynomial

▷ TracePolynomial(*L, K, z[, inum]*) (operation)

returns the polynomial that is the product of $(X - c)$ where c runs over the conjugates of *z* in the field extension *L* over *K*. The polynomial is returned as a univariate polynomial over *K* in the indeterminate number *inum* (defaulting to 1).

This polynomial is sometimes also called the *characteristic polynomial* of *z* w.r.t. the field extension $L > K$. Therefore methods are installed for CharacteristicPolynomial (24.13.1) that call TracePolynomial in the case of field extensions.

Example

```
gap> TracePolynomial( CF(8), Rationals, E(8) );
x_1^4+1
gap> TracePolynomial( CF(16), Rationals, E(8) );
x_1^8+2*x_1^4+1
```

58.3.4 Norm

▷ Norm([L[, K,]]z) (attribute)

Norm returns the norm of the field element z. If two fields L and K are given then the norm is computed w.r.t. the field extension L>K, if only one field L is given then LeftActingDomain(L) is taken as default for the subfield K, and if no field is given then DefaultField(z) is taken as default for L.

58.3.5 Traces of field elements and matrices

▷ Trace([L[, K,]]z) (attribute)
▷ Trace(mat) (attribute)

Trace returns the trace of the field element z. If two fields L and K are given then the trace is computed w.r.t. the field extension L > K, if only one field L is given then LeftActingDomain(L) is taken as default for the subfield K, and if no field is given then DefaultField(z) is taken as default for L.

The *trace of a matrix* is the sum of its diagonal entries. Note that this is *not* compatible with the definition of Trace for field elements, so the one-argument version is not suitable when matrices shall be regarded as field elements.

58.3.6 Conjugates

▷ Conjugates([L[, K,]]z) (attribute)

Conjugates returns the list of *conjugates* of the field element z. If two fields L and K are given then the conjugates are computed w.r.t. the field extension L>K, if only one field L is given then LeftActingDomain(L) is taken as default for the subfield K, and if no field is given then DefaultField(z) is taken as default for L.

The result list will contain duplicates if z lies in a proper subfield of L, or of the default field of z, respectively. The result list need not be sorted.

Example

```
gap> Norm( E(8) );  Norm( CF(8), E(8) );
1
1
gap> Norm( CF(8), CF(4), E(8) );
-E(4)
gap> Norm( AsField( CF(4), CF(8) ), E(8) );
-E(4)
gap> Trace( E(8) );  Trace( CF(8), CF(8), E(8) );
0
E(8)
gap> Conjugates( CF(8), E(8) );
[ E(8), E(8)^3, -E(8), -E(8)^3 ]
gap> Conjugates( CF(8), CF(4), E(8) );
[ E(8), -E(8) ]
gap> Conjugates( CF(16), E(8) );
[ E(8), E(8)^3, -E(8), -E(8)^3, E(8), E(8)^3, -E(8), -E(8)^3 ]
```

58.3.7 NormalBase

▷ NormalBase(*F[, elm]*) (attribute)

Let *F* be a field that is a Galois extension of its subfield LeftActingDomain(*F*). Then NormalBase returns a list of elements in *F* that form a normal basis of *F*, that is, a vector space basis that is closed under the action of the Galois group (see GaloisGroup (58.3.1)) of *F*.

If a second argument *elm* is given, it is used as a hint for the algorithm to find a normal basis with the algorithm described in [Art73].

Example

```
gap> NormalBase( CF(5) );
[ -E(5), -E(5)^2, -E(5)^3, -E(5)^4 ]
gap> NormalBase( CF(4) );
[ 1/2-1/2*E(4), 1/2+1/2*E(4) ]
gap> NormalBase( GF(3^6) );
[ Z(3^6)^2, Z(3^6)^6, Z(3^6)^18, Z(3^6)^54, Z(3^6)^162, Z(3^6)^486 ]
gap> NormalBase( GF( GF(8), 2 ) );
[ Z(2^6), Z(2^6)^8 ]
```

Chapter 59

Finite Fields

This chapter describes the special functionality which exists in GAP for finite fields and their elements. Of course the general functionality for fields (see Chapter 58) also applies to finite fields.

In the following, the term *finite field element* is used to denote GAP objects in the category `IsFFE` (59.1.1), and *finite field* means a field consisting of such elements. Note that in principle we must distinguish these fields from (abstract) finite fields. For example, the image of the embedding of a finite field into a field of rational functions in the same characteristic is of course a finite field but its elements are not in `IsFFE` (59.1.1), and in fact GAP does currently not support such fields.

Special representations exist for row vectors and matrices over small finite fields (see sections 23.3 and 24.14).

59.1 Finite Field Elements

59.1.1 IsFFE

▷ `IsFFE(`*obj*`)` (Category)
▷ `IsFFECollection(`*obj*`)` (Category)
▷ `IsFFECollColl(`*obj*`)` (Category)
▷ `IsFFECollCollColl(`*obj*`)` (Category)

Objects in the category `IsFFE` are used to implement elements of finite fields. In this manual, the term *finite field element* always means an object in `IsFFE`. All finite field elements of the same characteristic form a family in GAP (see 13.1). Any collection of finite field elements (see `IsCollection` (30.1.1)) lies in `IsFFECollection`, and a collection of such collections (e.g., a matrix of finite field elements) lies in `IsFFECollColl`.

59.1.2 Z (for field size)

▷ `Z(`*p*`^`*d*`)` (function)
▷ `Z(`*p*`, `*d*`)` (function)

For creating elements of a finite field, the function `Z` can be used. The call `Z(`*p*`,`*d*`)` (alternatively `Z(`*p*`^`*d*`)`) returns the designated generator of the multiplicative group of the finite field with *p*`^`*d* elements. *p* must be a prime integer.

GAP can represent elements of all finite fields `GF(p^d)` such that either (1) `p^d` $<= 65536$ (in which case an extremely efficient internal representation is used); (2) d = 1, (in which case, for large p, the field is represented using the machinery of residue class rings (see section 14.5) or (3) if the Conway polynomial of degree d over the field with p elements is known, or can be computed (see `ConwayPolynomial` (59.5.1)).

If you attempt to construct an element of `GF(p^d)` for which $d > 1$ and the relevant Conway polynomial is not known, and not necessarily easy to find (see `IsCheapConwayPolynomial` (59.5.2)), then GAP will stop with an error and enter the break loop. If you leave this break loop by entering `return;` GAP will attempt to compute the Conway polynomial, which may take a very long time.

The root returned by `Z` is a generator of the multiplicative group of the finite field with `p^d` elements, which is cyclic. The order of the element is of course `p^d` -1. The `p^d` -1 different powers of the root are exactly the nonzero elements of the finite field.

Thus all nonzero elements of the finite field with `p^d` elements can be entered as `Z(p^d)^i`. Note that this is also the form that GAP uses to output those elements when they are stored in the internal representation. In larger fields, it is more convenient to enter and print elements as linear combinations of powers of the primitive element, see section 59.6.

The additive neutral element is `0 * Z(p)`. It is different from the integer `0` in subtle ways. First `IsInt( 0 * Z(p) )` (see `IsInt` (14.2.1)) is `false` and `IsFFE( 0 * Z(p) )` (see `IsFFE` (59.1.1)) is `true`, whereas it is just the other way around for the integer `0`.

The multiplicative neutral element is `Z(p)^0`. It is different from the integer `1` in subtle ways. First `IsInt( Z(p)^0 )` (see `IsInt` (14.2.1)) is `false` and `IsFFE( Z(p)^0 )` (see `IsFFE` (59.1.1)) is `true`, whereas it is just the other way around for the integer `1`. Also `1+1` is `2`, whereas, e.g., `Z(2)^0 + Z(2)^0` is `0 * Z(2)`.

The various roots returned by `Z` for finite fields of the same characteristic are compatible in the following sense. If the field `GF(p,n)` is a subfield of the field `GF(p,m)`, i.e., n divides m, then `Z`$(p^n) =$`Z`$(p^m)^{(p^m-1)/(p^n-1)}$. Note that this is the simplest relation that may hold between a generator of `GF(p,n)` and `GF(p,m)`, since `Z`(p^n) is an element of order $p^m - 1$ and `Z`(p^m) is an element of order $p^n - 1$. This is achieved by choosing `Z(p)` as the smallest primitive root modulo p and `Z(p^n)` as a root of the n-th *Conway polynomial* (see `ConwayPolynomial` (59.5.1)) of characteristic p. Those polynomials were defined by J. H. Conway, and many of them were computed by R. A. Parker.

Example

```
gap> a:= Z( 32 );
Z(2^5)
gap> a+a;
0*Z(2)
gap> a*a;
Z(2^5)^2
gap> b := Z(3,12);
z
gap> b*b;
z2
gap> b+b;
2z
gap> Print(b^100,"\n");
Z(3)^0+Z(3,12)^5+Z(3,12)^6+2*Z(3,12)^8+Z(3,12)^10+Z(3,12)^11
```

Example

```
gap> Z(11,40);
Error, Conway Polynomial 11^40 will need to computed and might be slow
```

```
return to continue called from
FFECONWAY.ZNC( p, d ) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk>
```

59.1.3 IsLexOrderedFFE

▷ IsLexOrderedFFE(*ffe*) (Category)
▷ IsLogOrderedFFE(*ffe*) (Category)

Elements of finite fields can be compared using the operators = and <. The call `a = b` returns `true` if and only if the finite field elements `a` and `b` are equal. Furthermore `a < b` tests whether `a` is smaller than `b`. The exact behaviour of this comparison depends on which of two categories the field elements belong to:

Finite field elements are ordered in GAP (by \< (31.11.1)) first by characteristic and then by their degree (i.e. the sizes of the smallest fields containing them). Amongst irreducible elements of a given field, the ordering depends on which of these categories the elements of the field belong to (all irreducible elements of a given field should belong to the same one)

Elements in `IsLexOrderedFFE` are ordered lexicographically by their coefficients with respect to the canonical basis of the field.

Elements in `IsLogOrderedFFE` are ordered according to their discrete logarithms with respect to the `PrimitiveElement` (58.2.3) attribute of the field. For the comparison of finite field elements with other GAP objects, see 4.12.

Example

```
gap> Z( 16 )^10 = Z( 4 )^2;  # illustrates embedding of GF(4) in GF(16)
true
gap> 0 < 0*Z(101);
true
gap> Z(256) > Z(101);
false
gap> Z(2,20) < Z(2,20)^2; # this illustrates the lexicographic ordering
false
```

59.2 Operations for Finite Field Elements

Since finite field elements are scalars, the operations `Characteristic` (31.10.1), `One` (31.10.2), `Zero` (31.10.3), `Inverse` (31.10.8), `AdditiveInverse` (31.10.9), `Order` (31.10.10) can be applied to them (see 31.10). Contrary to the situation with other scalars, `Order` (31.10.10) is defined also for the zero element in a finite field, with value `0`.

Example

```
gap> Characteristic( Z( 16 )^10 );  Characteristic( Z( 9 )^2 );
2
3
gap> Characteristic( [ Z(4), Z(8) ] );
2
```

```
gap> One( Z(9) );  One( 0*Z(4) );
Z(3)^0
Z(2)^0
gap> Inverse( Z(9) );  AdditiveInverse( Z(9) );
Z(3^2)^7
Z(3^2)^5
gap> Order( Z(9)^7 );
8
```

59.2.1 DegreeFFE (for a FFE)

▷ DegreeFFE(*z*) (operation)
▷ DegreeFFE(*vec*) (operation)
▷ DegreeFFE(*mat*) (operation)

DegreeFFE returns the degree of the smallest finite field *F* containing the element *z*, respectively all elements of the row vector *vec* over a finite field (see 23), or the matrix *mat* over a finite field (see 24).

Example

```
gap> DegreeFFE( Z( 16 )^10 );
2
gap> DegreeFFE( Z( 16 )^11 );
4
gap> DegreeFFE( [ Z(2^13), Z(2^10) ] );
130
```

59.2.2 LogFFE

▷ LogFFE(*z*, *r*) (operation)

LogFFE returns the discrete logarithm of the element *z* in a finite field with respect to the root *r*. An error is signalled if *z* is zero. fail is returned if *z* is not a power of *r*.

The *discrete logarithm* of the element *z* with respect to the root *r* is the smallest nonnegative integer *i* such that $r^i = z$ holds.

Example

```
gap> LogFFE( Z(409)^116, Z(409) );  LogFFE( Z(409)^116, Z(409)^2 );
116
58
```

59.2.3 IntFFE

▷ IntFFE(*z*) (attribute)
▷ Int(*z*) (method)

IntFFE returns the integer corresponding to the element *z*, which must lie in a finite prime field. That is, IntFFE returns the smallest nonnegative integer *i* such that *i* * One(*z*) = *z*.

The correspondence between elements from a finite prime field of characteristic p (for $p < 2^{16}$) and the integers between 0 and $p-1$ is defined by choosing `Z(`p`)` the element corresponding to the smallest primitive root mod p (see `PrimitiveRootMod` (15.3.3)).

`IntFFE` is installed as a method for the operation `Int` (14.2.3) with argument a finite field element.

Example

```
gap> IntFFE( Z(13) );  PrimitiveRootMod( 13 );
2
2
gap> IntFFE( Z(409) );
21
gap> IntFFE( Z(409)^116 );  21^116 mod 409;
311
311
```

See also `IntFFESymm` (59.2.4).

59.2.4 IntFFESymm (for a FFE)

▷ `IntFFESymm(`*z*`)` (attribute)
▷ `IntFFESymm(`*vec*`)` (attribute)

For a finite prime field element *z*, `IntFFESymm` returns the corresponding integer of smallest absolute value. That is, `IntFFESymm` returns the integer i of smallest absolute value such that i `* One( z ) = z` holds.

For a vector *vec* of FFEs, the operation returns the result of applying `IntFFESymm` to every entry of the vector.

The correspondence between elements from a finite prime field of characteristic p (for $p < 2^{16}$) and the integers between $-p/2$ and $p/2$ is defined by choosing `Z(`p`)` the element corresponding to the smallest positive primitive root mod p (see `PrimitiveRootMod` (15.3.3)) and reducing results to the $-p/2..p/2$ range.

Example

```
gap> IntFFE(Z(13)^2);IntFFE(Z(13)^3);
4
8
gap> IntFFESymm(Z(13)^2);IntFFESymm(Z(13)^3);
4
-5
```

See also `IntFFE` (59.2.3)

59.2.5 IntVecFFE

▷ `IntVecFFE(`*vecffe*`)` (operation)

is the list of integers corresponding to the vector *vecffe* of finite field elements in a prime field (see `IntFFE` (59.2.3)).

59.2.6 AsInternalFFE

▷ AsInternalFFE(*ffe*) (attribute)

return an internal FFE equal to *ffe* if one exists, otherwise fail

59.3 Creating Finite Fields

59.3.1 DefaultField (for finite field elements)

▷ DefaultField(*list*) (method)
▷ DefaultRing(*list*) (method)

DefaultField and DefaultRing for finite field elements are defined to return the *smallest* field containing the given elements.

Example

```
gap> DefaultField( [ Z(4), Z(4)^2 ] );  DefaultField( [ Z(4), Z(8) ] );
GF(2^2)
GF(2^6)
```

59.3.2 GaloisField (for field size)

▷ GaloisField(*p^d*) (function)
▷ GF(*p^d*) (function)
▷ GaloisField(*p*, *d*) (function)
▷ GF(*p*, *d*) (function)
▷ GaloisField(*subfield*, *d*) (function)
▷ GF(*subfield*, *d*) (function)
▷ GaloisField(*p*, *pol*) (function)
▷ GF(*p*, *pol*) (function)
▷ GaloisField(*subfield*, *pol*) (function)
▷ GF(*subfield*, *pol*) (function)

GaloisField returns a finite field. It takes two arguments. The form GaloisField(*p*, *d*), where *p*, *d* are integers, can also be given as GaloisField(*p^d*). GF is an abbreviation for GaloisField.

The first argument specifies the subfield S over which the new field is to be taken. It can be a prime integer or a finite field. If it is a prime *p*, the subfield is the prime field of this characteristic.

The second argument specifies the extension. It can be an integer or an irreducible polynomial over the field S. If it is an integer *d*, the new field is constructed as the polynomial extension w.r.t. the Conway polynomial (see ConwayPolynomial (59.5.1)) of degree *d* over S. If it is an irreducible polynomial *pol* over S, the new field is constructed as polynomial extension of S with this polynomial; in this case, *pol* is accessible as the value of DefiningPolynomial (58.2.7) for the new field, and a root of *pol* in the new field is accessible as the value of RootOfDefiningPolynomial (58.2.8).

Note that the subfield over which a field was constructed determines over which field the Galois group, conjugates, norm, trace, minimal polynomial, and trace polynomial are

computed (see GaloisGroup (58.3.1), Conjugates (58.3.6), Norm (58.3.4), Trace (58.3.5), MinimalPolynomial (58.3.2), TracePolynomial (58.3.3)).

The field is regarded as a vector space (see 61) over the given subfield, so this determines the dimension and the canonical basis of the field.

Example

```
gap> f1:= GF( 2^4 );
GF(2^4)
gap> Size( GaloisGroup ( f1 ) );
4
gap> BasisVectors( Basis( f1 ) );
[ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ]
gap> f2:= GF( GF(4), 2 );
AsField( GF(2^2), GF(2^4) )
gap> Size( GaloisGroup( f2 ) );
2
gap> BasisVectors( Basis( f2 ) );
[ Z(2)^0, Z(2^4) ]
```

59.3.3 PrimitiveRoot

▷ PrimitiveRoot(*F*) (attribute)

A *primitive root* of a finite field is a generator of its multiplicative group. A primitive root is always a primitive element (see PrimitiveElement (58.2.3)), the converse is in general not true.

Example

```
gap> f:= GF( 3^5 );
GF(3^5)
gap> PrimitiveRoot( f );
Z(3^5)
```

59.4 Frobenius Automorphisms

59.4.1 FrobeniusAutomorphism

▷ FrobeniusAutomorphism(*F*) (attribute)

returns the Frobenius automorphism of the finite field *F* as a field homomorphism (see 32.12).

The *Frobenius automorphism* f of a finite field F of characteristic p is the function that takes each element z of F to its p-th power. Each field automorphism of F is a power of f. Thus f is a generator for the Galois group of F relative to the prime field of F, and an appropriate power of f is a generator of the Galois group of F over a subfield (see GaloisGroup (58.3.1)).

Example

```
gap> f := GF(16);
GF(2^4)
gap> x := FrobeniusAutomorphism( f );
FrobeniusAutomorphism( GF(2^4) )
gap> Z(16) ^ x;
Z(2^4)^2
```

```
gap> x^2;
FrobeniusAutomorphism( GF(2^4) )^2
```

The image of an element z under the i-th power of f is computed as the p^i-th power of z. The product of the i-th power and the j-th power of f is the k-th power of f, where k is ij mod `Size(F)`-1. The zeroth power of f is `IdentityMapping( F )`.

59.5 Conway Polynomials

59.5.1 ConwayPolynomial

▷ `ConwayPolynomial(p, n)` (function)

is the Conway polynomial of the finite field $GF(p^n)$ as polynomial over the prime field in characteristic `p`.

The *Conway polynomial* $\Phi_{n,p}$ of $GF(p^n)$ is defined by the following properties.

First define an ordering of polynomials of degree n over $GF(p)$, as follows. $f = \sum_{i=0}^{n}(-1)^i f_i x^i$ is smaller than $g = \sum_{i=0}^{n}(-1)^i g_i x^i$ if and only if there is an index $m \leq n$ such that $f_i = g_i$ for all $i > m$, and $\tilde{f_m} < \tilde{g_m}$, where $\tilde{c}$ denotes the integer value in $\{0, 1, \ldots, p-1\}$ that is mapped to $c \in GF(p)$ under the canonical epimorphism that maps the integers onto $GF(p)$.

$\Phi_{n,p}$ is *primitive* over $GF(p)$ (see `IsPrimitivePolynomial` (66.4.12)). That is, $\Phi_{n,p}$ is irreducible, monic, and is the minimal polynomial of a primitive root of $GF(p^n)$.

For all divisors d of n the compatibility condition $\Phi_{d,p}(x^{\frac{p^n-1}{p^m-1}}) \equiv 0 \pmod{\Phi_{n,p}(x)}$ holds. (That is, the appropriate power of a zero of $\Phi_{n,p}$ is a zero of the Conway polynomial $\Phi_{d,p}$.)

With respect to the ordering defined above, $\Phi_{n,p}$ shall be minimal.

The computation of Conway polynomials can be time consuming. Therefore, GAP comes with a list of precomputed polynomials. If a requested polynomial is not stored then GAP prints a warning and computes it by checking all polynomials in the order defined above for the defining conditions. If n is not a prime this is probably a very long computation. (Some previously known polynomials with prime n are not stored in GAP because they are quickly recomputed.) Use the function `IsCheapConwayPolynomial` (59.5.2) to check in advance if `ConwayPolynomial` will give a result after a short time.

Note that primitivity of a polynomial can only be checked if GAP can factorize $p^n - 1$. A sufficiently new version of the FactInt package contains many precomputed factors of such numbers from various factorization projects.

See [Lüb03] for further information on known Conway polynomials.

An interactive overview of the Conway polynomials known to GAP is provided by the function `BrowseConwayPolynomials` from the GAP package Browse, see `BrowseGapData` (**Browse: BrowseGapData**).

If *pol* is a result returned by `ConwayPolynomial` the command `Print( InfoText( pol ) );` will print some info on the origin of that particular polynomial.

For some purposes it may be enough to have any primitive polynomial for an extension of a finite field instead of the Conway polynomial, see `RandomPrimitivePolynomial` (59.5.3) below.

Example

```
gap> ConwayPolynomial( 2, 5 );  ConwayPolynomial( 3, 7 );
x_1^5+x_1^2+Z(2)^0
x_1^7-x_1^2+Z(3)^0
```

59.5.2 IsCheapConwayPolynomial

▷ `IsCheapConwayPolynomial(p, n)` (function)

Returns `true` if `ConwayPolynomial( p, n )` will give a result in *reasonable* time. This is either the case when this polynomial is pre-computed, or if `n` is a not too big prime.

59.5.3 RandomPrimitivePolynomial

▷ `RandomPrimitivePolynomial(F, n[, i])` (function)

For a finite field `F` and a positive integer `n` this function returns a primitive polynomial of degree `n` over `F`, that is a zero of this polynomial has maximal multiplicative order $|F|^n - 1$. If `i` is given then the polynomial is written in variable number `i` over `F` (see `Indeterminate` (66.1.1)), the default for `i` is 1.

Alternatively, `F` can be a prime power q, then `F` = GF(q) is assumed. And `i` can be a univariate polynomial over `F`, then the result is a polynomial in the same variable.

This function can work for much larger fields than those for which Conway polynomials are available, of course GAP must be able to factorize $|F|^n - 1$.

59.6 Printing, Viewing and Displaying Finite Field Elements

59.6.1 ViewObj (for a ffe)

▷ `ViewObj(z)` (method)
▷ `PrintObj(z)` (method)
▷ `Display(z)` (method)

Internal finite field elements are viewed, printed and displayed (see section 6.3 for the distinctions between these operations) as powers of the primitive root (except for the zero element, which is displayed as 0 times the primitive root). Thus:

Example

```
gap> Z(2);
Z(2)^0
gap> Z(5)+Z(5);
Z(5)^2
gap> Z(256);
Z(2^8)
gap> Zero(Z(125));
0*Z(5)
```

Note also that each element is displayed as an element of the field it generates, and that the size of the field is printed as a power of the characteristic.

Elements of larger fields are printed as GAP expressions which represent them as sums of low powers of the primitive root:

Example

```
gap> Print( Z(3,20)^100, "\n" );
2*Z(3,20)^2+Z(3,20)^4+Z(3,20)^6+Z(3,20)^7+2*Z(3,20)^9+2*Z(3,20)^10+2*Z\
```

```
(3,20)^12+2*Z(3,20)^15+2*Z(3,20)^17+Z(3,20)^18+Z(3,20)^19
gap> Print( Z(3,20)^((3^20-1)/(3^10-1)), "\n" );
Z(3,20)^3+2*Z(3,20)^4+2*Z(3,20)^7+Z(3,20)^8+2*Z(3,20)^10+Z(3,20)^11+2*\
Z(3,20)^12+Z(3,20)^13+Z(3,20)^14+Z(3,20)^15+Z(3,20)^17+Z(3,20)^18+2*Z(\
3,20)^19
gap> Z(3,20)^((3^20-1)/(3^10-1)) = Z(3,10);
true
```

Note from the second example above, that these elements are not always written over the smallest possible field before being output.

The `ViewObj` and `Display` methods for these large finite field elements use a slightly more compact, but mathematically equivalent representation. The primitive root is represented by `z`; its *i*-th power by `z`*i* and *k* times this power by *k*`z`*i*.

Example

```
gap> Z(5,20)^100;
z2+z4+4z5+2z6+z8+3z9+4z10+3z12+z13+2z14+4z16+3z17+2z18+2z19
```

This output format is always used for `Display`. For `ViewObj` it is used only if its length would not exceed the number of lines specified in the user preference `ViewLength` (see `SetUserPreference` (3.2.3). Longer output is replaced by `<<an element of GF(`*p*`, `*d*`)>>`.

Example

```
gap> Z(2,409)^100000;
<<an element of GF(2, 409)>>
gap> Display(Z(2,409)^100000);
z2+z3+z4+z5+z6+z7+z8+z10+z11+z13+z17+z19+z20+z29+z32+z34+z35+z37+z40+z\
45+z46+z48+z50+z52+z54+z55+z58+z59+z60+z66+z67+z68+z70+z74+z79+z80+z81\
+z82+z83+z86+z91+z93+z94+z95+z96+z98+z99+z100+z101+z102+z104+z106+z109\
+z110+z112+z114+z115+z118+z119+z123+z126+z127+z135+z138+z140+z142+z143\
+z146+z147+z154+z159+z161+z162+z168+z170+z171+z173+z174+z181+z182+z183\
+z186+z188+z189+z192+z193+z194+z195+z196+z199+z202+z204+z205+z207+z208\
+z209+z211+z212+z213+z214+z215+z216+z218+z219+z220+z222+z223+z229+z232\
+z235+z236+z237+z238+z240+z243+z244+z248+z250+z251+z256+z258+z262+z263\
+z268+z270+z271+z272+z274+z276+z282+z286+z288+z289+z294+z295+z299+z300\
+z301+z302+z303+z304+z305+z306+z307+z308+z309+z310+z312+z314+z315+z316\
+z320+z321+z322+z324+z325+z326+z327+z330+z332+z335+z337+z338+z341+z344\
+z348+z350+z352+z353+z356+z357+z358+z360+z362+z364+z366+z368+z372+z373\
+z374+z375+z378+z379+z380+z381+z383+z384+z386+z387+z390+z395+z401+z402\
+z406+z408
```

Finally note that elements of large prime fields are stored and displayed as residue class objects. So

Example

```
gap> Z(65537);
ZmodpZObj( 3, 65537 )
```

Chapter 60

Abelian Number Fields

An *abelian number field* is a field in characteristic zero that is a finite dimensional normal extension of its prime field such that the Galois group is abelian. In GAP, one implementation of abelian number fields is given by fields of cyclotomic numbers (see Chapter 18). Note that abelian number fields can also be constructed with the more general `AlgebraicExtension` (67.1.1), a discussion of advantages and disadvantages can be found in 18.6. The functions described in this chapter have been developed for fields whose elements are in the filter `IsCyclotomic` (18.1.3), they may or may not work well for abelian number fields consisting of other kinds of elements.

Throughout this chapter, $\mathbb{Q}_n$ will denote the cyclotomic field generated by the field $\mathbb{Q}$ of rationals together with n-th roots of unity.

In 60.1, constructors for abelian number fields are described, 60.2 introduces operations for abelian number fields, 60.3 deals with the vector space structure of abelian number fields, and 60.4 describes field automorphisms of abelian number fields,

60.1 Construction of Abelian Number Fields

Besides the usual construction using `Field` (58.1.3) or `DefaultField` (18.1.16) (see `DefaultField` (18.1.16)), abelian number fields consisting of cyclotomics can be created with `CyclotomicField` (60.1.1) and `AbelianNumberField` (60.1.2).

60.1.1 CyclotomicField (for (subfield and) conductor)

▷ `CyclotomicField(`*`[subfield, ]n`*`)` (function)
▷ `CyclotomicField(`*`[subfield, ]gens`*`)` (function)
▷ `CF(`*`[subfield, ]n`*`)` (function)
▷ `CF(`*`[subfield, ]gens`*`)` (function)

The first version creates the *n*-th cyclotomic field $\mathbb{Q}_n$. The second version creates the smallest cyclotomic field containing the elements in the list *gens*. In both cases the field can be generated as an extension of a designated subfield *subfield* (cf. 60.3).

`CyclotomicField` can be abbreviated to `CF`, this form is used also when GAP prints cyclotomic fields.

Fields constructed with the one argument version of `CF` are stored in the global list `CYCLOTOMIC_FIELDS`, so repeated calls of `CF` just fetch these field objects after they have been created

once.

Example

```
gap> CyclotomicField( 5 );  CyclotomicField( [ Sqrt(3) ] );
CF(5)
CF(12)
gap> CF( CF(3), 12 );  CF( CF(4), [ Sqrt(7) ] );
AsField( CF(3), CF(12) )
AsField( GaussianRationals, CF(28) )
```

60.1.2 AbelianNumberField

▷ AbelianNumberField(*n*, *stab*) (function)
▷ NF(*n*, *stab*) (function)

For a positive integer *n* and a list *stab* of prime residues modulo *n*, AbelianNumberField returns the fixed field of the group described by *stab* (cf. GaloisStabilizer (60.2.5)), in the *n*-th cyclotomic field. AbelianNumberField is mainly thought for internal use and for printing fields in a standard way; Field (58.1.3) (cf. also 60.2) is probably more suitable if one knows generators of the field in question.

AbelianNumberField can be abbreviated to NF, this form is used also when GAP prints abelian number fields.

Fields constructed with NF are stored in the global list ABELIAN_NUMBER_FIELDS, so repeated calls of NF just fetch these field objects after they have been created once.

Example

```
gap> NF( 7, [ 1 ] );
CF(7)
gap> f:= NF( 7, [ 1, 2 ] );  Sqrt(-7); Sqrt(-7) in f;
NF(7,[ 1, 2, 4 ])
E(7)+E(7)^2-E(7)^3+E(7)^4-E(7)^5-E(7)^6
true
```

60.1.3 GaussianRationals

▷ GaussianRationals (global variable)
▷ IsGaussianRationals(*obj*) (Category)

GaussianRationals is the field $\mathbb{Q}_4 = \mathbb{Q}(\sqrt{-1})$ of Gaussian rationals, as a set of cyclotomic numbers, see Chapter 18 for basic operations. This field can also be obtained as CF(4) (see CyclotomicField (60.1.1)).

The filter IsGaussianRationals returns true for the GAP object GaussianRationals, and false for all other GAP objects.

(For details about the field of rationals, see Chapter Rationals (17.1.1).)

Example

```
gap> CF(4) = GaussianRationals;
true
gap> Sqrt(-1) in GaussianRationals;
true
```

60.2 Operations for Abelian Number Fields

For operations for elements of abelian number fields, e.g., `Conductor` (18.1.7) or `ComplexConjugate` (18.5.2), see Chapter 18.

60.2.1 Factors (for polynomials over abelian number fields)

▷ `Factors(`*F*`)` (method)

Factoring of polynomials over abelian number fields consisting of cyclotomics works in principle but is not very efficient if the degree of the field extension is large.

Example

```
gap> x:= Indeterminate( CF(5) );
x_1
gap> Factors( PolynomialRing( Rationals ), x^5-1 );
[ x_1-1, x_1^4+x_1^3+x_1^2+x_1+1 ]
gap> Factors( PolynomialRing( CF(5) ), x^5-1 );
[ x_1-1, x_1+(-E(5)), x_1+(-E(5)^2), x_1+(-E(5)^3), x_1+(-E(5)^4) ]
```

60.2.2 IsNumberField

▷ `IsNumberField(`*F*`)` (property)

returns `true` if the field *F* is a finite dimensional extension of a prime field in characteristic zero, and `false` otherwise.

60.2.3 IsAbelianNumberField

▷ `IsAbelianNumberField(`*F*`)` (property)

returns `true` if the field *F* is a number field (see `IsNumberField` (60.2.2)) that is a Galois extension of the prime field, with abelian Galois group (see `GaloisGroup` (58.3.1)).

60.2.4 IsCyclotomicField

▷ `IsCyclotomicField(`*F*`)` (property)

returns `true` if the field *F* is a *cyclotomic field*, i.e., an abelian number field (see `IsAbelianNumberField` (60.2.3)) that can be generated by roots of unity.

Example

```
gap> IsNumberField( CF(9) ); IsAbelianNumberField( Field( [ ER(3) ] ) );
true
true
gap> IsNumberField( GF(2) );
false
gap> IsCyclotomicField( CF(9) );
true
gap> IsCyclotomicField( Field( [ Sqrt(-3) ] ) );
true
```

```
gap> IsCyclotomicField( Field( [ Sqrt(3) ] ) );
false
```

60.2.5 GaloisStabilizer

▷ `GaloisStabilizer(F)` (attribute)

Let F be an abelian number field (see `IsAbelianNumberField` (60.2.3)) with conductor n, say. (This means that the n-th cyclotomic field is the smallest cyclotomic field containing F, see `Conductor` (18.1.7).) `GaloisStabilizer` returns the set of all those integers k in the range $[1..n]$ such that the field automorphism induced by raising n-th roots of unity to the k-th power acts trivially on F.

Example

```
gap> r5:= Sqrt(5);
E(5)-E(5)^2-E(5)^3+E(5)^4
gap> GaloisCyc( r5, 4 ) = r5;  GaloisCyc( r5, 2 ) = r5;
true
false
gap> GaloisStabilizer( Field( [ r5 ] ) );
[ 1, 4 ]
```

60.3 Integral Bases of Abelian Number Fields

Each abelian number field is naturally a vector space over $\mathbb{Q}$. Moreover, if the abelian number field F contains the n-th cyclotomic field $\mathbb{Q}_n$ then F is a vector space over $\mathbb{Q}_n$. In GAP, each field object represents a vector space object over a certain subfield S, which depends on the way F was constructed. The subfield S can be accessed as the value of the attribute `LeftActingDomain` (57.1.11).

The return values of `NF` (60.1.2) and of the one argument versions of `CF` (60.1.1) represent vector spaces over $\mathbb{Q}$, and the return values of the two argument version of `CF` (60.1.1) represent vector spaces over the field that is given as the first argument. For an abelian number field F and a subfield S of F, a GAP object representing F as a vector space over S can be constructed using `AsField` (58.1.9).

Let F be the cyclotomic field $\mathbb{Q}_n$, represented as a vector space over the subfield S. If S is the cyclotomic field $\mathbb{Q}_m$, with m a divisor of n, then `CanonicalBasis( F )` returns the Zumbroich basis of F relative to S, which consists of the roots of unity `E(n)^i` where `i` is an element of the list `ZumbroichBase( n, m )` (see `ZumbroichBase` (60.3.1)). If S is an abelian number field that is not a cyclotomic field then `CanonicalBasis( F )` returns a normal S-basis of F, i.e., a basis that is closed under the field automorphisms of F.

Let F be the abelian number field `NF( n, stab )`, with conductor n, that is itself not a cyclotomic field, represented as a vector space over the subfield S. If S is the cyclotomic field $\mathbb{Q}_m$, with m a divisor of n, then `CanonicalBasis( F )` returns the Lenstra basis of F relative to S that consists of the sums of roots of unity described by `LenstraBase( n, stab, stab, m )` (see `LenstraBase` (60.3.2)). If S is an abelian number field that is not a cyclotomic field then `CanonicalBasis( F )` returns a normal S-basis of F.

Example

```
gap> f:= CF(8);;   # a cycl. field over the rationals
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1, E(8), E(4), E(8)^3 ]
```

```
gap> Coefficients( b, Sqrt(-2) );
[ 0, 1, 0, 1 ]
gap> f:= AsField( CF(4), CF(8) );;  # a cycl. field over a cycl. field
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1, E(8) ]
gap> Coefficients( b, Sqrt(-2) );
[ 0, 1+E(4) ]
gap> f:= AsField( Field( [ Sqrt(-2) ] ), CF(8) );;
gap> # a cycl. field over a non-cycl. field
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1/2+1/2*E(8)-1/2*E(8)^2-1/2*E(8)^3,
  1/2-1/2*E(8)+1/2*E(8)^2+1/2*E(8)^3 ]
gap> Coefficients( b, Sqrt(-2) );
[ E(8)+E(8)^3, E(8)+E(8)^3 ]
gap> f:= Field( [ Sqrt(-2) ] );   # a non-cycl. field over the rationals
NF(8,[ 1, 3 ])
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1, E(8)+E(8)^3 ]
gap> Coefficients( b, Sqrt(-2) );
[ 0, 1 ]
```

60.3.1 ZumbroichBase

▷ `ZumbroichBase(n, m)` (function)

Let `n` and `m` be positive integers, such that `m` divides `n`. `ZumbroichBase` returns the set of exponents i for which `E(n)^i` belongs to the (generalized) Zumbroich basis of the cyclotomic field $\mathbb{Q}_n$, viewed as a vector space over $\mathbb{Q}_m$.

This basis is defined as follows. Let P denote the set of prime divisors of `n`, $n = \prod_{p\in P} p^{\nu_p}$, and $m = \prod_{p\in P} p^{\mu_p}$ with $\mu_p \leq \nu_p$. Let $e_l = \mathtt{E}(l)$ for any positive integer l, and $\{e^j_{n_1}\}_{j\in J} \otimes \{e^k_{n_2}\}_{k\in K} = \{e^j_{n_1} \cdot e^k_{n_2}\}_{j\in J,k\in K}$.

Then the basis is

$$B_{n,m} = \bigotimes_{p\in P} \bigotimes_{k=\mu_p}^{\nu_p-1} \{e^j_{p^{k+1}}\}_{j\in J_{k,p}}$$

where $J_{k,p} =$

$$\begin{array}{lll} \{0\} & ; & k=0, p=2 \\ \{0,1\} & ; & k>0, p=2 \\ \{1,\ldots,p-1\} & ; & k=0, p\neq 2 \\ \{-(p-1)/2,\ldots,(p-1)/2\} & ; & k>0, p\neq 2 \end{array}$$

$B_{n,1}$ is equal to the basis of $\mathbb{Q}_n$ over the rationals which is introduced in [Zum89]. Also the conversion of arbitrary sums of roots of unity into its basis representation, and the reduction to the minimal cyclotomic field are described in this thesis. (Note that the notation here is slightly different from that there.)

$B_{n,m}$ consists of roots of unity, it is an integral basis (that is, exactly the integral elements in $\mathbb{Q}_n$ have integral coefficients w.r.t. $B_{n,m}$, cf. `IsIntegralCyclotomic` (18.1.4)), it is a normal basis for squarefree n and closed under complex conjugation for odd n.

Note: For $n \equiv 2 \pmod 4$, we have `ZumbroichBase(n, 1) = 2 * ZumbroichBase(n/2, 1)` and `List( ZumbroichBase(n, 1), x -> E(n)^x ) = List( ZumbroichBase(n/2, 1), x -> E(n/2)^x )`.

Example

```
gap> ZumbroichBase( 15, 1 ); ZumbroichBase( 12, 3 );
[ 1, 2, 4, 7, 8, 11, 13, 14 ]
[ 0, 3 ]
gap> ZumbroichBase( 10, 2 ); ZumbroichBase( 32, 4 );
[ 2, 4, 6, 8 ]
[ 0, 1, 2, 3, 4, 5, 6, 7 ]
```

60.3.2 LenstraBase

▷ `LenstraBase(n, stabilizer, super, m)` (function)

Let `n` and `m` be positive integers such that `m` divides `n`, `stabilizer` be a list of prime residues modulo `n`, which describes a subfield of the `n`-th cyclotomic field (see `GaloisStabilizer` (60.2.5)), and `super` be a list representing a supergroup of the group given by `stabilizer`.

`LenstraBase` returns a list $[b_1, b_2, \ldots, b_k]$ of lists, each b_i consisting of integers such that the elements $\sum_{j \in b_i} \mathtt{E(n)}^j$ form a basis of the abelian number field `NF( n, stabilizer )`, as a vector space over the `m`-th cyclotomic field (see `AbelianNumberField` (60.1.2)).

This basis is an integral basis, that is, exactly the integral elements in `NF( n, stabilizer )` have integral coefficients. (For details about this basis, see [Bre97].)

If possible then the result is chosen such that the group described by `super` acts on it, consistently with the action of `stabilizer`, i.e., each orbit of `super` is a union of orbits of `stabilizer`. (A usual case is `super = stabilizer`, so there is no additional condition.

Note: The b_i are in general not sets, since for `stabilizer = super`, the first entry is always an element of `ZumbroichBase( n, m )`; this property is used by `NF` (60.1.2) and `Coefficients` (61.6.3) (see 60.3).

`stabilizer` must not contain the stabilizer of a proper cyclotomic subfield of the `n`-th cyclotomic field, i.e., the result must describe a basis for a field with conductor `n`.

Example

```
gap> LenstraBase( 24, [ 1, 19 ], [ 1, 19 ], 1 );
[ [ 1, 19 ], [ 8 ], [ 11, 17 ], [ 16 ] ]
gap> LenstraBase( 24, [ 1, 19 ], [ 1, 5, 19, 23 ], 1 );
[ [ 1, 19 ], [ 5, 23 ], [ 8 ], [ 16 ] ]
gap> LenstraBase( 15, [ 1, 4 ], PrimeResidues( 15 ), 1 );
[ [ 1, 4 ], [ 2, 8 ], [ 7, 13 ], [ 11, 14 ] ]
```

The first two results describe two bases of the field $\mathbb{Q}_3(\sqrt{6})$, the third result describes a normal basis of $\mathbb{Q}_3(\sqrt{5})$.

60.4 Galois Groups of Abelian Number Fields

The field automorphisms of the cyclotomic field $\mathbb{Q}_n$ (see Chapter 18) are given by the linear maps $*k$ on $\mathbb{Q}_n$ that are defined by $\mathtt{E}(n)^{*k} = \mathtt{E}(n)^k$, where $1 \leq k < n$ and $\mathtt{Gcd}(n,k) = 1$ hold (see `GaloisCyc` (18.5.1)). Note that this action is *not* equal to exponentiation of cyclotomics, i.e., for general cyclotomics z, z^{*k} is different from z^k.

(In GAP, the image of a cyclotomic z under $*k$ can be computed as `GaloisCyc( z,k )`.)

Example

```
gap> ( E(5) + E(5)^4 )^2; GaloisCyc( E(5) + E(5)^4, 2 );
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
E(5)^2+E(5)^3
```

For $\mathtt{Gcd}(n,k) \neq 1$, the map $\mathtt{E}(n) \mapsto \mathtt{E}(n)^k$ does *not* define a field automorphism of $\mathbb{Q}_n$ but only a $\mathbb{Q}$-linear map.

Example

```
gap> GaloisCyc( E(5)+E(5)^4, 5 ); GaloisCyc( ( E(5)+E(5)^4 )^2, 5 );
2
-6
```

60.4.1 GaloisGroup (for abelian number fields)

▷ `GaloisGroup(`F`)` (method)

The Galois group $Gal(\mathbb{Q}_n,\mathbb{Q})$ of the field extension $\mathbb{Q}_n/\mathbb{Q}$ is isomorphic to the group $(\mathbb{Z}/n\mathbb{Z})^*$ of prime residues modulo n, via the isomorphism $(\mathbb{Z}/n\mathbb{Z})^* \to Gal(\mathbb{Q}_n,\mathbb{Q})$ that is defined by $k+n\mathbb{Z} \mapsto (z \mapsto z^{*k})$.

The Galois group of the field extension $\mathbb{Q}_n/L$ with any abelian number field $L \subseteq \mathbb{Q}_n$ is simply the factor group of $Gal(\mathbb{Q}_n,\mathbb{Q})$ modulo the stabilizer of L, and the Galois group of L/L', with L' an abelian number field contained in L, is the subgroup in this group that stabilizes L'. These groups are easily described in terms of $(\mathbb{Z}/n\mathbb{Z})^*$. Generators of $(\mathbb{Z}/n\mathbb{Z})^*$ can be computed using `GeneratorsPrimeResidues` (15.2.4).

In GAP, a field extension L/L' is given by the field object L with `LeftActingDomain` (57.1.11) value L' (see 60.3).

Example

```
gap> f:= CF(15);
CF(15)
gap> g:= GaloisGroup( f );
<group with 2 generators>
gap> Size( g ); IsCyclic( g ); IsAbelian( g );
8
false
true
gap> Action( g, NormalBase( f ), OnPoints );
Group([ (1,6)(2,4)(3,8)(5,7), (1,4,3,7)(2,8,5,6) ])
```

The following example shows Galois groups of a cyclotomic field and of a proper subfield that is not a cyclotomic field.

Example

```
gap> gens1:= GeneratorsOfGroup( GaloisGroup( CF(5) ) );
[ ANFAutomorphism( CF(5), 2 ) ]
gap> gens2:= GeneratorsOfGroup( GaloisGroup( Field( Sqrt(5) ) ) );
[ ANFAutomorphism( NF(5,[ 1, 4 ]), 2 ) ]
gap> Order( gens1[1] );  Order( gens2[1] );
4
```

```
2
gap> Sqrt(5)^gens1[1] = Sqrt(5)^gens2[1];
true
```

The following example shows the Galois group of a cyclotomic field over a non-cyclotomic field.

Example

```
gap> g:= GaloisGroup( AsField( Field( [ Sqrt(5) ] ), CF(5) ) );
<group with 1 generators>
gap> gens:= GeneratorsOfGroup( g );
[ ANFAutomorphism( AsField( NF(5,[ 1, 4 ]), CF(5) ), 4 ) ]
gap> x:= last[1];;  x^2;
IdentityMapping( AsField( NF(5,[ 1, 4 ]), CF(5) ) )
```

60.4.2 ANFAutomorphism

▷ `ANFAutomorphism(`*F*`,` *k*`)` (function)

Let *F* be an abelian number field and *k* be an integer that is coprime to the conductor (see `Conductor` (18.1.7)) of *F*. Then `ANFAutomorphism` returns the automorphism of *F* that is defined as the linear extension of the map that raises each root of unity in *F* to its *k*-th power.

Example

```
gap> f:= CF(25);
CF(25)
gap> alpha:= ANFAutomorphism( f, 2 );
ANFAutomorphism( CF(25), 2 )
gap> alpha^2;
ANFAutomorphism( CF(25), 4 )
gap> Order( alpha );
20
gap> E(5)^alpha;
E(5)^2
```

60.5 Gaussians

60.5.1 GaussianIntegers

▷ `GaussianIntegers` (global variable)

`GaussianIntegers` is the ring $\mathbb{Z}[\sqrt{-1}]$ of Gaussian integers. This is a subring of the cyclotomic field `GaussianRationals` (60.1.3).

60.5.2 IsGaussianIntegers

▷ `IsGaussianIntegers(`*obj*`)` (Category)

is the defining category for the domain `GaussianIntegers` (60.5.1).

Chapter 61

Vector Spaces

61.1 IsLeftVectorSpace (Filter)

61.1.1 IsLeftVectorSpace

▷ `IsLeftVectorSpace(V)` (Category)
▷ `IsVectorSpace(V)` (Category)

A *vector space* in GAP is a free left module (see `IsFreeLeftModule` (57.3.1)) over a division ring (see Chapter 58).

Whenever we talk about an F-vector space `V` then `V` is an additive group (see `IsAdditiveGroup` (55.1.6)) on which the division ring F acts via multiplication from the left such that this action and the addition in `V` are left and right distributive. The division ring F can be accessed as value of the attribute `LeftActingDomain` (57.1.11).

Vector spaces in GAP are always *left* vector spaces, `IsLeftVectorSpace` and `IsVectorSpace` are synonyms.

61.2 Constructing Vector Spaces

61.2.1 VectorSpace

▷ `VectorSpace(F, gens[, zero][, "basis"])` (function)

For a field `F` and a collection *gens* of vectors, `VectorSpace` returns the `F`-vector space spanned by the elements in *gens*.

The optional argument *zero* can be used to specify the zero element of the space; *zero must* be given if *gens* is empty. The optional string `"basis"` indicates that *gens* is known to be linearly independent over `F`, in particular the dimension of the vector space is immediately set; note that `Basis` (61.5.2) need *not* return the basis formed by *gens* if the string `"basis"` is given as an argument.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );
<vector space over Rationals, with 2 generators>
```

61.2.2 Subspace

▷ Subspace(*V, gens[, "basis"]*) (function)
▷ SubspaceNC(*V, gens[, "basis"]*) (function)

For an F-vector space *V* and a list or collection *gens* that is a subset of *V*, Subspace returns the F-vector space spanned by *gens*; if *gens* is empty then the trivial subspace (see TrivialSubspace (61.3.2)) of *V* is returned. The parent (see 31.7) of the returned vector space is set to *V*.

SubspaceNC does the same as Subspace, except that it omits the check whether *gens* is a subset of *V*.

The optional string *"basis"* indicates that *gens* is known to be linearly independent over F. In this case the dimension of the subspace is immediately set, and both Subspace and SubspaceNC do *not* check whether *gens* really is linearly independent and whether *gens* is a subset of *V*.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );;
gap> W:= Subspace( V, [ [ 0, 1, 2 ] ] );
<vector space over Rationals, with 1 generators>
```

61.2.3 AsVectorSpace

▷ AsVectorSpace(*F, D*) (operation)

Let *F* be a division ring and *D* a domain. If the elements in *D* form an *F*-vector space then AsVectorSpace returns this *F*-vector space, otherwise fail is returned.

AsVectorSpace can be used for example to view a given vector space as a vector space over a smaller or larger division ring.

Example

```
gap> V:= FullRowSpace( GF( 27 ), 3 );
( GF(3^3)^3 )
gap> Dimension( V );  LeftActingDomain( V );
3
GF(3^3)
gap> W:= AsVectorSpace( GF( 3 ), V );
<vector space over GF(3), with 9 generators>
gap> Dimension( W );  LeftActingDomain( W );
9
GF(3)
gap> AsVectorSpace( GF( 9 ), V );
fail
```

61.2.4 AsSubspace

▷ AsSubspace(*V, U*) (operation)

Let *V* be an F-vector space, and *U* a collection. If *U* is a subset of *V* such that the elements of *U* form an F-vector space then AsSubspace returns this vector space, with parent set to *V* (see AsVectorSpace (61.2.3)). Otherwise fail is returned.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );;
gap> W:= VectorSpace( Rationals, [ [ 1/2, 1/2, 1/2 ] ] );;
```

```
gap> U:= AsSubspace( V, W );
<vector space over Rationals, with 1 generators>
gap> Parent( U ) = V;
true
gap> AsSubspace( V, [ [ 1, 1, 1 ] ] );
fail
```

61.3 Operations and Attributes for Vector Spaces

61.3.1 GeneratorsOfLeftVectorSpace

▷ `GeneratorsOfLeftVectorSpace(V)` (attribute)
▷ `GeneratorsOfVectorSpace(V)` (attribute)

For an F-vector space V, `GeneratorsOfLeftVectorSpace` returns a list of vectors in V that generate V as an F-vector space.

Example

```
gap> GeneratorsOfVectorSpace( FullRowSpace( Rationals, 3 ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
```

61.3.2 TrivialSubspace

▷ `TrivialSubspace(V)` (attribute)

For a vector space V, `TrivialSubspace` returns the subspace of V that consists of the zero vector in V.

Example

```
gap> V:= GF(3)^3;;
gap> triv:= TrivialSubspace( V );
<vector space over GF(3), with 0 generators>
gap> AsSet( triv );
[ [ 0*Z(3), 0*Z(3), 0*Z(3) ] ]
```

61.4 Domains of Subspaces of Vector Spaces

61.4.1 Subspaces

▷ `Subspaces(V[, k])` (attribute)

Called with a finite vector space v, `Subspaces` returns the domain of all subspaces of V.

Called with V and a nonnegative integer k, `Subspaces` returns the domain of all k-dimensional subspaces of V.

Special `Size` (30.4.6) and `Iterator` (30.8.1) methods are provided for these domains.

61.4.2 IsSubspacesVectorSpace

▷ `IsSubspacesVectorSpace(D)` (Category)

The domain of all subspaces of a (finite) vector space or of all subspaces of fixed dimension, as returned by Subspaces (61.4.1) (see Subspaces (61.4.1)) lies in the category IsSubspacesVectorSpace.

Example

```
gap> D:= Subspaces( GF(3)^3 );
Subspaces( ( GF(3)^3 ) )
gap> Size( D );
28
gap> iter:= Iterator( D );;
gap> NextIterator( iter );
<vector space over GF(3), with 0 generators>
gap> NextIterator( iter );
<vector space of dimension 1 over GF(3)>
gap> IsSubspacesVectorSpace( D );
true
```

61.5 Bases of Vector Spaces

In GAP, a *basis* of a free left F-module V is a list of vectors $B = [v_1, v_2, \ldots, v_n]$ in V such that V is generated as a left F-module by these vectors and such that B is linearly independent over F. The integer n is the dimension of V (see Dimension (57.3.3)). In particular, as each basis is a list (see Chapter 21), it has a length (see Length (21.17.5)), and the i-th vector of B can be accessed as $B[i]$.

Example

```
gap> V:= Rationals^3;
( Rationals^3 )
gap> B:= Basis( V );
CanonicalBasis( ( Rationals^3 ) )
gap> Length( B );
3
gap> B[1];
[ 1, 0, 0 ]
```

The operations described below make sense only for bases of *finite* dimensional vector spaces. (In practice this means that the vector spaces must be *low* dimensional, that is, the dimension should not exceed a few hundred.)

Besides the basic operations for lists (see 21.2), the *basic operations for bases* are BasisVectors (61.6.1), Coefficients (61.6.3), LinearCombination (61.6.4), and UnderlyingLeftModule (61.6.2). These and other operations for arbitrary bases are described in 61.6.

For special kinds of bases, further operations are defined (see 61.7).

GAP supports the following three kinds of bases.

Relative bases delegate the work to other bases of the same free left module, via basechange matrices (see RelativeBasis (61.5.4)).

Bases handled by nice bases delegate the work to bases of isomorphic left modules over the same left acting domain (see 61.11).

Finally, of course there must be bases in GAP that really do the work.

For example, in the case of a Gaussian row or matrix space *V* (see 61.9), Basis(*V*) is a semi-echelonized basis (see IsSemiEchelonized (61.9.7)) that uses Gaussian elimination; such a basis is of the third kind. Basis(*V*, *vectors*) is either semi-echelonized or a relative basis. Other examples of bases of the third kind are canonical bases of finite fields and of abelian number fields.

Bases handled by nice bases are described in 61.11. Examples are non-Gaussian row and matrix spaces, and subspaces of finite fields and abelian number fields that are themselves not fields.

61.5.1 IsBasis

▷ IsBasis(*obj*) (Category)

In GAP, a *basis* of a free left module is an object that knows how to compute coefficients w.r.t. its basis vectors (see Coefficients (61.6.3)). Bases are constructed by Basis (61.5.2). Each basis is an immutable list, the i-th entry being the i-th basis vector.

(See 61.8 for mutable bases.)

Example

```
gap> V:= GF(2)^2;;
gap> B:= Basis( V );;
gap> IsBasis( B );
true
gap> IsBasis( [ [ 1, 0 ], [ 0, 1 ] ] );
false
gap> IsBasis( Basis( Rationals^2, [ [ 1, 0 ], [ 0, 1 ] ] ) );
true
```

61.5.2 Basis

▷ Basis(*V[, vectors]*) (attribute)
▷ BasisNC(*V, vectors*) (operation)

Called with a free left F-module *V* as the only argument, Basis returns an F-basis of *V* whose vectors are not further specified.

If additionally a list *vectors* of vectors in *V* is given that forms an F-basis of *V* then Basis returns this basis; if *vectors* is not linearly independent over F or does not generate *V* as a free left F-module then fail is returned.

BasisNC does the same as the two argument version of Basis, except that it does not check whether *vectors* form a basis.

If no basis vectors are prescribed then Basis need not compute basis vectors; in this case, the vectors are computed in the first call to BasisVectors (61.6.1).

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V );
SemiEchelonBasis( <vector space over Rationals, with
2 generators>, ... )
gap> BasisVectors( B );
[ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ]
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 3, 2, 30 ] ] );
Basis( <vector space over Rationals, with 2 generators>,
[ [ 1, 2, 7 ], [ 3, 2, 30 ] ] )
gap> Basis( V, [ [ 1, 2, 3 ] ] );
fail
```

61.5.3 CanonicalBasis

▷ `CanonicalBasis(V)` (attribute)

If the vector space `V` supports a *canonical basis* then `CanonicalBasis` returns this basis, otherwise `fail` is returned.

The defining property of a canonical basis is that its vectors are uniquely determined by the vector space. If canonical bases exist for two vector spaces over the same left acting domain (see `LeftActingDomain` (57.1.11)) then the equality of these vector spaces can be decided by comparing the canonical bases.

The exact meaning of a canonical basis depends on the type of `V`. Canonical bases are defined for example for Gaussian row and matrix spaces (see 61.9).

If one designs a new kind of vector spaces (see 61.12) and defines a canonical basis for these spaces then the `CanonicalBasis` method one installs (see `InstallMethod` (78.2.1)) must *not* call `Basis` (61.5.2). On the other hand, one probably should install a `Basis` (61.5.2) method that simply calls `CanonicalBasis`, the value of the method (see 78.2 and 78.3) being `CANONICAL_BASIS_FLAGS`.

Example

```
gap> vecs:= [ [ 1, 2, 3 ], [ 1, 1, 1 ], [ 1, 1, 1 ] ];;
gap> V:= VectorSpace( Rationals, vecs );;
gap> B:= CanonicalBasis( V );
CanonicalBasis( <vector space over Rationals, with 3 generators> )
gap> BasisVectors( B );
[ [ 1, 0, -1 ], [ 0, 1, 2 ] ]
```

61.5.4 RelativeBasis

▷ `RelativeBasis(B, vectors)` (operation)

▷ `RelativeBasisNC(B, vectors)` (operation)

A relative basis is a basis of the free left module `V` that delegates the computation of coefficients etc. to another basis of `V` via a basechange matrix.

Let `B` be a basis of the free left module `V`, and *`vectors`* a list of vectors in `V`.

`RelativeBasis` checks whether *`vectors`* form a basis of `V`, and in this case a basis is returned in which *`vectors`* are the basis vectors; otherwise `fail` is returned.

`RelativeBasisNC` does the same, except that it omits the check.

61.6 Operations for Vector Space Bases

61.6.1 BasisVectors

▷ `BasisVectors(B)` (attribute)

For a vector space basis `B`, `BasisVectors` returns the list of basis vectors of `B`. The lists `B` and `BasisVectors( B )` are equal; the main purpose of `BasisVectors` is to provide access to a list of vectors that does *not* know about an underlying vector space.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ] );;
```

```
gap> BasisVectors( B );
[ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ]
```

61.6.2 UnderlyingLeftModule

▷ `UnderlyingLeftModule(`*B*`)` (attribute)

For a basis *B* of a free left module *V*, say, `UnderlyingLeftModule` returns *V*.

The reason why a basis stores a free left module is that otherwise one would have to store the basis vectors and the coefficient domain separately. Storing the module allows one for example to deal with bases whose basis vectors have not yet been computed yet (see `Basis` (61.5.2)); furthermore, in some cases it is convenient to test membership of a vector in the module before computing coefficients w.r.t. a basis.

Example

```
gap> B:= Basis( GF(2)^6 );;  UnderlyingLeftModule( B );
( GF(2)^6 )
```

61.6.3 Coefficients

▷ `Coefficients(`*B, v*`)` (operation)

Let *V* be the underlying left module of the basis *B*, and *v* a vector such that the family of *v* is the elements family of the family of *V*. Then `Coefficients(` *B, v* `)` is the list of coefficients of *v* w.r.t. *B* if *v* lies in *V*, and `fail` otherwise.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ] );;
gap> Coefficients( B, [ 1/2, 1/3, 5 ] );
[ 1/2, -2/3 ]
gap> Coefficients( B, [ 1, 0, 0 ] );
fail
```

61.6.4 LinearCombination

▷ `LinearCombination(`*B, coeff*`)` (operation)

If *B* is a basis object (see `IsBasis` (61.5.1)) or a homogeneous list of length *n*, say, and *coeff* is a row vector of the same length, `LinearCombination` returns the vector $\sum_{i=1}^{n} coeff[i] * B[i]$.

Perhaps the most important usage is the case where *B* forms a basis.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ] );;
gap> LinearCombination( B, [ 1/2, -2/3 ] );
[ 1/2, 1/3, 5 ]
```

61.6.5 EnumeratorByBasis

▷ `EnumeratorByBasis(`*B*`)` (attribute)

For a basis *B* of the free left *F*-module *V* of dimension *n*, say, `EnumeratorByBasis` returns an enumerator that loops over the elements of *V* as linear combinations of the vectors of *B* with coefficients the row vectors in the full row space (see `FullRowSpace` (61.9.4)) of dimension *n* over *F*, in the succession given by the default enumerator of this row space.

Example

```
gap> V:= GF(2)^3;;
gap> enum:= EnumeratorByBasis( CanonicalBasis( V ) );;
gap> Print( enum{ [ 1 .. 4 ] }, "\n" );
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
gap> B:= Basis( V, [ [ 1, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 0 ] ] * Z(2) );;
gap> enum:= EnumeratorByBasis( B );;
gap> Print( enum{ [ 1 .. 4 ] }, "\n" );
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ] ]
```

61.6.6 IteratorByBasis

▷ `IteratorByBasis(`*B*`)` (operation)

For a basis *B* of the free left *F*-module *V* of dimension *n*, say, `IteratorByBasis` returns an iterator that loops over the elements of *V* as linear combinations of the vectors of *B* with coefficients the row vectors in the full row space (see `FullRowSpace` (61.9.4)) of dimension *n* over *F*, in the succession given by the default enumerator of this row space.

Example

```
gap> V:= GF(2)^3;;
gap> iter:= IteratorByBasis( CanonicalBasis( V ) );;
gap> for i in [ 1 .. 4 ] do Print( NextIterator( iter ), "\n" ); od;
[ 0*Z(2), 0*Z(2), 0*Z(2) ]
[ 0*Z(2), 0*Z(2), Z(2)^0 ]
[ 0*Z(2), Z(2)^0, 0*Z(2) ]
[ 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> B:= Basis( V, [ [ 1, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 0 ] ] * Z(2) );;
gap> iter:= IteratorByBasis( B );;
gap> for i in [ 1 .. 4 ] do Print( NextIterator( iter ), "\n" ); od;
[ 0*Z(2), 0*Z(2), 0*Z(2) ]
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
[ Z(2)^0, Z(2)^0, 0*Z(2) ]
[ 0*Z(2), Z(2)^0, 0*Z(2) ]
```

61.7 Operations for Special Kinds of Bases

61.7.1 IsCanonicalBasis

▷ `IsCanonicalBasis(`*B*`)` (property)

If the underlying free left module V of the basis B supports a canonical basis (see CanonicalBasis (61.5.3)) then IsCanonicalBasis returns true if B is equal to the canonical basis of V, and false otherwise.

61.7.2 IsIntegralBasis

▷ IsIntegralBasis(*B*) (property)

Let B be an S-basis of a *field* F, say, for a subfield S of F, and let R and M be the rings of algebraic integers in S and F, respectively. IsIntegralBasis returns true if B is also an R-basis of M, and false otherwise.

61.7.3 IsNormalBasis

▷ IsNormalBasis(*B*) (property)

Let B be an S-basis of a *field* F, say, for a subfield S of F. IsNormalBasis returns true if B is invariant under the Galois group (see GaloisGroup (58.3.1)) of the field extension F/S, and false otherwise.

Example

```
gap> B:= CanonicalBasis( GaussianRationals );
CanonicalBasis( GaussianRationals )
gap> IsIntegralBasis( B );  IsNormalBasis( B );
true
false
```

61.8 Mutable Bases

It is useful to have a *mutable basis* of a free module when successively closures with new vectors are formed, since one does not want to create a new module and a corresponding basis for each step.

Note that the situation here is different from the situation with stabilizer chains, which are (mutable or immutable) records that do not need to know about the groups they describe, whereas each (immutable) basis stores the underlying left module (see UnderlyingLeftModule (61.6.2)).

So immutable bases and mutable bases are different categories of objects. The only thing they have in common is that one can ask both for their basis vectors and for the coefficients of a given vector.

Since Immutable produces an immutable copy of any GAP object, it would in principle be possible to construct a mutable basis that is in fact immutable. In the sequel, we will deal only with mutable bases that are in fact *mutable* GAP objects, hence these objects are unable to store attribute values.

Basic operations for immutable bases are NrBasisVectors (61.8.3), IsContainedInSpan (61.8.5), CloseMutableBasis (61.8.6), ImmutableBasis (61.8.4), Coefficients (61.6.3), and BasisVectors (61.6.1). ShallowCopy (12.7.1) for a mutable basis returns a mutable plain list containing the current basis vectors.

Since mutable bases do not admit arbitrary changes of their lists of basis vectors, a mutable basis is *not* a list. It is, however, a collection, more precisely its family (see 13.1) equals the family of its collection of basis vectors.

Mutable bases can be constructed with MutableBasis.

Similar to the situation with bases (cf. 61.5), GAP supports the following three kinds of mutable bases.

The *generic method* of MutableBasis returns a mutable basis that simply stores an immutable basis; clearly one wants to avoid this whenever possible with reasonable effort.

There are mutable bases that store a mutable basis for a nicer module. Note that this is meaningful only if the mechanism of computing nice and ugly vectors (see 61.11) is invariant under closures of the basis; this is the case for example if the vectors are matrices, Lie objects, or elements of structure constants algebras.

There are mutable bases that use special information to perform their tasks; examples are mutable bases of Gaussian row and matrix spaces.

61.8.1 IsMutableBasis

▷ IsMutableBasis(*MB*) (Category)

Every mutable basis lies in the category IsMutableBasis.

61.8.2 MutableBasis

▷ MutableBasis(*R, vectors[, zero]*) (operation)

MutableBasis returns a mutable basis for the *R*-free module generated by the vectors in the list *vectors*. The optional argument *zero* is the zero vector of the module; it must be given if *vectors* is empty.

Note that *vectors* will in general *not* be the basis vectors of the mutable basis!

Example

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 2, 3 ], [ 0, 1, 0 ] ] );
<mutable basis over Rationals, 2 vectors>
```

61.8.3 NrBasisVectors

▷ NrBasisVectors(*MB*) (operation)

For a mutable basis *MB*, NrBasisVectors returns the current number of basis vectors of *MB*. Note that this operation is *not* an attribute, as it makes no sense to store the value. NrBasisVectors is used mainly as an equivalent of Dimension for the underlying left module in the case of immutable bases.

Example

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 1], [ 2, 2 ] ] );;
gap> NrBasisVectors( MB );
1
```

61.8.4 ImmutableBasis

▷ ImmutableBasis(*MB[, V]*) (operation)

ImmutableBasis returns the immutable basis *B*, say, with the same basis vectors as in the mutable basis *MB*.

If the second argument *V* is present then *V* is the value of `UnderlyingLeftModule` (61.6.2) for *B*. The second variant is used mainly for the case that one knows the module for the desired basis in advance, and if it has a nicer structure than the module known to *MB*, for example if it is an algebra.

Example

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 1 ], [ 2, 2 ] ] );;
gap> B:= ImmutableBasis( MB );
SemiEchelonBasis( <vector space of dimension 1 over Rationals>,
[ [ 1, 1 ] ] )
gap> UnderlyingLeftModule( B );
<vector space of dimension 1 over Rationals>
```

61.8.5 IsContainedInSpan

▷ `IsContainedInSpan(`*MB, v*`)` (operation)

For a mutable basis *MB* over the coefficient ring *R*, say, and a vector *v*, `IsContainedInSpan` returns `true` is *v* lies in the *R*-span of the current basis vectors of *MB*, and `false` otherwise.

61.8.6 CloseMutableBasis

▷ `CloseMutableBasis(`*MB, v*`)` (operation)

For a mutable basis *MB* over the coefficient ring *R*, say, and a vector *v*, `CloseMutableBasis` changes *MB* such that afterwards it describes the *R*-span of the former basis vectors together with *v*.

Note that if *v* enlarges the dimension then this does in general *not* mean that *v* is simply added to the basis vectors of *MB*. Usually a linear combination of *v* and the other basis vectors is added, and also the old basis vectors may be modified, for example in order to keep the list of basis vectors echelonized (see `IsSemiEchelonized` (61.9.7)).

Example

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 1, 3 ], [ 2, 2, 1 ] ] );
<mutable basis over Rationals, 2 vectors>
gap> IsContainedInSpan( MB, [ 1, 0, 0 ] );
false
gap> CloseMutableBasis( MB, [ 1, 0, 0 ] );
gap> MB;
<mutable basis over Rationals, 3 vectors>
gap> IsContainedInSpan( MB, [ 1, 0, 0 ] );
true
```

61.9 Row and Matrix Spaces

61.9.1 IsRowSpace

▷ `IsRowSpace(`*V*`)` (function)

A *row space* in GAP is a vector space that consists of row vectors (see Chapter 23).

61.9.2 IsMatrixSpace

▷ `IsMatrixSpace(V)` (function)

A *matrix space* in GAP is a vector space that consists of matrices (see Chapter 24).

61.9.3 IsGaussianSpace

▷ `IsGaussianSpace(V)` (function)

The filter `IsGaussianSpace` (see 13.2) for the row space (see `IsRowSpace` (61.9.1)) or matrix space (see `IsMatrixSpace` (61.9.2)) `V` over the field F, say, indicates that the entries of all row vectors or matrices in `V`, respectively, are all contained in F. In this case, `V` is called a *Gaussian* vector space. Bases for Gaussian spaces can be computed using Gaussian elimination for a given list of vector space generators.

Example

```
gap> mats:= [ [[1,1],[2,2]], [[3,4],[0,1]] ];;
gap> V:= VectorSpace( Rationals, mats );;
gap> IsGaussianSpace( V );
true
gap> mats[1][1][1]:= E(4);;   # an element in an extension field
gap> V:= VectorSpace( Rationals, mats );;
gap> IsGaussianSpace( V );
false
gap> V:= VectorSpace( Field( Rationals, [ E(4) ] ), mats );;
gap> IsGaussianSpace( V );
true
```

61.9.4 FullRowSpace

▷ `FullRowSpace(F, n)` (function)
▷ `\^(F, n)` (method)

For a field `F` and a nonnegative integer `n`, `FullRowSpace` returns the `F`-vector space that consists of all row vectors (see `IsRowVector` (23.1.1)) of length `n` with entries in `F`.

An alternative to construct this vector space is via `F^n`.

Example

```
gap> FullRowSpace( GF( 9 ), 3 );
( GF(3^2)^3 )
gap> GF(9)^3;           # the same as above
( GF(3^2)^3 )
```

61.9.5 FullMatrixSpace

▷ `FullMatrixSpace(F, m, n)` (function)
▷ `\^(F, dims)` (method)

For a field `F` and two positive integers `m` and `n`, `FullMatrixSpace` returns the `F`-vector space that consists of all `m` by `n` matrices (see `IsMatrix` (24.2.1)) with entries in `F`.

If *m* = *n* then the result is in fact an algebra (see FullMatrixAlgebra (62.5.4)).

An alternative to construct this vector space is via *F*^[*m*,*n*].

Example

```
gap> FullMatrixSpace( GF(2), 4, 5 );
( GF(2)^[ 4, 5 ] )
gap> GF(2)^[ 4, 5 ];    # the same as above
( GF(2)^[ 4, 5 ] )
```

61.9.6 DimensionOfVectors

▷ DimensionOfVectors(*M*) (attribute)

For a left module *M* that consists of row vectors (see IsRowModule (57.3.6)), DimensionOfVectors returns the common length of all row vectors in *M*. For a left module *M* that consists of matrices (see IsMatrixModule (57.3.7)), DimensionOfVectors returns the common matrix dimensions (see DimensionsMat (24.4.1)) of all matrices in *M*.

Example

```
gap> DimensionOfVectors( GF(2)^5 );
5
gap> DimensionOfVectors( GF(2)^[2,3] );
[ 2, 3 ]
```

61.9.7 IsSemiEchelonized

▷ IsSemiEchelonized(*B*) (property)

Let *B* be a basis of a Gaussian row or matrix space *V*, say (see IsGaussianSpace (61.9.3)) over the field *F*.

If *V* is a row space then *B* is semi-echelonized if the matrix formed by its basis vectors has the property that the first nonzero element in each row is the identity of *F*, and all values exactly below these pivot elements are the zero of *F* (cf. SemiEchelonMat (24.10.1)).

If *V* is a matrix space then *B* is semi-echelonized if the matrix obtained by replacing each basis vector by the concatenation of its rows is semi-echelonized (see above, cf. SemiEchelonMats (24.10.4)).

Example

```
gap> V:= GF(2)^2;;
gap> B1:= Basis( V, [ [ 0, 1 ], [ 1, 0 ] ] * Z(2) );;
gap> IsSemiEchelonized( B1 );
true
gap> B2:= Basis( V, [ [ 0, 1 ], [ 1, 1 ] ] * Z(2) );;
gap> IsSemiEchelonized( B2 );
false
```

61.9.8 SemiEchelonBasis

▷ SemiEchelonBasis(*V[, vectors]*) (attribute)

▷ SemiEchelonBasisNC(*V, vectors*) (operation)

Let *V* be a Gaussian row or matrix vector space over the field *F* (see IsGaussianSpace (61.9.3), IsRowSpace (61.9.1), IsMatrixSpace (61.9.2)).

Called with *V* as the only argument, SemiEchelonBasis returns a basis of *V* that has the property IsSemiEchelonized (61.9.7).

If additionally a list *vectors* of vectors in *V* is given that forms a semi-echelonized basis of *V* then SemiEchelonBasis returns this basis; if *vectors* do not form a basis of *V* then fail is returned.

SemiEchelonBasisNC does the same as the two argument version of SemiEchelonBasis, except that it is not checked whether *vectors* form a semi-echelonized basis.

Example

```
gap> V:= GF(2)^2;;
gap> B:= SemiEchelonBasis( V );
SemiEchelonBasis( ( GF(2)^2 ), ... )
gap> Print( BasisVectors( B ), "\n" );
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
gap> B:= SemiEchelonBasis( V, [ [ 1, 1 ], [ 0, 1 ] ] * Z(2) );
SemiEchelonBasis( ( GF(2)^2 ), <an immutable 2x2 matrix over GF2> )
gap> Print( BasisVectors( B ), "\n" );
[ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ]
gap> Coefficients( B, [ 0, 1 ] * Z(2) );
[ 0*Z(2), Z(2)^0 ]
gap> Coefficients( B, [ 1, 0 ] * Z(2) );
[ Z(2)^0, Z(2)^0 ]
gap> SemiEchelonBasis( V, [ [ 0, 1 ], [ 1, 1 ] ] * Z(2) );
fail
```

61.9.9 IsCanonicalBasisFullRowModule

▷ IsCanonicalBasisFullRowModule(*B*) (property)

IsCanonicalBasisFullRowModule returns true if *B* is the canonical basis (see IsCanonicalBasis (61.7.1)) of a full row module (see IsFullRowModule (57.3.8)), and false otherwise.

The *canonical basis* of a Gaussian row space is defined as the unique semi-echelonized (see IsSemiEchelonized (61.9.7)) basis with the additional property that for $j > i$ the position of the pivot of row j is bigger than the position of the pivot of row i, and that each pivot column contains exactly one nonzero entry.

61.9.10 IsCanonicalBasisFullMatrixModule

▷ IsCanonicalBasisFullMatrixModule(*B*) (property)

IsCanonicalBasisFullMatrixModule returns true if *B* is the canonical basis (see IsCanonicalBasis (61.7.1)) of a full matrix module (see IsFullMatrixModule (57.3.10)), and false otherwise.

The *canonical basis* of a Gaussian matrix space is defined as the unique semi-echelonized (see IsSemiEchelonized (61.9.7)) basis for which the list of concatenations of the basis vectors forms the canonical basis of the corresponding Gaussian row space.

61.9.11 NormedRowVectors

▷ NormedRowVectors(*V*) (attribute)

For a finite Gaussian row space *V* (see IsRowSpace (61.9.1), IsGaussianSpace (61.9.3)), NormedRowVectors returns a list of those nonzero vectors in *V* that have a one in the first nonzero component.

The result list can be used as action domain for the action of a matrix group via OnLines (41.2.12), which yields the natural action on one-dimensional subspaces of *V* (see also Subspaces (61.4.1)).

Example

```
gap> vecs:= NormedRowVectors( GF(3)^2 );
[ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3)^0 ],
  [ Z(3)^0, Z(3) ] ]
gap> Action( GL(2,3), vecs, OnLines );
Group([ (3,4), (1,2,4) ])
```

61.9.12 SiftedVector

▷ SiftedVector(*B, v*) (operation)

Let *B* be a semi-echelonized basis (see IsSemiEchelonized (61.9.7)) of a Gaussian row or matrix space *V* (see IsGaussianSpace (61.9.3)), and *v* a row vector or matrix, respectively, of the same dimension as the elements in *V*. SiftedVector returns the *residuum* of *v* with respect to *B*, which is obtained by successively cleaning the pivot positions in *v* by subtracting multiples of the basis vectors in *B*. So the result is the zero vector in *V* if and only if *v* lies in *V*.

B may also be a mutable basis (see 61.8) of a Gaussian row or matrix space.

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V );;
gap> SiftedVector( B, [ 1, 2, 8 ] );
[ 0, 0, 1 ]
```

61.10 Vector Space Homomorphisms

Vector space homomorphisms (or *linear mappings*) are defined in Section 32.11. GAP provides special functions to construct a particular linear mapping from images of given elements in the source, from a matrix of coefficients, or as a natural epimorphism.

F-linear mappings with same source and same range can be added, so one can form vector spaces of linear mappings.

61.10.1 LeftModuleGeneralMappingByImages

▷ LeftModuleGeneralMappingByImages(*V, W, gens, imgs*) (operation)

Let *V* and *W* be two left modules over the same left acting domain *R*, say, and *gens* and *imgs* lists (of the same length) of elements in *V* and *W*, respectively. LeftModuleGeneralMappingByImages returns the general mapping with source *V* and range *W* that is defined by mapping the elements in *gens* to the corresponding elements in *imgs*, and taking the *R*-linear closure.

gens need not generate *V* as a left *R*-module, and if the specification does not define a linear mapping then the result will be multi-valued; hence in general it is not a mapping (see IsMapping (32.3.3)).

Example

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [1,2,3], [1,0,1] ] );;
gap> f:= LeftModuleGeneralMappingByImages( V, W,
>                                 [[1,0],[2,0]], [[1,0,1],[1,0,1] ] );
[ [ 1, 0 ], [ 2, 0 ] ] -> [ [ 1, 0, 1 ], [ 1, 0, 1 ] ]
gap> IsMapping( f );
false
```

61.10.2 LeftModuleHomomorphismByImages

▷ LeftModuleHomomorphismByImages(*V, W, gens, imgs*) (function)
▷ LeftModuleHomomorphismByImagesNC(*V, W, gens, imgs*) (operation)

Let *V* and *W* be two left modules over the same left acting domain *R*, say, and *gens* and *imgs* lists (of the same length) of elements in *V* and *W*, respectively. LeftModuleHomomorphismByImages returns the left *R*-module homomorphism with source *V* and range *W* that is defined by mapping the elements in *gens* to the corresponding elements in *imgs*.

If *gens* does not generate *V* or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then fail is returned. For creating a possibly multi-valued mapping from *V* to *W* that respects addition, multiplication, and scalar multiplication, LeftModuleGeneralMappingByImages (61.10.1) can be used.

LeftModuleHomomorphismByImagesNC does the same as LeftModuleHomomorphismByImages, except that it omits all checks.

Example

```
gap> V:=Rationals^2;;
gap> W:=VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> f:=LeftModuleHomomorphismByImages( V, W,
> [ [ 1, 0 ], [ 0, 1 ] ], [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );
[ [ 1, 0 ], [ 0, 1 ] ] -> [ [ 1, 0, 1 ], [ 1, 2, 3 ] ]
gap> Image( f, [1,1] );
[ 2, 2, 4 ]
```

61.10.3 LeftModuleHomomorphismByMatrix

▷ LeftModuleHomomorphismByMatrix(*BS, matrix, BR*) (operation)

Let *BS* and *BR* be bases of the left *R*-modules *V* and *W*, respectively. LeftModuleHomomorphismByMatrix returns the *R*-linear mapping from *V* to *W* that is defined by the matrix *matrix*, as follows. The image of the *i*-th basis vector of *BS* is the linear combination of the basis vectors of *BR* with coefficients the *i*-th row of *matrix*.

Example

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> f:= LeftModuleHomomorphismByMatrix( Basis( V ),
> [ [ 1, 2 ], [ 3, 1 ] ], Basis( W ) );
```

```
<linear mapping by matrix, ( Rationals^
2 ) -> <vector space over Rationals, with 2 generators>>
```

61.10.4 NaturalHomomorphismBySubspace

▷ NaturalHomomorphismBySubspace(*V, W*) (operation)

For an *R*-vector space *V* and a subspace *W* of *V*, NaturalHomomorphismBySubspace returns the *R*-linear mapping that is the natural projection of *V* onto the factor space *V / W*.

Example

```
gap> V:= Rationals^3;;
gap> W:= VectorSpace( Rationals, [ [ 1, 1, 1 ] ] );;
gap> f:= NaturalHomomorphismBySubspace( V, W );
<linear mapping by matrix, ( Rationals^3 ) -> ( Rationals^2 )>
```

61.10.5 Hom

▷ Hom(*F, V, W*) (operation)

For a field *F* and two vector spaces *V* and *W* that can be regarded as *F*-modules (see AsLeftModule (57.1.5)), Hom returns the *F*-vector space of all *F*-linear mappings from *V* to *W*.

Example

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> H:= Hom( Rationals, V, W );
Hom( Rationals, ( Rationals^2 ), <vector space over Rationals, with
2 generators> )
gap> Dimension( H );
4
```

61.10.6 End

▷ End(*F, V*) (operation)

For a field *F* and a vector space *V* that can be regarded as an *F*-module (see AsLeftModule (57.1.5)), End returns the *F*-algebra of all *F*-linear mappings from *V* to *V*.

Example

```
gap> A:= End( Rationals, Rationals^2 );
End( Rationals, ( Rationals^2 ) )
gap> Dimension( A );
4
```

61.10.7 IsFullHomModule

▷ IsFullHomModule(*M*) (property)

A *full hom module* is a module of all *R*-linear mappings between two left *R*-modules. The function Hom (61.10.5) can be used to construct a full hom module.

Example

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> H:= Hom( Rationals, V, W );;
gap> IsFullHomModule( H );
true
```

61.10.8 IsPseudoCanonicalBasisFullHomModule

▷ `IsPseudoCanonicalBasisFullHomModule(`*B*`)` (property)

A basis of a full hom module is called pseudo canonical basis if the matrices of its basis vectors w.r.t. the stored bases of source and range contain exactly one identity entry and otherwise zeros.

Note that this is not a canonical basis (see `CanonicalBasis` (61.5.3)) because it depends on the stored bases of source and range.

Example

```
gap> IsPseudoCanonicalBasisFullHomModule( Basis( H ) );
true
```

61.10.9 IsLinearMappingsModule

▷ `IsLinearMappingsModule(`*V*`)` (filter)

If an F-vector space `V` is in the filter `IsLinearMappingsModule` then this expresses that `V` consists of linear mappings, and that `V` is handled via the mechanism of nice bases (see 61.11), in the following way. Let S and R be the source and the range, respectively, of each mapping in V. Then the `NiceFreeLeftModuleInfo` (61.11.3) value of `V` is a record with the components `basissource` (a basis B_S of S) and `basisrange` (a basis B_R of R), and the `NiceVector` (61.11.2) value of $v \in$ `V` is defined as the matrix of the F-linear mapping v w.r.t. the bases B_S and B_R.

61.11 Vector Spaces Handled By Nice Bases

There are kinds of free R-modules for which efficient computations are possible because the elements are “nice”, for example subspaces of full row modules or of full matrix modules. In other cases, a “nice” canonical basis is known that allows one to do the necessary computations in the corresponding row module, for example algebras given by structure constants.

In many other situations, one knows at least an isomorphism from the given module V to a “nicer” free left module W, in the sense that for each vector in V, the image in W can easily be computed, and analogously for each vector in W, one can compute the preimage in V.

This allows one to delegate computations w.r.t. a basis B, say, of V to the corresponding basis C, say, of W. We call W the *nice free left module* of V, and C the *nice basis* of B. (Note that it may happen that also C delegates questions to a “nicer” basis.) The basis B indicates the intended behaviour by the filter `IsBasisByNiceBasis` (61.11.5), and stores C as value of the attribute `NiceBasis` (61.11.4). V indicates the intended behaviour by the filter `IsHandledByNiceBasis` (61.11.6), and stores W as value of the attribute `NiceFreeLeftModule` (61.11.1).

The bijection between V and W is implemented by the functions NiceVector (61.11.2) and UglyVector (61.11.2); additional data needed to compute images and preimages can be stored as value of NiceFreeLeftModuleInfo (61.11.3).

61.11.1 NiceFreeLeftModule

▷ NiceFreeLeftModule(*V*) (attribute)

For a free left module *V* that is handled via the mechanism of nice bases, this attribute stores the associated free left module to which the tasks are delegated.

61.11.2 NiceVector

▷ NiceVector(*V*, *v*) (operation)
▷ UglyVector(*V*, *r*) (operation)

NiceVector and UglyVector provide the linear bijection between the free left module *V* and *W*:= NiceFreeLeftModule(*V*).

If *v* lies in the elements family of the family of *V* then NiceVector(*v*) is either fail or an element in the elements family of the family of *W*.

If *r* lies in the elements family of the family of *W* then UglyVector(*r*) is either fail or an element in the elements family of the family of *V*.

If *v* lies in *V* (which usually *cannot* be checked without using *W*) then UglyVector(*V*, NiceVector(*V*, *v*)) = *v*. If *r* lies in *W* (which usually *can* be checked) then NiceVector(*V*, UglyVector(*V*, *r*)) = *r*.

(This allows one to implement for example a membership test for *V* using the membership test in *W*.)

61.11.3 NiceFreeLeftModuleInfo

▷ NiceFreeLeftModuleInfo(*V*) (attribute)

For a free left module *V* that is handled via the mechanism of nice bases, this operation has to provide the necessary information (if any) for calls of NiceVector (61.11.2) and UglyVector (61.11.2).

61.11.4 NiceBasis

▷ NiceBasis(*B*) (attribute)

Let *B* be a basis of a free left module *V* that is handled via nice bases. If *B* has no basis vectors stored at the time of the first call to NiceBasis then NiceBasis(*B*) is obtained as Basis(NiceFreeLeftModule(*V*)). If basis vectors are stored then NiceBasis(*B*) is the result of the call of Basis with arguments NiceFreeLeftModule(*V*) and the NiceVector values of the basis vectors of *B*.

Note that the result is fail if and only if the "basis vectors" stored in *B* are in fact not basis vectors.

The attributes GeneratorsOfLeftModule of the underlying left modules of *B* and the result of NiceBasis correspond via NiceVector (61.11.2) and UglyVector (61.11.2).

61.11.5 IsBasisByNiceBasis

▷ IsBasisByNiceBasis(*B*) (Category)

This filter indicates that the basis *B* delegates tasks such as the computation of coefficients (see Coefficients (61.6.3)) to a basis of an isomorphic "nicer" free left module.

61.11.6 IsHandledByNiceBasis

▷ IsHandledByNiceBasis(*M*) (Category)

For a free left module *M* in this category, essentially all operations are performed using a "nicer" free left module, which is usually a row module.

61.12 How to Implement New Kinds of Vector Spaces

61.12.1 DeclareHandlingByNiceBasis

▷ DeclareHandlingByNiceBasis(*name, info*) (function)
▷ InstallHandlingByNiceBasis(*name, record*) (function)

These functions are used to implement a new kind of free left modules that shall be handled via the mechanism of nice bases (see 61.11).

name must be a string, a filter f with this name is created, and a logical implication from f to IsHandledByNiceBasis (61.11.6) is installed.

record must be a record with the following components.

detect
: a function of four arguments R, l, V, and z, where V is a free left module over the ring R with generators the list or collection l, and z is either the zero element of V or false (then l is nonempty); the function returns true if V shall lie in the filter f, and false otherwise; the return value may also be fail, which indicates that V is *not* to be handled via the mechanism of nice bases at all,

NiceFreeLeftModuleInfo
: the NiceFreeLeftModuleInfo method for left modules in f,

NiceVector
: the NiceVector method for left modules V in f; called with V and a vector $v \in V$, this function returns the nice vector r associated with v, and

UglyVector
: the UglyVector (61.11.2) method for left modules V in f; called with V and a vector r in the NiceFreeLeftModule value of V, this function returns the vector $v \in V$ to which r is associated.

The idea is that all one has to do for implementing a new kind of free left modules handled by the mechanism of nice bases is to call DeclareHandlingByNiceBasis and InstallHandlingByNiceBasis, which causes the installation of the necessary methods and adds the pair [*f*,*record*.detect] to the global list NiceBasisFiltersInfo. The LeftModuleByGenerators (57.1.10) methods call CheckForHandlingByNiceBasis (61.12.3), which sets the appropriate filter for the desired left module if applicable.

61.12.2 NiceBasisFiltersInfo

▷ NiceBasisFiltersInfo (global variable)

An overview of all kinds of vector spaces that are currently handled by nice bases is given by the global list NiceBasisFiltersInfo. Examples of such vector spaces are vector spaces of field elements (but not the fields themselves) and non-Gaussian row and matrix spaces (see IsGaussianSpace (61.9.3)).

61.12.3 CheckForHandlingByNiceBasis

▷ CheckForHandlingByNiceBasis(*R, gens, M, zero*) (function)

Whenever a free left module is constructed for which the filter IsHandledByNiceBasis may be useful, CheckForHandlingByNiceBasis should be called. (This is done in the methods for VectorSpaceByGenerators, AlgebraByGenerators, IdealByGenerators etc. in the GAP library.)

The arguments of this function are the coefficient ring *R*, the list *gens* of generators, the constructed module *M* itself, and the zero element *zero* of *M*; if *gens* is nonempty then the *zero* value may also be false.

Chapter 62

Algebras

An algebra is a vector space equipped with a bilinear map (multiplication). This chapter describes the functions in GAP that deal with general algebras and associative algebras.

Algebras in GAP are vector spaces in a natural way. So all the functionality for vector spaces (see Chapter 61) is also applicable to algebras.

62.1 InfoAlgebra (Info Class)

62.1.1 InfoAlgebra

▷ `InfoAlgebra` (info class)

is the info class for the functions dealing with algebras (see 7.4).

62.2 Constructing Algebras by Generators

62.2.1 Algebra

▷ `Algebra(`*`F, gens[, zero][, "basis"]`*`)` (function)

`Algebra(` *`F`*`,` *`gens`* `)` is the algebra over the division ring *`F`*, generated by the vectors in the list *`gens`*.

If there are three arguments, a division ring *`F`* and a list *`gens`* and an element *`zero`*, then `Algebra(` *`F`*`,` *`gens`*`,` *`zero`* `)` is the *`F`*-algebra generated by *`gens`*, with zero element *`zero`*.

If the last argument is the string `"basis"` then the vectors in *`gens`* are known to form a basis of the algebra (as an *`F`*-vector space).

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= Algebra( Rationals, [ m ] );
<algebra over Rationals, with 1 generators>
gap> Dimension( A );
2
```

62.2.2 AlgebraWithOne

▷ AlgebraWithOne(*F, gens[, zero][, "basis"]*) (function)

AlgebraWithOne(*F*, *gens*) is the algebra-with-one over the division ring *F*, generated by the vectors in the list *gens*.

If there are three arguments, a division ring *F* and a list *gens* and an element *zero*, then AlgebraWithOne(*F*, *gens*, *zero*) is the *F*-algebra-with-one generated by *gens*, with zero element *zero*.

If the last argument is the string "basis" then the vectors in *gens* are known to form a basis of the algebra (as an *F*-vector space).

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> Dimension( A );
3
gap> One(A);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
```

62.3 Constructing Algebras as Free Algebras

62.3.1 FreeAlgebra (for ring, rank (and name))

▷ FreeAlgebra(*R, rank[, name]*) (function)
▷ FreeAlgebra(*R, name1, name2, ...*) (function)

is a free (nonassociative) algebra of rank *rank* over the division ring *R*. Here *name*, and *name1*, *name2*, ... are optional strings that can be used to provide names for the generators.

Example

```
gap> A:= FreeAlgebra( Rationals, "a", "b" );
<algebra over Rationals, with 2 generators>
gap> g:= GeneratorsOfAlgebra( A );
[ (1)*a, (1)*b ]
gap> (g[1]*g[2])*((g[2]*g[1])*g[1]);
(1)*((a*b)*((b*a)*a))
```

62.3.2 FreeAlgebraWithOne (for ring, rank (and name))

▷ FreeAlgebraWithOne(*R, rank[, name]*) (function)
▷ FreeAlgebraWithOne(*R, name1, name2, ...*) (function)

is a free (nonassociative) algebra-with-one of rank *rank* over the division ring *R*. Here *name*, and *name1*, *name2*, ... are optional strings that can be used to provide names for the generators.

Example

```
gap> A:= FreeAlgebraWithOne( Rationals, 4, "q" );
<algebra-with-one over Rationals, with 4 generators>
gap> GeneratorsOfAlgebra( A );
[ (1)*<identity ...>, (1)*q.1, (1)*q.2, (1)*q.3, (1)*q.4 ]
```

```
gap> One( A );
(1)*<identity ...>
```

62.3.3 FreeAssociativeAlgebra (for ring, rank (and name))

▷ FreeAssociativeAlgebra(*R, rank[, name]*) (function)
▷ FreeAssociativeAlgebra(*R, name1, name2, ...*) (function)

is a free associative algebra of rank *rank* over the division ring *R*. Here *name*, and *name1*, *name2*, ... are optional strings that can be used to provide names for the generators.

Example

```
gap> A:= FreeAssociativeAlgebra( GF( 5 ), 4, "a" );
<algebra over GF(5), with 4 generators>
```

62.3.4 FreeAssociativeAlgebraWithOne (for ring, rank (and name))

▷ FreeAssociativeAlgebraWithOne(*R, rank[, name]*) (function)
▷ FreeAssociativeAlgebraWithOne(*R, name1, name2, ...*) (function)

is a free associative algebra-with-one of rank *rank* over the division ring *R*. Here *name*, and *name1*, *name2*, ... are optional strings that can be used to provide names for the generators.

Example

```
gap> A:= FreeAssociativeAlgebraWithOne( Rationals, "a", "b", "c" );
<algebra-with-one over Rationals, with 3 generators>
gap> GeneratorsOfAlgebra( A );
[ (1)*<identity ...>, (1)*a, (1)*b, (1)*c ]
gap> One( A );
(1)*<identity ...>
```

62.4 Constructing Algebras by Structure Constants

For an introduction into structure constants and how they are handled by GAP, we refer to Section (**Tutorial: Algebras**) of the user's tutorial.

62.4.1 AlgebraByStructureConstants

▷ AlgebraByStructureConstants(*R, sctable[, nameinfo]*) (function)

returns a free left module *A* over the division ring *R*, with multiplication defined by the structure constants table *sctable*. The optional argument *nameinfo* can be used to prescribe names for the elements of the canonical basis of *A*; it can be either a string *name* (then *name*1, *name*2 etc. are chosen) or a list of strings which are then chosen. The vectors of the canonical basis of *A* correspond to the vectors of the basis given by *sctable*.

It is *not* checked whether the coefficients in *sctable* are really elements in *R*.

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1/2, 1, 2/3, 2 ] );
gap> A:= AlgebraByStructureConstants( Rationals, T );
```

```
<algebra of dimension 2 over Rationals>
gap> b:= BasisVectors( Basis( A ) );;
gap> b[1]^2;
(1/2)*v.1+(2/3)*v.2
gap> b[1]*b[2];
0*v.1
```

62.4.2 StructureConstantsTable

▷ `StructureConstantsTable(`*B*`)` (attribute)

Let *B* be a basis of a free left module *R*, say, that is also a ring. In this case `StructureConstantsTable` returns a structure constants table *T* in sparse representation, as used for structure constants algebras (see Section (**Tutorial: Algebras**) of the GAP User's Tutorial).

If *B* has length *n* then *T* is a list of length $n+2$. The first *n* entries of *T* are lists of length *n*. $T[n+1]$ is one of 1, -1, or 0; in the case of 1 the table is known to be symmetric, in the case of -1 it is known to be antisymmetric, and 0 occurs in all other cases. $T[n+2]$ is the zero element of the coefficient domain.

The coefficients w.r.t. *B* of the product of the *i*-th and *j*-th basis vector of *B* are stored in $T[i][j]$ as a list of length 2; its first entry is the list of positions of nonzero coefficients, the second entry is the list of these coefficients themselves.

The multiplication in an algebra *A* with vector space basis *B* with basis vectors $[v_1,\ldots,v_n]$ is determined by the so-called structure matrices $M_k=[m_{ijk}]_{ij}$, $1\le k\le n$. The M_k are defined by $v_iv_j=\sum_k m_{ijk}v_k$. Let $a=[a_1,\ldots,a_n]$ and $b=[b_1,\ldots,b_n]$. Then

$$\left(\sum_i a_iv_i\right)\left(\sum_j b_jv_j\right)=\sum_{i,j}a_ib_j\,(v_iv_j)=\sum_k\left(\sum_j\left(\sum_i a_im_{ijk}\right)b_j\right)v_k=\sum_k\left(aM_kb^{tr}\right)v_k.$$

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> StructureConstantsTable( Basis( A ) );
[ [ [ [ 1 ], [ 1 ] ], [ [ 2 ], [ 1 ] ], [ [ 3 ], [ 1 ] ],
      [ [ 4 ], [ 1 ] ] ],
  [ [ [ 2 ], [ 1 ] ], [ [ 1 ], [ -1 ] ], [ [ 4 ], [ 1 ] ],
      [ [ 3 ], [ -1 ] ] ],
  [ [ [ 3 ], [ 1 ] ], [ [ 4 ], [ -1 ] ], [ [ 1 ], [ -1 ] ],
      [ [ 2 ], [ 1 ] ] ],
  [ [ [ 4 ], [ 1 ] ], [ [ 3 ], [ 1 ] ], [ [ 2 ], [ -1 ] ],
      [ [ 1 ], [ -1 ] ] ], 0, 0 ]
```

62.4.3 EmptySCTable

▷ `EmptySCTable(`*dim, zero[, flag]*`)` (function)

`EmptySCTable` returns a structure constants table for an algebra of dimension *dim*, describing trivial multiplication. *zero* must be the zero of the coefficients domain. If the multiplication is known to be (anti)commutative then this can be indicated by the optional third argument *flag*, which must be one of the strings `"symmetric"`, `"antisymmetric"`.

For filling up the structure constants table, see `SetEntrySCTable` (62.4.4).

Example

```
gap> EmptySCTable( 2, Zero( GF(5) ), "antisymmetric" );
[ [ [ [  ], [  ] ], [ [  ], [  ] ] ],
  [ [ [  ], [  ] ], [ [  ], [  ] ] ], -1, 0*Z(5) ]
```

62.4.4 SetEntrySCTable

▷ SetEntrySCTable(*T, i, j, list*) (function)

sets the entry of the structure constants table *T* that describes the product of the *i*-th basis element with the *j*-th basis element to the value given by the list *list*.

If *T* is known to be antisymmetric or symmetric then also the value *T*[*j*][*i*] is set.

list must be of the form $[c_{ij}^{k_1}, k_1, c_{ij}^{k_2}, k_2, \ldots]$.

The entries at the odd positions of *list* must be compatible with the zero element stored in *T*. For convenience, these entries may also be rational numbers that are automatically replaced by the corresponding elements in the appropriate prime field in finite characteristic if necessary.

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1/2, 1, 2/3, 2 ] );
gap> T;
[ [ [ [ 1, 2 ], [ 1/2, 2/3 ] ], [ [  ], [  ] ] ],
  [ [ [  ], [  ] ], [ [  ], [  ] ] ], 0, 0 ]
```

62.4.5 GapInputSCTable

▷ GapInputSCTable(*T, varname*) (function)

is a string that describes the structure constants table *T* in terms of EmptySCTable (62.4.3) and SetEntrySCTable (62.4.4). The assignments are made to the variable *varname*.

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );
gap> SetEntrySCTable( T, 2, 1, [ 1, 2 ] );
gap> GapInputSCTable( T, "T" );
"T:= EmptySCTable( 2, 0 );\nSetEntrySCTable( T, 1, 2, [1,2] );\nSetEnt\
rySCTable( T, 2, 1, [1,2] );\n"
```

62.4.6 TestJacobi

▷ TestJacobi(*T*) (function)

tests whether the structure constants table *T* satisfies the Jacobi identity $v_i*(v_j*v_k)+v_j*(v_k*v_i)+v_k*(v_i*v_j)=0$ for all basis vectors v_i of the underlying algebra, where $i \leq j \leq k$. (Thus antisymmetry is assumed.)

The function returns true if the Jacobi identity is satisfied, and a failing triple $[i,j,k]$ otherwise.

Example

```
gap> T:= EmptySCTable( 2, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );;
```

```
gap> TestJacobi( T );
true
```

62.4.7 IdentityFromSCTable

▷ `IdentityFromSCTable(`*T*`)` (function)

Let *T* be a structure constants table of an algebra *A* of dimension *n*. `IdentityFromSCTable( T )` is either `fail` or the vector of length *n* that contains the coefficients of the multiplicative identity of *A* with respect to the basis that belongs to *T*.

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1, 1 ] );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );;
gap> SetEntrySCTable( T, 2, 1, [ 1, 2 ] );;
gap> IdentityFromSCTable( T );
[ 1, 0 ]
```

62.4.8 QuotientFromSCTable

▷ `QuotientFromSCTable(`*T, num, den*`)` (function)

Let *T* be a structure constants table of an algebra *A* of dimension *n*. `QuotientFromSCTable( T )` is either `fail` or the vector of length *n* that contains the coefficients of the quotient of *num* and *den* with respect to the basis that belongs to *T*.

We solve the equation system $num = x * den$. If no solution exists, `fail` is returned.

In terms of the basis *B* with vectors $b_1, \ldots, b_n$ this means for $num = \sum_{i=1}^{n} a_i b_i$, $den = \sum_{i=1}^{n} c_i b_i$, $x = \sum_{i=1}^{n} x_i b_i$ that $a_k = \sum_{i,j} c_i x_j c_{ijk}$ for all k. Here c_{ijk} denotes the structure constants with respect to *B*. This means that (as a vector) $a = xM$ with $M_{jk} = \sum_{i=1}^{n} c_{ijk} c_i$.

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1, 1 ] );;
gap> SetEntrySCTable( T, 2, 1, [ 1, 2 ] );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );;
gap> QuotientFromSCTable( T, [0,1], [1,0] );
[ 0, 1 ]
```

62.5 Some Special Algebras

62.5.1 QuaternionAlgebra

▷ `QuaternionAlgebra(`*F[, a, b]*`)` (function)

Returns: a quaternion algebra over *F*, with parameters `a` and `b`.

Let *F* be a field or a list of field elements, let *F* be the field generated by *F*, and let `a` and `b` two elements in *F*. `QuaternionAlgebra` returns a quaternion algebra over *F*, with parameters `a` and `b`, i.e., a four-dimensional associative *F*-algebra with basis (e, i, j, k) and multiplication defined by $ee = e$, $ei = ie = i$, $ej = je = j$, $ek = ke = k$, $ii = \mathtt{a}e$, $ij = -ji = k$, $ik = -ki = \mathtt{a}j$, $jj = \mathtt{b}e$, $jk = -kj = \mathtt{b}i$, $kk = -\mathtt{a}\mathtt{b}e$. The default value for both `a` and `b` is $-1 \in F$.

The GeneratorsOfAlgebra (62.9.1) and CanonicalBasis (61.5.3) value of an algebra constructed with QuaternionAlgebra is the list $[e,i,j,k]$.

Two quaternion algebras with the same parameters a, b lie in the same family, so it makes sense to consider their intersection or to ask whether they are contained in each other. (This is due to the fact that the results of QuaternionAlgebra are cached, in the global variable QuaternionAlgebraData.)

The embedding of the field GaussianRationals (60.1.3) into a quaternion algebra A over Rationals (17.1.1) is not uniquely determined. One can specify one embedding as a vector space homomorphism that maps 1 to the first algebra generator of A, and E(4) to one of the others.

Example

```
gap> QuaternionAlgebra( Rationals );
<algebra-with-one of dimension 4 over Rationals>
```

62.5.2 ComplexificationQuat (for a vector)

▷ ComplexificationQuat(*vector*) (function)
▷ ComplexificationQuat(*matrix*) (function)

Let $A = eF \oplus iF \oplus jF \oplus kF$ be a quaternion algebra over the field F of cyclotomics, with basis (e,i,j,k).

If $v = v_1 + v_2 j$ is a row vector over A with $v_1 = ew_1 + iw_2$ and $v_2 = ew_3 + iw_4$ then ComplexificationQuat called with argument v returns the concatenation of w_1+E(4)w_2 and w_3+E(4)w_4.

If $M = M_1 + M_2 j$ is a matrix over A with $M_1 = eN_1 + iN_2$ and $M_2 = eN_3 + iN_4$ then ComplexificationQuat called with argument M returns the block matrix A over $eF \oplus iF$ such that $A(1,1) = N_1$+E(4)N_2, $A(2,2) = N_1$−E(4)N_2, $A(1,2) = N_3$+E(4)N_4, and $A(2,1) = -N_3$+E(4)N_4.

Then ComplexificationQuat(v) * ComplexificationQuat(M)= ComplexificationQuat(v * M), since

$$vM = v_1M_1 + v_2jM_1 + v_1M_2j + v_2jM_2j = (v_1M_1 - v_2\overline{M_2}) + (v_1M_2 + v_2\overline{M_1})j.$$

62.5.3 OctaveAlgebra

▷ OctaveAlgebra(*F*) (function)

The algebra of octonions over F.

Example

```
gap> OctaveAlgebra( Rationals );
<algebra of dimension 8 over Rationals>
```

62.5.4 FullMatrixAlgebra

▷ FullMatrixAlgebra(*R, n*) (function)
▷ MatrixAlgebra(*R, n*) (function)
▷ MatAlgebra(*R, n*) (function)

is the full matrix algebra of $n \times n$ matrices over the ring R, for a nonnegative integer n.

Example

```
gap> A:=FullMatrixAlgebra( Rationals, 20 );
( Rationals^[ 20, 20 ] )
gap> Dimension( A );
400
```

62.5.5 NullAlgebra

▷ NullAlgebra(*R*) (attribute)

The zero-dimensional algebra over *R*.

Example

```
gap> A:= NullAlgebra( Rationals );
<algebra over Rationals>
gap> Dimension( A );
0
```

62.6 Subalgebras

62.6.1 Subalgebra

▷ Subalgebra(*A, gens[, "basis"]*) (function)

is the F-algebra generated by *gens*, with parent algebra *A*, where F is the left acting domain of *A*.

Note that being a subalgebra of *A* means to be an algebra, to be contained in *A*, *and* to have the same left acting domain as *A*.

An optional argument "basis" may be added if it is known that the generators already form a basis of the algebra. Then it is *not* checked whether *gens* really are linearly independent and whether all elements in *gens* lie in *A*.

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= Algebra( Rationals, [ m ] );
<algebra over Rationals, with 1 generators>
gap> B:= Subalgebra( A, [ m^2 ] );
<algebra over Rationals, with 1 generators>
```

62.6.2 SubalgebraNC

▷ SubalgebraNC(*A, gens[, "basis"]*) (function)

SubalgebraNC does the same as Subalgebra (62.6.1), except that it does not check whether all elements in *gens* lie in *A*.

Example

```
gap> m:= RandomMat( 3, 3 );;
gap> A:= Algebra( Rationals, [ m ] );
<algebra over Rationals, with 1 generators>
gap> SubalgebraNC( A, [ IdentityMat( 3, 3 ) ], "basis" );
<algebra of dimension 1 over Rationals>
```

62.6.3 SubalgebraWithOne

▷ SubalgebraWithOne(*A, gens[, "basis"]*) (function)

is the algebra-with-one generated by *gens*, with parent algebra *A*.

The optional third argument, the string "basis", may be added if it is known that the elements from *gens* are linearly independent. Then it is *not* checked whether *gens* really are linearly independent and whether all elements in *gens* lie in *A*.

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> B1:= SubalgebraWithOne( A, [ m ] );;
gap> B2:= Subalgebra( A, [ m ] );;
gap> Dimension( B1 );
3
gap> Dimension( B2 );
2
```

62.6.4 SubalgebraWithOneNC

▷ SubalgebraWithOneNC(*A, gens[, "basis"]*) (function)

SubalgebraWithOneNC does the same as SubalgebraWithOne (62.6.3), except that it does not check whether all elements in *gens* lie in *A*.

Example

```
gap> m:= RandomMat( 3, 3 );; A:= Algebra( Rationals, [ m ] );;
gap> SubalgebraWithOneNC( A, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
```

62.6.5 TrivialSubalgebra

▷ TrivialSubalgebra(*A*) (attribute)

The zero dimensional subalgebra of the algebra *A*.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= TrivialSubalgebra( A );
<algebra over Rationals>
gap> Dimension( B );
0
```

62.7 Ideals

For constructing and working with ideals in algebras the same functions are available as for ideals in rings. So for the precise description of these functions we refer to Chapter 56. Here we give examples demonstrating the use of ideals in algebras. For an introduction into the construction of quotient algebras we refer to Chapter (**Tutorial: Algebras**) of the user's tutorial.

Example

```
gap> m:= [ [ 0, 2, 3 ], [ 0, 0, 4 ], [ 0, 0, 0] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );;
gap> I:= Ideal( A, [ m ] );  # the two-sided ideal of ‘A’ generated by ‘m’
<two-sided ideal in <algebra-with-one of dimension 3 over Rationals>,
  (1 generators)>
gap> Dimension( I );
2
gap> GeneratorsOfIdeal( I );
[ [ [ 0, 2, 3 ], [ 0, 0, 4 ], [ 0, 0, 0 ] ] ]
gap> BasisVectors( Basis( I ) );
[ [ [ 0, 1, 3/2 ], [ 0, 0, 2 ], [ 0, 0, 0 ] ],
  [ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ] ]
gap> A:= FullMatrixAlgebra( Rationals, 4 );;
gap> m:= NullMat( 4, 4 );; m[1][4]:=1;;
gap> I:= LeftIdeal( A, [ m ] );
<left ideal in ( Rationals^[ 4, 4 ] ), (1 generators)>
gap> Dimension( I );
4
gap> GeneratorsOfLeftIdeal( I );
[ [ [ 0, 0, 0, 1 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ] ]
gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> A:= Algebra( Rationals, mats );;
gap> # Form the two-sided ideal for which ‘mats[2]’ is known to be
gap> # the unique basis element.
gap> I:= Ideal( A, [ mats[2] ], "basis" );
<two-sided ideal in <algebra of dimension 3 over Rationals>,
  (dimension 1)>
```

62.8 Categories and Properties of Algebras

62.8.1 IsFLMLOR

▷ IsFLMLOR(*obj*) (Category)

A FLMLOR (“free left module left operator ring”) in GAP is a ring that is also a free left module.

Note that this means that being a FLMLOR is not a property a ring can get, since a ring is usually not represented as an external left set.

Examples are magma rings (e.g. over the integers) or algebras.

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> IsFLMLOR ( A );
true
```

62.8.2 IsFLMLORWithOne

▷ IsFLMLORWithOne(*obj*) (Category)

A FLMLOR-with-one in GAP is a ring-with-one that is also a free left module.

Note that this means that being a FLMLOR-with-one is not a property a ring-with-one can get, since a ring-with-one is usually not represented as an external left set.

Examples are magma rings-with-one or algebras-with-one (but also over the integers).

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> IsFLMLORWithOne ( A );
true
```

62.8.3 IsAlgebra

▷ IsAlgebra(*obj*) (Category)

An algebra in GAP is a ring that is also a left vector space. Note that this means that being an algebra is not a property a ring can get, since a ring is usually not represented as an external left set.

Example

```
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsAlgebra( A );
true
```

62.8.4 IsAlgebraWithOne

▷ IsAlgebraWithOne(*obj*) (Category)

An algebra-with-one in GAP is a ring-with-one that is also a left vector space. Note that this means that being an algebra-with-one is not a property a ring-with-one can get, since a ring-with-one is usually not represented as an external left set.

Example

```
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsAlgebraWithOne( A );
true
```

62.8.5 IsLieAlgebra

▷ IsLieAlgebra(*A*) (property)

An algebra *A* is called Lie algebra if $a*a = 0$ for all a in *A* and $(a*(b*c))+(b*(c*a))+(c*(a*b)) = 0$ for all $a,b,c \in A$ (Jacobi identity).

Example

```
gap> A:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> IsLieAlgebra( A );
true
```

62.8.6 IsSimpleAlgebra

▷ IsSimpleAlgebra(*A*) (property)

is `true` if the algebra *A* is simple, and `false` otherwise. This function is only implemented for the cases where *A* is an associative or a Lie algebra. And for Lie algebras it is only implemented for the case where the ground field is of characteristic zero.

Example

```
gap> A:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> IsSimpleAlgebra( A );
false
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsSimpleAlgebra( A );
true
```

62.8.7 IsFiniteDimensional (for matrix algebras)

▷ IsFiniteDimensional(*matalg*) (method)

returns true (always) for a matrix algebra *matalg*, since matrix algebras are always finite dimensional.

Example

```
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsFiniteDimensional( A );
true
```

62.8.8 IsQuaternion

▷ IsQuaternion(*obj*) (Category)
▷ IsQuaternionCollection(*obj*) (Category)
▷ IsQuaternionCollColl(*obj*) (Category)

IsQuaternion is the category of elements in an algebra constructed by QuaternionAlgebra (62.5.1). A collection of quaternions lies in the category IsQuaternionCollection. Finally, a collection of quaternion collections (e.g., a matrix of quaternions) lies in the category IsQuaternionCollColl.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> b:= BasisVectors( Basis( A ) );
[ e, i, j, k ]
gap> IsQuaternion( b[1] );
true
gap> IsQuaternionCollColl( [ [ b[1], b[2] ], [ b[3], b[4] ] ] );
true
```

62.9 Attributes and Operations for Algebras

62.9.1 GeneratorsOfAlgebra

▷ GeneratorsOfAlgebra(*A*) (attribute)

returns a list of elements that generate *A* as an algebra.

For a free algebra, each generator can also be accessed using the . operator (see GeneratorsOfDomain (31.9.2)).

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> GeneratorsOfAlgebra( A );
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ] ]
```

62.9.2 GeneratorsOfAlgebraWithOne

▷ `GeneratorsOfAlgebraWithOne(A)` (attribute)

returns a list of elements of A that generate A as an algebra with one.

For a free algebra with one, each generator can also be accessed using the . operator (see `GeneratorsOfDomain` (31.9.2)).

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> GeneratorsOfAlgebraWithOne( A );
[ [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ] ]
```

62.9.3 ProductSpace

▷ `ProductSpace(U, V)` (operation)

is the vector space $\langle u * v; u \in U, v \in V \rangle$, where U and V are subspaces of the same algebra.

If $U = V$ is known to be an algebra then the product space is also an algebra, moreover it is an ideal in U. If U and V are known to be ideals in an algebra A then the product space is known to be an algebra and an ideal in A.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> b:= BasisVectors( Basis( A ) );;
gap> B:= Subalgebra( A, [ b[4] ] );
<algebra over Rationals, with 1 generators>
gap> ProductSpace( A, B );
<vector space of dimension 4 over Rationals>
```

62.9.4 PowerSubalgebraSeries

▷ `PowerSubalgebraSeries(A)` (attribute)

returns a list of subalgebras of A, the first term of which is A; and every next term is the product space of the previous term with itself.

Example

```
gap> A:= QuaternionAlgebra( Rationals );
<algebra-with-one of dimension 4 over Rationals>
gap> PowerSubalgebraSeries( A );
[ <algebra-with-one of dimension 4 over Rationals> ]
```

62.9.5 AdjointBasis

▷ `AdjointBasis(B)` (attribute)

The *adjoint map* $ad(x)$ of an element x in an F-algebra A, say, is the left multiplication by x. This map is F-linear and thus, w.r.t. the given basis $B = (x_1, x_2, \ldots, x_n)$ of A, $ad(x)$ can be represented by a matrix over F. Let V denote the F-vector space of the matrices corresponding to $ad(x)$, for $x \in A$. Then `AdjointBasis` returns the basis of V that consists of the matrices for $ad(x_1), \ldots, ad(x_n)$.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> AdjointBasis( Basis( A ) );
Basis( <vector space over Rationals, with 4 generators>,
[ [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ],
  [ [ 0, -1, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, -1 ], [ 0, 0, 1, 0 ] ]
    ,
  [ [ 0, 0, -1, 0 ], [ 0, 0, 0, 1 ], [ 1, 0, 0, 0 ], [ 0, -1, 0, 0 ] ]
    ,
  [ [ 0, 0, 0, -1 ], [ 0, 0, -1, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ]
    ] ] )
```

62.9.6 IndicesOfAdjointBasis

▷ `IndicesOfAdjointBasis(B)` (attribute)

Let `A` be an algebra and let `B` be the basis that is output by `AdjointBasis( Basis( A ) )`. This function returns a list of indices. If i is an index belonging to this list, then adx_i is a basis vector of the matrix space spanned by adA, where x_i is the i-th basis vector of the basis `B`.

Example

```
gap> L:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> B:= AdjointBasis( Basis( L ) );;
gap> IndicesOfAdjointBasis( B );
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

62.9.7 AsAlgebra

▷ `AsAlgebra(F, A)` (operation)

Returns the algebra over `F` generated by `A`.

Example

```
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> AsAlgebra( Rationals, V );
<algebra of dimension 1 over Rationals>
```

62.9.8 AsAlgebraWithOne

▷ `AsAlgebraWithOne(F, A)` (operation)

If the algebra `A` has an identity, then it can be viewed as an algebra with one over `F`. This function returns this algebra with one.

Example
```
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> A:= AsAlgebra( Rationals, V );;
gap> AsAlgebraWithOne( Rationals, A );
<algebra-with-one over Rationals, with 1 generators>
```

62.9.9 AsSubalgebra

▷ AsSubalgebra(*A*, *B*) (operation)

If all elements of the algebra *B* happen to be contained in the algebra *A*, then *B* can be viewed as a subalgebra of *A*. This function returns this subalgebra.

Example
```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> B:= AsAlgebra( Rationals, V );;
gap> BA:= AsSubalgebra( A, B );
<algebra of dimension 1 over Rationals>
```

62.9.10 AsSubalgebraWithOne

▷ AsSubalgebraWithOne(*A*, *B*) (operation)

If *B* is an algebra with one, all elements of which happen to be contained in the algebra with one *A*, then *B* can be viewed as a subalgebra with one of *A*. This function returns this subalgebra with one.

Example
```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> B:= AsAlgebra( Rationals, V );;
gap> C:= AsAlgebraWithOne( Rationals, B );;
gap> AC:= AsSubalgebraWithOne( A, C );
<algebra-with-one over Rationals, with 1 generators>
```

62.9.11 MutableBasisOfClosureUnderAction

▷ MutableBasisOfClosureUnderAction(*F*, *Agens*, *from*, *init*, *opr*, *zero*, *maxdim*) (function)

Let *F* be a ring, *Agens* a list of generators for an *F*-algebra *A*, and *from* one of "left", "right", "both"; this means that elements of *A* act via multiplication from the respective side(s). *init* must be a list of initial generating vectors, and *opr* the operation (a function of two arguments).

MutableBasisOfClosureUnderAction returns a mutable basis of the *F*-free left module generated by the vectors in *init* and their images under the action of *Agens* from the respective side(s).

zero is the zero element of the desired module. *maxdim* is an upper bound for the dimension of the closure; if no such upper bound is known then the value of *maxdim* must be infinity (18.2.1).

MutableBasisOfClosureUnderAction can be used to compute a basis of an *associative* algebra generated by the elements in *Agens*. In this case *from* may be "left" or "right", *opr* is the multiplication *, and *init* is a list containing either the identity of the algebra or a list of algebra

generators. (Note that if the algebra has an identity then it is in general not sufficient to take algebra-with-one generators as *init*, whereas of course *Agens* need not contain the identity.)

(Note that bases of *not* necessarily associative algebras can be computed using MutableBasisOfNonassociativeAlgebra (62.9.12).)

Other applications of MutableBasisOfClosureUnderAction are the computations of bases for (left/ right/ two-sided) ideals I in an *associative* algebra A from ideal generators of I; in these cases *Agens* is a list of algebra generators of A, *from* denotes the appropriate side(s), *init* is a list of ideal generators of I, and *opr* is again $*$.

(Note that bases of ideals in *not* necessarily associative algebras can be computed using MutableBasisOfIdealInNonassociativeAlgebra (62.9.13).)

Finally, bases of right A-modules also can be computed using MutableBasisOfClosureUnderAction. The only difference to the ideal case is that *init* is now a list of right module generators, and *opr* is the operation of the module.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> g:= GeneratorsOfAlgebra( A );;
gap> B:= MutableBasisOfClosureUnderAction( Rationals,
>                                g, "left", [ g[1] ], \*, Zero(A), 4 );
<mutable basis over Rationals, 4 vectors>
gap> BasisVectors( B );
[ e, i, j, k ]
```

62.9.12 MutableBasisOfNonassociativeAlgebra

▷ MutableBasisOfNonassociativeAlgebra(*F, Agens, zero, maxdim*) (function)

is a mutable basis of the (not necessarily associative) F-algebra that is generated by *Agens*, has zero element *zero*, and has dimension at most *maxdim*. If no finite bound for the dimension is known then infinity (18.2.1) must be the value of *maxdim*.

The difference to MutableBasisOfClosureUnderAction (62.9.11) is that in general it is not sufficient to multiply just with algebra generators. (For special cases of nonassociative algebras, especially for Lie algebras, multiplying with algebra generators suffices.)

Example

```
gap> L:= FullMatrixLieAlgebra( Rationals, 4 );;
gap> m1:= Random( L );;
gap> m2:= Random( L );;
gap> MutableBasisOfNonassociativeAlgebra( Rationals, [ m1, m2 ],
> Zero( L ), 16 );
<mutable basis over Rationals, 16 vectors>
```

62.9.13 MutableBasisOfIdealInNonassociativeAlgebra

▷ MutableBasisOfIdealInNonassociativeAlgebra(*F, Vgens, Igens, zero, from, maxdim*) (function)

is a mutable basis of the ideal generated by *Igens* under the action of the (not necessarily associative) F-algebra with vector space generators *Vgens*. The zero element of the ideal is *zero*, *from* is one of "left", "right", "both" (with the same meaning as in

MutableBasisOfClosureUnderAction (62.9.11)), and *maxdim* is a known upper bound on the dimension of the ideal; if no finite bound for the dimension is known then infinity (18.2.1) must be the value of *maxdim*.

The difference to MutableBasisOfClosureUnderAction (62.9.11) is that in general it is not sufficient to multiply just with algebra generators. (For special cases of nonassociative algebras, especially for Lie algebras, multiplying with algebra generators suffices.)

Example

```
gap> mats:= [  [[ 1, 0 ], [ 0, -1 ]], [[0,1],[0,0]] ];;
gap> A:= Algebra( Rationals, mats );;
gap> basA:= BasisVectors( Basis( A ) );;
gap> B:= MutableBasisOfIdealInNonassociativeAlgebra( Rationals, basA,
> [ mats[2] ], 0*mats[1], "both", infinity );
<mutable basis over Rationals, 1 vectors>
gap> BasisVectors( B );
[ [ [ 0, 1 ], [ 0, 0 ] ] ]
```

62.9.14 DirectSumOfAlgebras (for two algebras)

▷ DirectSumOfAlgebras(*A1, A2*) (operation)

▷ DirectSumOfAlgebras(*list*) (operation)

is the direct sum of the two algebras *A1* and *A2* respectively of the algebras in the list *list*.

If all involved algebras are associative algebras then the result is also known to be associative. If all involved algebras are Lie algebras then the result is also known to be a Lie algebra.

All involved algebras must have the same left acting domain.

The default case is that the result is a structure constants algebra. If all involved algebras are matrix algebras, and either both are Lie algebras or both are associative then the result is again a matrix algebra of the appropriate type.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> DirectSumOfAlgebras( [A, A, A] );
<algebra of dimension 12 over Rationals>
```

62.9.15 FullMatrixAlgebraCentralizer

▷ FullMatrixAlgebraCentralizer(*F, lst*) (function)

Let *lst* be a nonempty list of square matrices of the same dimension n, say, with entries in the field *F*. FullMatrixAlgebraCentralizer returns the (pointwise) centralizer of all matrices in *lst*, inside the full matrix algebra of $n \times n$ matrices over *F*.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> b:= Basis( A );;
gap> mats:= List( BasisVectors( b ), x -> AdjointMatrix( b, x ) );;
gap> FullMatrixAlgebraCentralizer( Rationals, mats );
<algebra-with-one of dimension 4 over Rationals>
```

62.9.16 RadicalOfAlgebra

▷ RadicalOfAlgebra(*A*) (attribute)

is the maximal nilpotent ideal of *A*, where *A* is an associative algebra.

Example

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOneByGenerators( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> RadicalOfAlgebra( A );
<algebra of dimension 2 over Rationals>
```

62.9.17 CentralIdempotentsOfAlgebra

▷ CentralIdempotentsOfAlgebra(*A*) (attribute)

For an associative algebra *A*, this function returns a list of central primitive idempotents such that their sum is the identity element of *A*. Therefore *A* is required to have an identity.

(This is a synonym of CentralIdempotentsOfSemiring.)

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= DirectSumOfAlgebras( [A, A, A] );
<algebra of dimension 12 over Rationals>
gap> CentralIdempotentsOfAlgebra( B );
[ v.9, v.5, v.1 ]
```

62.9.18 DirectSumDecomposition (for Lie algebras)

▷ DirectSumDecomposition(*L*) (attribute)

This function calculates a list of ideals of the algebra *L* such that *L* is equal to their direct sum. Currently this is only implemented for semisimple associative algebras, and for Lie algebras (semisimple or not).

Example

```
gap> G:= SymmetricGroup( 4 );;
gap> A:= GroupRing( Rationals, G );
<algebra-with-one over Rationals, with 2 generators>
gap> dd:= DirectSumDecomposition( A );
[ <two-sided ideal in
      <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
  <two-sided ideal in
      <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
  <two-sided ideal in
      <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
  <two-sided ideal in
      <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
```

```
    <two-sided ideal in
        <algebra-with-one of dimension 24 over Rationals>,
        (1 generators)> ]
  gap> List( dd, Dimension );
  [ 1, 1, 4, 9, 9 ]
```

Example

```
gap> L:= FullMatrixLieAlgebra( Rationals, 5 );;
gap> DirectSumDecomposition( L );
[ <two-sided ideal in
      <two-sided ideal in <Lie algebra of dimension 25 over Rationals>
            , (dimension 1)>, (dimension 1)>,
  <two-sided ideal in
      <two-sided ideal in <Lie algebra of dimension 25 over Rationals>
            , (dimension 24)>, (dimension 24)> ]
```

62.9.19 LeviMalcevDecomposition (for Lie algebras)

▷ `LeviMalcevDecomposition(`*L*`)` (attribute)

A Levi-Malcev subalgebra of the algebra *L* is a semisimple subalgebra complementary to the radical of *L*. This function returns a list with two components. The first component is a Levi-Malcev subalgebra, the second the radical. This function is implemented for associative and Lie algebras.

Example

```
gap> m:= [ [ 1, 2, 0 ], [ 0, 1, 3 ], [ 0, 0, 1] ];;
gap> A:= Algebra( Rationals, [ m ] );;
gap> LeviMalcevDecomposition( A );
[ <algebra of dimension 1 over Rationals>,
  <algebra of dimension 2 over Rationals> ]
```

Example

```
gap> L:= FullMatrixLieAlgebra( Rationals, 5 );;
gap> LeviMalcevDecomposition( L );
[ <Lie algebra of dimension 24 over Rationals>,
  <two-sided ideal in <Lie algebra of dimension 25 over Rationals>,
      (dimension 1)> ]
```

62.9.20 Grading

▷ `Grading(`*A*`)` (attribute)

Let G be an Abelian group and A an algebra. Then A is said to be graded over G if for every $g \in G$ there is a subspace A_g of A such that $A_g \cdot A_h \subset A_{g+h}$ for $g,h \in G$. In GAP 4 a *grading* of an algebra is a record containing the following components.

`source`
: the Abelian group over which the algebra is graded.

`hom_components`
: a function assigning to each element from the source a subspace of the algebra.

`min_degree`
: in the case where the algebra is graded over the integers this is the minimum number for which `hom_components` returns a nonzero subspace.

`max_degree`
: is analogous to `min_degree`.

We note that there are no methods to compute a grading of an arbitrary algebra; however some algebras get a natural grading when they are constructed (see `JenningsLieAlgebra` (64.8.4), `NilpotentQuotientOfFpLieAlgebra` (64.11.2)).

We note also that these components may be not enough to handle the grading efficiently, and another record component may be needed. For instance in a Lie algebra L constructed by `JenningsLieAlgebra` (64.8.4), the length of the of the range `[ Grading(L)!.min_degree .. Grading(L)!.max_degree ]` may be non-polynomial in the dimension of L. To handle efficiently this situation, an optional component can be used:

`non_zero_hom_components`
: the subset of `source` for which `hom_components` returns a nonzero subspace.

Example

```
gap> G:= SmallGroup(3^6, 100 );
<pc group of size 729 with 6 generators>
gap> L:= JenningsLieAlgebra( G );
<Lie algebra of dimension 6 over GF(3)>
gap> g:= Grading( L );
rec( hom_components := function( d ) ... end, max_degree := 9,
  min_degree := 1, source := Integers )
gap> g.hom_components( 3 );
<vector space over GF(3), with 1 generators>
gap> g.hom_components( 14 );
<vector space over GF(3), with 0 generators>
```

62.10 Homomorphisms of Algebras

Algebra homomorphisms are vector space homomorphisms that preserve the multiplication. So the default methods for vector space homomorphisms work, and in fact there is not much use of the fact that source and range are algebras, except that preimages and images are algebras (or even ideals) in certain cases.

62.10.1 AlgebraGeneralMappingByImages

▷ `AlgebraGeneralMappingByImages(`*A, B, gens, imgs*`)` (operation)

is a general mapping from the F-algebra *A* to the F-algebra *B*. This general mapping is defined by mapping the entries in the list *gens* (elements of *A*) to the entries in the list *imgs* (elements of *B*), and taking the F-linear and multiplicative closure.

gens need not generate *A* as an F-algebra, and if the specification does not define a linear and multiplicative mapping then the result will be multivalued. Hence, in general it is not a mapping. For constructing a linear map that is not necessarily multiplicative, we refer to `LeftModuleHomomorphismByImages` (61.10.2).

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= FullMatrixAlgebra( Rationals, 2 );;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraGeneralMappingByImages( A, B, bA, bB );
[ e, i, j, k ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0 ], [ 1, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
gap> Images( f, bA[1] );
<add. coset of <algebra over Rationals, with 16 generators>>
```

62.10.2 AlgebraHomomorphismByImages

▷ AlgebraHomomorphismByImages(*A, B, gens, imgs*) (function)

AlgebraHomomorphismByImages returns the algebra homomorphism with source *A* and range *B* that is defined by mapping the list *gens* of generators of *A* to the list *imgs* of images in *B*.

If *gens* does not generate *A* or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then fail is returned.

One can avoid the checks by calling AlgebraHomomorphismByImagesNC (62.10.3), and one can construct multi-valued mappings with AlgebraGeneralMappingByImages (62.10.1).

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] ); SetEntrySCTable( T, 2, 2, [1,2] );
gap> A:= AlgebraByStructureConstants( Rationals, T );;
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> B:= AlgebraByGenerators( Rationals, [ m1, m2 ] );;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraHomomorphismByImages( A, B, bA, bB );
[ v.1, v.2 ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
gap> Image( f, bA[1]+bA[2] );
[ [ 1, 0 ], [ 0, 1 ] ]
```

62.10.3 AlgebraHomomorphismByImagesNC

▷ AlgebraHomomorphismByImagesNC(*A, B, gens, imgs*) (operation)

AlgebraHomomorphismByImagesNC is the operation that is called by the function AlgebraHomomorphismByImages (62.10.2). Its methods may assume that *gens* generates *A* and that the mapping of *gens* to *imgs* defines an algebra homomorphism. Results are unpredictable if these conditions do not hold.

For creating a possibly multi-valued mapping from *A* to *B* that respects addition, multiplication, and scalar multiplication, AlgebraGeneralMappingByImages (62.10.1) can be used.

For the definitions of the algebras A and B in the next example we refer to the previous example.

Example

```
gap> f:= AlgebraHomomorphismByImagesNC( A, B, bA, bB );
[ v.1, v.2 ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
```

62.10.4 AlgebraWithOneGeneralMappingByImages

▷ AlgebraWithOneGeneralMappingByImages(*A, B, gens, imgs*) (operation)

This function is analogous to AlgebraGeneralMappingByImages (62.10.1); the only difference being that the identity of *A* is automatically mapped to the identity of *B*.

Example

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= FullMatrixAlgebra( Rationals, 2 );;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:=AlgebraWithOneGeneralMappingByImages(A,B,bA{[2,3,4]},bB{[1,2,3]});
[ i, j, k, e ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0 ], [ 1, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ]
```

62.10.5 AlgebraWithOneHomomorphismByImages

▷ AlgebraWithOneHomomorphismByImages(*A, B, gens, imgs*) (function)

AlgebraWithOneHomomorphismByImages returns the algebra-with-one homomorphism with source *A* and range *B* that is defined by mapping the list *gens* of generators of *A* to the list *imgs* of images in *B*.

The difference between an algebra homomorphism and an algebra-with-one homomorphism is that in the latter case, it is assumed that the identity of *A* is mapped to the identity of *B*, and therefore *gens* needs to generate *A* only as an algebra-with-one.

If *gens* does not generate *A* or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then fail is returned.

One can avoid the checks by calling AlgebraWithOneHomomorphismByImagesNC (62.10.6), and one can construct multi-valued mappings with AlgebraWithOneGeneralMappingByImages (62.10.4).

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:=1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:=1;;
gap> A:= AlgebraByGenerators( Rationals, [m1,m2] );;
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] );
gap> SetEntrySCTable( T, 2, 2, [1,2] );
gap> B:= AlgebraByStructureConstants(Rationals, T);;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraWithOneHomomorphismByImages( A, B, bA{[1]}, bB{[1]} );
[ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] -> [ v.1, v.1+v.2 ]
```

62.10.6 AlgebraWithOneHomomorphismByImagesNC

▷ AlgebraWithOneHomomorphismByImagesNC(*A, B, gens, imgs*) (operation)

AlgebraWithOneHomomorphismByImagesNC is the operation that is called by the function AlgebraWithOneHomomorphismByImages (62.10.5). Its methods may assume that *gens* generates *A* and that the mapping of *gens* to *imgs* defines an algebra-with-one homomorphism. Results are unpredictable if these conditions do not hold.

For creating a possibly multi-valued mapping from *A* to *B* that respects addition, multiplication, identity, and scalar multiplication, AlgebraWithOneGeneralMappingByImages (62.10.4) can be used.

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:=1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:=1;;
gap> A:= AlgebraByGenerators( Rationals, [m1,m2] );;
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] );
gap> SetEntrySCTable( T, 2, 2, [1,2] );
gap> B:= AlgebraByStructureConstants( Rationals, T);;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraWithOneHomomorphismByImagesNC( A, B, bA{[1]}, bB{[1]} );
[ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] -> [ v.1, v.1+v.2 ]
```

62.10.7 NaturalHomomorphismByIdeal (for an algebra and an ideal)

▷ NaturalHomomorphismByIdeal(*A, I*) (method)

For an algebra *A* and an ideal *I* in *A*, the return value of NaturalHomomorphismByIdeal (56.8.4) is a homomorphism of algebras, in particular the range of this mapping is also an algebra.

Example

```
gap> L:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> C:= LieCentre( L );
<two-sided ideal in <Lie algebra of dimension 9 over Rationals>,
  (dimension 1)>
gap> hom:= NaturalHomomorphismByIdeal( L, C );
<linear mapping by matrix, <Lie algebra of dimension
9 over Rationals> -> <Lie algebra of dimension 8 over Rationals>>
gap> ImagesSource( hom );
<Lie algebra of dimension 8 over Rationals>
```

62.10.8 OperationAlgebraHomomorphism (action w.r.t. a basis of the module)

▷ OperationAlgebraHomomorphism(*A, B[, opr]*) (operation)
▷ OperationAlgebraHomomorphism(*A, V[, opr]*) (operation)

OperationAlgebraHomomorphism returns an algebra homomorphism from the F-algebra *A* into a matrix algebra over F that describes the F-linear action of *A* on the basis *B* of a free left module respectively on the free left module *V* (in which case some basis of *V* is chosen), via the operation *opr*.

The homomorphism need not be surjective. The default value for *opr* is OnRight (41.2.2).

If *A* is an algebra-with-one then the operation homomorphism is an algebra-with-one homomorphism because the identity of *A* must act as the identity.

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> B:= AlgebraByGenerators( Rationals, [ m1, m2 ] );;
gap> V:= FullRowSpace( Rationals, 2 );
```

```
( Rationals^2 )
gap> f:=OperationAlgebraHomomorphism( B, Basis( V ), OnRight );
<op. hom. Algebra( Rationals,
[ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ]
 ] ) -> matrices of dim. 2>
gap> Image( f, m1 );
[ [ 1, 0 ], [ 0, 0 ] ]
```

62.10.9 NiceAlgebraMonomorphism

▷ NiceAlgebraMonomorphism(*A*) (attribute)

If *A* is an associative algebra with one, returns an isomorphism from *A* onto a matrix algebra (see IsomorphismMatrixAlgebra (62.10.11) for an example). If *A* is a finitely presented Lie algebra, returns an isomorphism from *A* onto a Lie algebra defined by a structure constants table (see 64.11 for an example).

62.10.10 IsomorphismFpAlgebra

▷ IsomorphismFpAlgebra(*A*) (attribute)

isomorphism from the algebra *A* onto a finitely presented algebra. Currently this is only implemented for associative algebras with one.

Example

```
gap> A:= QuaternionAlgebra( Rationals );
<algebra-with-one of dimension 4 over Rationals>
gap> f:= IsomorphismFpAlgebra( A );
[ e, i, j, k, e ] -> [ [(1)*x.1], [(1)*x.2], [(1)*x.3], [(1)*x.4],
  [(1)*<identity ...>] ]
```

62.10.11 IsomorphismMatrixAlgebra

▷ IsomorphismMatrixAlgebra(*A*) (attribute)

isomorphism from the algebra *A* onto a matrix algebra. Currently this is only implemented for associative algebras with one.

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] ); SetEntrySCTable( T, 2, 2, [1,2] );
gap> A:= AlgebraByStructureConstants( Rationals, T );;
gap> A:= AsAlgebraWithOne( Rationals, A );;
gap> f:=IsomorphismMatrixAlgebra( A );
<op. hom. AlgebraWithOne( Rationals, ... ) -> matrices of dim. 2>
gap> Image( f, BasisVectors( Basis( A ) )[1] );
[ [ 1, 0 ], [ 0, 0 ] ]
```

62.10.12 IsomorphismSCAlgebra (w.r.t. a given basis)

▷ IsomorphismSCAlgebra(*B*) (attribute)
▷ IsomorphismSCAlgebra(*A*) (attribute)

For a basis *B* of an algebra *A*, say, IsomorphismSCAlgebra returns an algebra isomorphism from *A* to an algebra *S* given by structure constants (see 62.4), such that the canonical basis of *S* is the image of *B*.

For an algebra *A*, IsomorphismSCAlgebra chooses a basis of *A* and returns the IsomorphismSCAlgebra value for that basis.

Example

```
gap> IsomorphismSCAlgebra( GF(8) );
CanonicalBasis( GF(2^3) ) -> CanonicalBasis( <algebra of dimension
3 over GF(2)> )
gap> IsomorphismSCAlgebra( GF(2)^[2,2] );
CanonicalBasis( ( GF(2)^
[ 2, 2 ] ) ) -> CanonicalBasis( <algebra of dimension 4 over GF(2)> )
```

62.10.13 RepresentativeLinearOperation

▷ RepresentativeLinearOperation(*A, v, w, opr*) (operation)

is an element of the algebra *A* that maps the vector *v* to the vector *w* under the linear operation described by the function *opr*. If no such element exists then fail is returned.

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> B:= AlgebraByGenerators( Rationals, [ m1, m2 ] );;
gap> RepresentativeLinearOperation( B, [1,0], [1,0], OnRight );
[ [ 1, 0 ], [ 0, 0 ] ]
gap> RepresentativeLinearOperation( B, [1,0], [0,1], OnRight );
fail
```

62.11 Representations of Algebras

An algebra module is a vector space together with an action of an algebra. So a module over an algebra is constructed by giving generators of a vector space, and a function for calculating the action of algebra elements on elements of the vector space. When creating an algebra module, the generators of the vector space are wrapped up and given the category IsLeftAlgebraModuleElement or IsRightModuleElement if the algebra acts from the left, or right respectively. (So in the case of a bi-module the elements get both categories.) Most linear algebra computations are delegated to the original vector space.

The transition between the original vector space and the corresponding algebra module is handled by ExtRepOfObj and ObjByExtRep. For an element v of the algebra module, ExtRepOfObj(v) returns the underlying element of the original vector space. Furthermore, if vec is an element of the original vector space, and fam the elements family of the corresponding algebra module, then ObjByExtRep(fam, vec) returns the corresponding element of the algebra module. Below is an example of this.

The action of the algebra on elements of the algebra module is constructed by using the operator `^`. If `x` is an element of an algebra `A`, and `v` an element of a left `A`-module, then `x^v` calculates the result of the action of `x` on `v`. Similarly, if `v` is an element of a right `A`-module, then `v^x` calculates the action of `x` on `v`.

62.11.1 LeftAlgebraModuleByGenerators

▷ `LeftAlgebraModuleByGenerators(`*A, op, gens*`)` (operation)

Constructs the left algebra module over *A* generated by the list of vectors *gens*. The action of *A* is described by the function *op*. This must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector; it outputs the result of applying the algebra element to the vector.

62.11.2 RightAlgebraModuleByGenerators

▷ `RightAlgebraModuleByGenerators(`*A, op, gens*`)` (operation)

Constructs the right algebra module over *A* generated by the list of vectors *gens*. The action of *A* is described by the function *op*. This must be a function of two arguments; the first argument is a vector, and the second argument is the algebra element; it outputs the result of applying the algebra element to the vector.

62.11.3 BiAlgebraModuleByGenerators

▷ `BiAlgebraModuleByGenerators(`*A, B, opl, opr, gens*`)` (operation)

Constructs the algebra bi-module over *A* and *B* generated by the list of vectors *gens*. The left action of *A* is described by the function *opl*, and the right action of *B* by the function *opr*. *opl* must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector; it outputs the result of applying the algebra element on the left to the vector. *opr* must be a function of two arguments; the first argument is a vector, and the second argument is the algebra element; it outputs the result of applying the algebra element on the right to the vector.

Example

```
gap> A:= Rationals^[3,3];
( Rationals^[ 3, 3 ] )
gap> V:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> W:= RightAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );
<right-module over ( Rationals^[ 3, 3 ] )>
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );
<bi-module over ( Rationals^[ 3, 3 ] ) (left) and ( Rationals^
[ 3, 3 ] ) (right)>
```

In the above examples, the modules `V`, `W`, and `M` are 3-dimensional vector spaces over the rationals. The algebra `A` acts from the left on `V`, from the right on `W`, and from the left and from the right on `M`.

62.11.4 LeftAlgebraModule

▷ LeftAlgebraModule(*A, op, V*) (operation)

Constructs the left algebra module over *A* with underlying space *V*. The action of *A* is described by the function *op*. This must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector from *V*; it outputs the result of applying the algebra element to the vector.

62.11.5 RightAlgebraModule

▷ RightAlgebraModule(*A, op, V*) (operation)

Constructs the right algebra module over *A* with underlying space *V*. The action of *A* is described by the function *op*. This must be a function of two arguments; the first argument is a vector, from *V* and the second argument is the algebra element; it outputs the result of applying the algebra element to the vector.

62.11.6 BiAlgebraModule

▷ BiAlgebraModule(*A, B, opl, opr, V*) (operation)

Constructs the algebra bi-module over *A* and *B* with underlying space *V*. The left action of *A* is described by the function *opl*, and the right action of *B* by the function *opr*. *opl* must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector from *V*; it outputs the result of applying the algebra element on the left to the vector. *opr* must be a function of two arguments; the first argument is a vector from *V*, and the second argument is the algebra element; it outputs the result of applying the algebra element on the right to the vector.

Example

```
gap> A:= Rationals^[3,3];;
gap> V:= Rationals^3;
( Rationals^3 )
gap> V:= Rationals^3;;
gap> M:= BiAlgebraModule( A, A, \*, \*, V );
<bi-module over ( Rationals^[ 3, 3 ] ) (left) and ( Rationals^
[ 3, 3 ] ) (right)>
gap> Dimension( M );
3
```

62.11.7 GeneratorsOfAlgebraModule

▷ GeneratorsOfAlgebraModule(*M*) (attribute)

A list of elements of *M* that generate *M* as an algebra module.

Example

```
gap> A:= Rationals^[3,3];;
gap> V:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> GeneratorsOfAlgebraModule( V );
[ [ 1, 0, 0 ] ]
```

62.11.8 IsAlgebraModuleElement

▷ IsAlgebraModuleElement(*obj*) (Category)
▷ IsAlgebraModuleElementCollection(*obj*) (Category)
▷ IsAlgebraModuleElementFamily(*fam*) (Category)

Category of algebra module elements. If an object has IsAlgebraModuleElementCollection, then it is an algebra module. If a family has IsAlgebraModuleElementFamily, then it is a family of algebra module elements (every algebra module has its own elements family).

62.11.9 IsLeftAlgebraModuleElement

▷ IsLeftAlgebraModuleElement(*obj*) (Category)
▷ IsLeftAlgebraModuleElementCollection(*obj*) (Category)

Category of left algebra module elements. If an object has IsLeftAlgebraModuleElementCollection, then it is a left-algebra module.

62.11.10 IsRightAlgebraModuleElement

▷ IsRightAlgebraModuleElement(*obj*) (Category)
▷ IsRightAlgebraModuleElementCollection(*obj*) (Category)

Category of right algebra module elements. If an object has IsRightAlgebraModuleElementCollection, then it is a right-algebra module.

Example

```
gap> A:= Rationals^[3,3];
( Rationals^[ 3, 3 ] )
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );
<bi-module over ( Rationals^[ 3, 3 ] ) (left) and ( Rationals^
[ 3, 3 ] ) (right)>
gap> vv:= BasisVectors( Basis( M ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> IsLeftAlgebraModuleElement( vv[1] );
true
gap> IsRightAlgebraModuleElement( vv[1] );
true
gap> vv[1] = [ 1, 0, 0 ];
false
gap> ExtRepOfObj( vv[1] ) = [ 1, 0, 0 ];
true
gap> ObjByExtRep( ElementsFamily( FamilyObj( M ) ), [ 1, 0, 0 ] ) in M;
true
gap> xx:= BasisVectors( Basis( A ) );;
gap> xx[4]^vv[1];  # left action
[ 0, 1, 0 ]
gap> vv[1]^xx[2];  # right action
[ 0, 1, 0 ]
```

62.11.11 LeftActingAlgebra

▷ LeftActingAlgebra(*V*) (attribute)

Here *V* is a left-algebra module; this function returns the algebra that acts from the left on *V*.

62.11.12 RightActingAlgebra

▷ RightActingAlgebra(*V*) (attribute)

Here *V* is a right-algebra module; this function returns the algebra that acts from the right on *V*.

62.11.13 ActingAlgebra

▷ ActingAlgebra(*V*) (operation)

Here *V* is an algebra module; this function returns the algebra that acts on *V* (this is the same as LeftActingAlgebra(*V*) if *V* is a left module, and RightActingAlgebra(*V*) if *V* is a right module; it will signal an error if *V* is a bi-module).

Example

```
gap> A:= Rationals^[3,3];;
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );;
gap> LeftActingAlgebra( M );
( Rationals^[ 3, 3 ] )
gap> RightActingAlgebra( M );
( Rationals^[ 3, 3 ] )
gap> V:= RightAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> ActingAlgebra( V );
( Rationals^[ 3, 3 ] )
```

62.11.14 IsBasisOfAlgebraModuleElementSpace

▷ IsBasisOfAlgebraModuleElementSpace(*B*) (Category)

If a basis *B* lies in the category IsBasisOfAlgebraModuleElementSpace, then *B* is a basis of a subspace of an algebra module. This means that *B* has the record field *B*!.delegateBasis set. This last object is a basis of the corresponding subspace of the vector space underlying the algebra module (i.e., the vector space spanned by all ExtRepOfObj(v) for v in the algebra module).

Example

```
gap> A:= Rationals^[3,3];;
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );;
gap> B:= Basis( M );
Basis( <3-dimensional bi-module over ( Rationals^
[ 3, 3 ] ) (left) and ( Rationals^[ 3, 3 ] ) (right)>,
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] )
gap> IsBasisOfAlgebraModuleElementSpace( B );
true
gap> B!.delegateBasis;
SemiEchelonBasis( <vector space of dimension 3 over Rationals>,
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] )
```

62.11.15 MatrixOfAction

▷ MatrixOfAction(*B, x[, side]*) (operation)

Here *B* is a basis of an algebra module and *x* is an element of the algebra that acts on this module. This function returns the matrix of the action of *x* with respect to *B*. If *x* acts from the left, then the coefficients of the images of the basis elements of *B* (under the action of *x*) are the columns of the output. If *x* acts from the right, then they are the rows of the output.

If the module is a bi-module, then the third parameter *side* must be specified. This is the string "left", or "right" depending whether *x* acts from the left or the right.

Example

```
gap> M:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> x:= Basis(A)[3];
[ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ]
gap> MatrixOfAction( Basis( M ), x );
[ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ]
```

62.11.16 SubAlgebraModule

▷ SubAlgebraModule(*M, gens[, "basis"]*) (operation)

is the sub-module of the algebra module *M*, generated by the vectors in *gens*. If as an optional argument the string basis is added, then it is assumed that the vectors in *gens* form a basis of the submodule.

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> A:= Algebra( Rationals, [ m1, m2 ] );;
gap> M:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0 ], [ 0, 1 ] ] );
<left-module over <algebra over Rationals, with 2 generators>>
gap> bb:= BasisVectors( Basis( M ) );
[ [ 1, 0 ], [ 0, 1 ] ]
gap> V:= SubAlgebraModule( M, [ bb[1] ] );
<left-module over <algebra over Rationals, with 2 generators>>
gap> Dimension( V );
1
```

62.11.17 LeftModuleByHomomorphismToMatAlg

▷ LeftModuleByHomomorphismToMatAlg(*A, hom*) (operation)

Here *A* is an algebra and *hom* a homomorphism from *A* into a matrix algebra. This function returns the left *A*-module defined by the homomorphism *hom*.

62.11.18 RightModuleByHomomorphismToMatAlg

▷ RightModuleByHomomorphismToMatAlg(*A, hom*) (operation)

Here *A* is an algebra and *hom* a homomorphism from *A* into a matrix algebra. This function returns the right *A*-module defined by the homomorphism *hom*.

First we produce a structure constants algebra with basis elements x, y, z such that $x^2 = x$, $y^2 = y$, $xz = z$, $zy = z$ and all other products are zero.

Example

```
gap> T:= EmptySCTable( 3, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1, 1 ]);
gap> SetEntrySCTable( T, 2, 2, [ 1, 2 ]);
gap> SetEntrySCTable( T, 1, 3, [ 1, 3 ]);
gap> SetEntrySCTable( T, 3, 2, [ 1, 3 ]);
gap> A:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 3 over Rationals>
```

Now we construct an isomorphic matrix algebra.

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> m3:= NullMat( 2, 2 );; m3[1][2]:= 1;;
gap> B:= Algebra( Rationals, [ m1, m2, m3 ] );
<algebra over Rationals, with 3 generators>
```

Finally we construct the homomorphism and the corresponding right module.

Example

```
gap> f:= AlgebraHomomorphismByImages( A, B, Basis(A), [ m1, m2, m3 ] );;
gap> RightModuleByHomomorphismToMatAlg( A, f );
<right-module over <algebra of dimension 3 over Rationals>>
```

62.11.19 AdjointModule

▷ `AdjointModule(`*A*`)` (attribute)

returns the *A*-module defined by the left action of *A* on itself.

Example

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> m3:= NullMat( 2, 2 );; m3[1][2]:= 1;;
gap> A:= Algebra( Rationals, [ m1, m2, m3 ] );
<algebra over Rationals, with 3 generators>
gap> V:= AdjointModule( A );
<3-dimensional left-module over <algebra of dimension
3 over Rationals>>
gap> v:= Basis( V )[3];
[ [ 0, 1 ], [ 0, 0 ] ]
gap> W:= SubAlgebraModule( V, [ v ] );
<left-module over <algebra of dimension 3 over Rationals>>
gap> Dimension( W );
1
```

62.11.20 FaithfulModule (for Lie algebras)

▷ `FaithfulModule(A)` (attribute)

returns a faithful finite-dimensional left-module over the algebra `A`. This is only implemented for associative algebras, and for Lie algebras of characteristic 0. (It may also work for certain Lie algebras of characteristic $p > 0$.)

Example

```
gap> T:= EmptySCTable( 2, 0 );;
gap> A:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 2 over Rationals>
```

Example

```
gap> T:= EmptySCTable( 3, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 3 ]);
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 3 over Rationals>
gap> V:= FaithfulModule( L );
<left-module over <Lie algebra of dimension 3 over Rationals>>
gap> vv:= BasisVectors( Basis( V ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> x:= Basis( L )[3];
v.3
gap> List( vv, v -> x^v );
[ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 0, 0 ] ]
```

`A` is a 2-dimensional algebra where all products are zero.

Example

```
gap> V:= FaithfulModule( A );
<left-module over <algebra of dimension 2 over Rationals>>
gap> vv:= BasisVectors( Basis( V ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> xx:= BasisVectors( Basis( A ) );
[ v.1, v.2 ]
gap> xx[1]^vv[3];
[ 1, 0, 0 ]
```

62.11.21 ModuleByRestriction

▷ `ModuleByRestriction(V, sub1[, sub2])` (operation)

Here `V` is an algebra module and *sub1* is a subalgebra of the acting algebra of `V`. This function returns the module that is the restriction of `V` to *sub1*. So it has the same underlying vector space as `V`, but the acting algebra is *sub*. If two subalgebras *sub1*, *sub2* are given then `V` is assumed to be a bi-module, and *sub1* a subalgebra of the algebra acting on the left, and *sub2* a subalgebra of the algebra acting on the right.

Example

```
gap> A:= Rationals^[3,3];;
gap> V:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> B:= Subalgebra( A, [ Basis(A)[1] ] );
<algebra over Rationals, with 1 generators>
```

```
gap> W:= ModuleByRestriction( V, B );
<left-module over <algebra over Rationals, with 1 generators>>
```

62.11.22 NaturalHomomorphismBySubAlgebraModule

▷ NaturalHomomorphismBySubAlgebraModule(*V*, *W*) (operation)

Here *V* must be a sub-algebra module of *V*. This function returns the projection from *V* onto *V*/*W*. It is a linear map, that is also a module homomorphism. As usual images can be formed with Image(f, v) and pre-images with PreImagesRepresentative(f, u).

The quotient module can also be formed by entering *V*/*W*.

Example

```
gap> A:= Rationals^[3,3];;
gap> B:= DirectSumOfAlgebras( A, A );
<algebra over Rationals, with 6 generators>
gap> T:= StructureConstantsTable( Basis( B ) );;
gap> C:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 18 over Rationals>
gap> V:= AdjointModule( C );
<left-module over <algebra of dimension 18 over Rationals>>
gap> W:= SubAlgebraModule( V, [ Basis(V)[1] ] );
<left-module over <algebra of dimension 18 over Rationals>>
gap> f:= NaturalHomomorphismBySubAlgebraModule( V, W );
<linear mapping by matrix, <
18-dimensional left-module over <algebra of dimension
18 over Rationals>> -> <
9-dimensional left-module over <algebra of dimension
18 over Rationals>>>
gap> quo:= ImagesSource( f );  # i.e., the quotient module
<9-dimensional left-module over <algebra of dimension
18 over Rationals>>
gap> v:= Basis( quo )[1];
[ 1, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> PreImagesRepresentative( f, v );
v.4
gap> Basis( C )[4]^v;
[ 1, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

62.11.23 DirectSumOfAlgebraModules (for a list of Lie algebra modules)

▷ DirectSumOfAlgebraModules(*list*) (operation)

▷ DirectSumOfAlgebraModules(*V*, *W*) (operation)

Here *list* must be a list of algebra modules. This function returns the direct sum of the elements in the list (as an algebra module). The modules must be defined over the same algebras.

In the second form is short for DirectSumOfAlgebraModules([*V*, *W*])

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> V:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [1,0,0] ] );;
gap> W:= DirectSumOfAlgebraModules( V, V );
```

```
<6-dimensional left-module over ( Rationals^[ 3, 3 ] )>
gap> BasisVectors( Basis( W ) );
[ ( [ 1, 0, 0 ] )(+)( [ 0, 0, 0 ] ), ( [ 0, 1, 0 ] )(+)( [ 0, 0, 0 ] )
    , ( [ 0, 0, 1 ] )(+)( [ 0, 0, 0 ] ),
  ( [ 0, 0, 0 ] )(+)( [ 1, 0, 0 ] ), ( [ 0, 0, 0 ] )(+)( [ 0, 1, 0 ] )
    , ( [ 0, 0, 0 ] )(+)( [ 0, 0, 1 ] ) ]
```

Example

```
gap> L:= SimpleLieAlgebra( "C", 3, Rationals );;
gap> V:= HighestWeightModule( L, [ 1, 1, 0 ] );
<64-dimensional left-module over <Lie algebra of dimension
21 over Rationals>>
gap> W:= HighestWeightModule( L, [ 0, 0, 2 ] );
<84-dimensional left-module over <Lie algebra of dimension
21 over Rationals>>
gap> U:= DirectSumOfAlgebraModules( V, W );
<148-dimensional left-module over <Lie algebra of dimension
21 over Rationals>>
```

62.11.24 TranslatorSubalgebra

▷ `TranslatorSubalgebra(`*M, U, W*`)` (operation)

Here *M* is an algebra module, and *U* and *W* are two subspaces of *M*. Let *A* be the algebra acting on *M*. This function returns the subspace of elements of *A* that map *U* into *W*. If *W* is a sub-algebra-module (i.e., closed under the action of *A*), then this space is a subalgebra of *A*.

This function works for left, or right modules over a finite-dimensional algebra. We stress that it is not checked whether *U* and *W* are indeed subspaces of *M*. If this is not the case nothing is guaranteed about the behaviour of the function.

Example

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );
( Rationals^[ 3, 3 ] )
gap> V:= Rationals^[3,2];
( Rationals^[ 3, 2 ] )
gap> M:= LeftAlgebraModule( A, \*, V );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> bm:= Basis(M);;
gap> U:= SubAlgebraModule( M, [ bm[1] ] );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> TranslatorSubalgebra( M, U, M );
<algebra of dimension 9 over Rationals>
gap> W:= SubAlgebraModule( M, [ bm[4] ] );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> T:=TranslatorSubalgebra( M, U, W );
<algebra of dimension 0 over Rationals>
```

Chapter 63

Finitely Presented Algebras

Currently the GAP library contains only few functions dealing with general finitely presented algebras, so this chapter is merely a placeholder.

The special case of finitely presented *Lie* algebras is described in 64.11, and there is also a GAP package `fplsa` for computing structure constants of *f*initely *p*resented *L*ie (*s*uper)*a*lgebras.

Chapter 64

Lie Algebras

A Lie algebra L is an algebra such that $xx = 0$ and $x(yz)+y(zx)+z(xy) = 0$ for all $x, y, z \in L$. A common way of creating a Lie algebra is by taking an associative algebra together with the commutator as product. Therefore the product of two elements x, y of a Lie algebra is usually denoted by $[x, y]$, but in GAP this denotes the list of the elements x and y; hence the product of elements is made by the usual `*`. This gives no problems when dealing with Lie algebras given by a table of structure constants. However, for matrix Lie algebras the situation is not so easy as `*` denotes the ordinary (associative) matrix multiplication. In GAP this problem is solved by wrapping elements of a matrix Lie algebra up as `LieObjects`, and then define the `*` for `LieObjects` to be the commutator (see 64.1).

64.1 Lie Objects

Let `x` be a ring element, then `LieObject(x)` (see `LieObject` (64.1.1)) wraps `x` up into an object that contains the same data (namely `x`). The multiplication `*` for Lie objects is formed by taking the commutator. More exactly, if `l1` and `l2` are the Lie objects corresponding to the ring elements `r1` and `r2`, then `l1 * l2` is equal to the Lie object corresponding to `r1 * r2 - r2 * r1`. Two rules for Lie objects are worth noting:

- An element is *not* equal to its Lie element.
- If we take the Lie object of an ordinary (associative) matrix then this is again a matrix; it is therefore a collection (of its rows) and a list. But it is *not* a collection of collections of its entries, and its family is *not* a collections family.

Given a family `F` of ring elements, we can form its Lie family `L`. The elements of `F` and `L` are in bijection, only the multiplications via `*` differ for both families. More exactly, if `l1` and `l2` are the Lie elements corresponding to the elements `f1` and `f2` in `F`, we have `l1 * l2` equal to the Lie element corresponding to `f1 * f2 - f2 * f1`. Furthermore, the product of Lie elements `l1`, `l2` and `l3` is left-normed, that is `l1*l2*l3` is equal to `(l1*l2)*l3`.

The main reason to distinguish elements and Lie elements on the family level is that this helps to avoid forming domains that contain elements of both types. For example, if we could form vector spaces of matrices then at first glance it would be no problem to have both ordinary and Lie matrices in it, but as soon as we find out that the space is in fact an algebra (e.g., because its dimension is that of the full matrix algebra), we would run into strange problems.

Note that the family situation with Lie families may be not familiar.

- We have to be careful when installing methods for certain types of domains that may involve Lie elements. For example, the zero element of a matrix space is either an ordinary matrix or its Lie element, depending on the space. So either the method must be aware of both cases, or the method selection must distinguish the two cases. In the latter situation, only one method may be applicable to each case; this means that it is not sufficient to treat the Lie case with the additional requirement `IsLieObjectCollection` but that we must explicitly require non-Lie elements for the non-Lie case.
- Being a full matrix space is a property that may hold for a space of ordinary matrices or a space of Lie matrices. So methods for full matrix spaces must also be aware of Lie matrices.

64.1.1 LieObject

▷ `LieObject(`*obj*`)` (attribute)

Let *obj* be a ring element. Then `LieObject( obj )` is the corresponding Lie object. If *obj* lies in the family `F`, then `LieObject( obj )` lies in the family `LieFamily( F )` (see `LieFamily` (64.1.3)).

Example

```
gap> m:= [ [ 1, 0 ], [ 0, 1 ] ];;
gap> lo:= LieObject( m );
LieObject( [ [ 1, 0 ], [ 0, 1 ] ] )
gap> m*m;
[ [ 1, 0 ], [ 0, 1 ] ]
gap> lo*lo;
LieObject( [ [ 0, 0 ], [ 0, 0 ] ] )
```

64.1.2 IsLieObject

▷ `IsLieObject(`*obj*`)` (Category)
▷ `IsLieObjectCollection(`*obj*`)` (Category)
▷ `IsRestrictedLieObject(`*obj*`)` (Category)
▷ `IsRestrictedLieObjectCollection(`*obj*`)` (Category)

An object lies in `IsLieObject` if and only if it lies in a family constructed by `LieFamily` (64.1.3).

Example

```
gap> m:= [ [ 1, 0 ], [ 0, 1 ] ];;
gap> lo:= LieObject( m );
LieObject( [ [ 1, 0 ], [ 0, 1 ] ] )
gap> IsLieObject( m );
false
gap> IsLieObject( lo );
true
```

64.1.3 LieFamily

▷ `LieFamily(`*Fam*`)` (attribute)

is a family F in bijection with the family *Fam*, but with the Lie bracket as infix multiplication. That is, for x, y in *Fam*, the product of the images in F will be the image of x * y - y * x.

The standard type of objects in a Lie family F is *F*!.packedType.

The bijection from *Fam* to F is given by Embedding(*Fam*, F) (see Embedding (32.2.10)); this bijection respects addition and additive inverses.

64.1.4 UnderlyingFamily

▷ UnderlyingFamily(*Fam*) (attribute)

If *Fam* is a Lie family then UnderlyingFamily(*Fam*) is a family F such that *Fam* = LieFamily(F).

64.1.5 UnderlyingRingElement

▷ UnderlyingRingElement(*obj*) (attribute)

Let *obj* be a Lie object constructed from a ring element r by calling LieObject(r). Then UnderlyingRingElement(*obj*) returns the ring element r used to construct *obj*. If r lies in the family F, then *obj* lies in the family LieFamily(F) (see LieFamily (64.1.3)).

Example

```
gap> lo:= LieObject( [ [ 1, 0 ], [ 0, 1 ] ] );
LieObject( [ [ 1, 0 ], [ 0, 1 ] ] )
gap> m:=UnderlyingRingElement(lo);
[ [ 1, 0 ], [ 0, 1 ] ]
gap> lo*lo;
LieObject( [ [ 0, 0 ], [ 0, 0 ] ] )
gap> m*m;
[ [ 1, 0 ], [ 0, 1 ] ]
```

64.2 Constructing Lie algebras

In this section we describe functions that create Lie algebras. Creating and working with subalgebras goes exactly in the same way as for general algebras; so for that we refer to Chapter 62.

64.2.1 LieAlgebraByStructureConstants

▷ LieAlgebraByStructureConstants(*R, sct[, nameinfo]*) (function)

LieAlgebraByStructureConstants does the same as AlgebraByStructureConstants (62.4.1), and has the same meaning of arguments, except that the result is assumed to be a Lie algebra. Note that the function does not check whether *sct* satisfies the Jacobi identity. (So if one creates a Lie algebra this way with a table that does not satisfy the Jacobi identity, errors may occur later on.)

Example

```
gap> T:= EmptySCTable( 2, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1/2, 1 ] );
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 2 over Rationals>
```

64.2.2 RestrictedLieAlgebraByStructureConstants

▷ `RestrictedLieAlgebraByStructureConstants(`*R, sct[, nameinfo], pmapping*`)` (function)

`RestrictedLieAlgebraByStructureConstants` does the same as `LieAlgebraByStructureConstants` (64.2.1), and has the same meaning of all arguments, except that the result is assumed to be a restricted Lie algebra (see 64.8) with the p-map given by the additional argument *pmapping*. This last argument is a list of the length equal to the dimension of the algebra; its i-th entry specifies the p-th power of the i-th basis vector in the same format `[ coeff1, position1, coeff2, position2, ... ]` as `SetEntrySCTable` (62.4.4) uses to specify entries of the structure constants table.

Note that the function does not check whether *sct* satisfies the Jacobi identity, of whether *pmapping* specifies a legitimate p-mapping.

The following example creates a commutative restricted Lie algebra of dimension 3, in which the p-th power of the i-th basis element is the $i+1$-th basis element (except for the 3rd basis element which goes to zero).

Example

```
gap> T:= EmptySCTable( 3, Zero(GF(5)), "antisymmetric" );;
gap> L:= RestrictedLieAlgebraByStructureConstants(
>                                     GF(5), T, [[1,2],[1,3],[]] );
<Lie algebra of dimension 3 over GF(5)>
gap> List(Basis(L),PthPowerImage);
[ v.2, v.3, 0*v.1 ]
gap> PthPowerImage(L.1+L.2);
v.2+v.3
```

64.2.3 LieAlgebra (for an associative algebra)

▷ `LieAlgebra(`*L*`)` (function)

▷ `LieAlgebra(`*F, gens[, zero][, "basis"]*`)` (function)

For an associative algebra *L*, `LieAlgebra( L )` is the Lie algebra isomorphic to *L* as a vector space but with the Lie bracket as product.

`LieAlgebra( F, gens )` is the Lie algebra over the division ring *F*, generated *as Lie algebra* by the Lie objects corresponding to the vectors in the list *gens*.

Note that the algebra returned by `LieAlgebra` does not contain the vectors in *gens*. The elements in *gens* are wrapped up as Lie objects (see 64.1). This allows one to create Lie algebras from ring elements with respect to the Lie bracket as product. But of course the product in the Lie algebra is the usual `*`.

If there are three arguments, a division ring *F* and a list *gens* and an element *zero*, then `LieAlgebra( F, gens, zero )` is the corresponding *F*-Lie algebra with zero element the Lie object corresponding to *zero*.

If the last argument is the string `"basis"` then the vectors in *gens* are known to form a basis of the algebra (as an *F*-vector space).

Note that even if each element in *gens* is already a Lie element, i.e., is of the form `LieElement( elm )` for an object *elm*, the elements of the result lie in the Lie family of the family that contains *gens* as a subset.

Example

```
gap> A:= FullMatrixAlgebra( GF( 7 ), 4 );;
gap> L:= LieAlgebra( A );
<Lie algebra of dimension 16 over GF(7)>
gap> mats:= [ [ [ 1, 0 ], [ 0, -1 ] ], [ [ 0, 1 ], [ 0, 0 ] ],
>             [ [ 0, 0 ], [ 1, 0] ] ];;
gap> L:= LieAlgebra( Rationals, mats );
<Lie algebra over Rationals, with 3 generators>
```

64.2.4 FreeLieAlgebra (for ring, rank (and name))

▷ FreeLieAlgebra(*R, rank[, name]*) (function)
▷ FreeLieAlgebra(*R, name1, name2, ...*) (function)

Returns a free Lie algebra of rank *rank* over the ring *R*. FreeLieAlgebra(*R*, *name1*, *name2*,...) returns a free Lie algebra over *R* with generators named *name1*, *name2*, and so on. The elements of a free Lie algebra are written on the Hall-Lyndon basis.

Example

```
gap> L:= FreeLieAlgebra( Rationals, "x", "y", "z" );
<Lie algebra over Rationals, with 3 generators>
gap> g:= GeneratorsOfAlgebra( L );; x:= g[1];; y:=g[2];; z:= g[3];;
gap> z*(y*(x*(z*y)));
(-1)*((x*(y*z))*(y*z))+(-1)*((x*((y*z)*z))*y)+(-1)*(((x*z)*(y*z))*y)
```

64.2.5 FullMatrixLieAlgebra

▷ FullMatrixLieAlgebra(*R, n*) (function)
▷ MatrixLieAlgebra(*R, n*) (function)
▷ MatLieAlgebra(*R, n*) (function)

is the full matrix Lie algebra of $n \times n$ matrices over the ring *R*, for a nonnegative integer *n*.

Example

```
gap> FullMatrixLieAlgebra( GF(9), 10 );
<Lie algebra over GF(3^2), with 19 generators>
```

64.2.6 RightDerivations

▷ RightDerivations(*B*) (attribute)
▷ LeftDerivations(*B*) (attribute)
▷ Derivations(*B*) (attribute)

These functions all return the matrix Lie algebra of derivations of the algebra A with basis *B*.

RightDerivations(*B*) returns the algebra of derivations represented by their right action on the algebra A. This means that with respect to the basis B of A, the derivation D is described by the matrix $[d_{i,j}]$ which means that D maps the i-th basis element b_i to $\sum_{j=1}^{n} d_{i,j} b_j$.

LeftDerivations(*B*) returns the Lie algebra of derivations represented by their left action on the algebra A. So the matrices contained in the algebra output by LeftDerivations(*B*) are the transposes of the matrices contained in the output of RightDerivations(*B*).

Derivations is just a synonym for RightDerivations.

Example

```
gap> A:= OctaveAlgebra( Rationals );
<algebra of dimension 8 over Rationals>
gap> L:= Derivations( Basis( A ) );
<Lie algebra of dimension 14 over Rationals>
```

64.2.7 SimpleLieAlgebra

▷ SimpleLieAlgebra(*type*, *n*, *F*) (function)

This function constructs the simple Lie algebra of type given by the string *type* and rank *n* over the field *F*. The string *type* must be one of "A", "B", "C", "D", "E", "F", "G", "H", "K", "S", "W" or "M". For the types A to G, *n* must be a positive integer. The last five types only exist over fields of characteristic $p > 0$. If the type is H, then *n* must be a list of positive integers of even length. If the type is K, then *n* must be a list of positive integers of odd length. For the types S and W, *n* must be a list of positive integers of any length. If the type is M, then the Melikyan algebra is constructed. In this case *n* must be a list of two positive integers. This Lie algebra only exists over fields of characteristic 5. This Lie algebra is $\mathbb{Z} \times \mathbb{Z}$ graded; and the grading can be accessed via the attribute Grading(L) (see Grading (62.9.20)). In some cases the Lie algebra returned by this function is not simple. Examples are the Lie algebras of type A_n over a field of characteristic $p > 0$ where p divides $n+1$, and the Lie algebras of type K_n where n is a list of length 1.

If *type* is one of A, B, C, D, E, F, G, and *F* is a field of characteristic zero, then the basis of the returned Lie algebra is a Chevalley basis.

Example

```
gap> SimpleLieAlgebra( "E", 6, Rationals );
<Lie algebra of dimension 78 over Rationals>
gap> SimpleLieAlgebra( "A", 6, GF(5) );
<Lie algebra of dimension 48 over GF(5)>
gap> SimpleLieAlgebra( "W", [1,2], GF(5) );
<Lie algebra of dimension 250 over GF(5)>
gap> SimpleLieAlgebra( "H", [1,2], GF(5) );
<Lie algebra of dimension 123 over GF(5)>
gap> L:= SimpleLieAlgebra( "M", [1,1], GF(5) );
<Lie algebra of dimension 125 over GF(5)>
```

64.3 Distinguished Subalgebras

Here we describe functions that calculate well-known subalgebras and ideals of a Lie algebra (such as the centre, the centralizer of a subalgebra, etc.).

64.3.1 LieCentre

▷ LieCentre(*L*) (attribute)
▷ LieCenter(*L*) (attribute)

The *Lie* centre of the Lie algebra *L* is the kernel of the adjoint mapping, that is, the set $\{a \in L : \forall x \in L : ax = 0\}$.

In characteristic 2 this may differ from the usual centre (that is the set of all $a \in L$ such that $ax = xa$ for all $x \in L$). Therefore, this operation is named `LieCentre` and not `Centre` (35.4.5).

Example

```
gap> L:= FullMatrixLieAlgebra( GF(3), 3 );
<Lie algebra over GF(3), with 5 generators>
gap> LieCentre( L );
<two-sided ideal in <Lie algebra of dimension 9 over GF(3)>,
  (dimension 1)>
```

64.3.2 LieCentralizer

▷ `LieCentralizer(L, S)` (operation)

is the annihilator of S in the Lie algebra L, that is, the set $\{a \in L : \forall s \in S : a * s = 0\}$. Here S may be a subspace or a subalgebra of L.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> b:= BasisVectors( Basis( L ) );;
gap> LieCentralizer( L, Subalgebra( L, [ b[1], b[2] ] ) );
<Lie algebra of dimension 1 over Rationals>
```

64.3.3 LieNormalizer

▷ `LieNormalizer(L, U)` (operation)

is the normalizer of the subspace U in the Lie algebra L, that is, the set $N_L(U) = \{x \in L : [x,U] \subset U\}$.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> b:= BasisVectors( Basis( L ) );;
gap> LieNormalizer( L, Subalgebra( L, [ b[1], b[2] ] ) );
<Lie algebra of dimension 8 over Rationals>
```

64.3.4 LieDerivedSubalgebra

▷ `LieDerivedSubalgebra(L)` (attribute)

is the (Lie) derived subalgebra of the Lie algebra L.

Example

```
gap>  L:= FullMatrixLieAlgebra( GF( 3 ), 3 );
<Lie algebra over GF(3), with 5 generators>
gap> LieDerivedSubalgebra( L );
<Lie algebra of dimension 8 over GF(3)>
```

64.3.5 LieNilRadical

▷ LieNilRadical(*L*) (attribute)

This function calculates the (Lie) nil radical of the Lie algebra *L*.

Example

```
gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:= LieAlgebra( Rationals, mats );;
gap> LieNilRadical( L );
<two-sided ideal in <Lie algebra of dimension 3 over Rationals>,
  (dimension 2)>
```

64.3.6 LieSolvableRadical

▷ LieSolvableRadical(*L*) (attribute)

Returns the (Lie) solvable radical of the Lie algebra *L*.

Example

```
gap> L:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> LieSolvableRadical( L );
<two-sided ideal in <Lie algebra of dimension 9 over Rationals>,
  (dimension 1)>
```

64.3.7 CartanSubalgebra

▷ CartanSubalgebra(*L*) (attribute)

A Cartan subalgebra of a Lie algebra *L* is defined as a nilpotent subalgebra of *L* equal to its own Lie normalizer in *L*.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> CartanSubalgebra( L );
<Lie algebra of dimension 2 over Rationals>
```

64.4 Series of Ideals

64.4.1 LieDerivedSeries

▷ LieDerivedSeries(*L*) (attribute)

is the (Lie) derived series of the Lie algebra *L*.

Example

```
gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:= LieAlgebra( Rationals, mats );;
gap> LieDerivedSeries( L );
[ <Lie algebra of dimension 3 over Rationals>,
  <Lie algebra of dimension 1 over Rationals>,
  <Lie algebra of dimension 0 over Rationals> ]
```

64.4.2 LieLowerCentralSeries

▷ LieLowerCentralSeries(*L*) (attribute)

is the (Lie) lower central series of the Lie algebra *L*.

Example

```
gap> mats:= [ [[ 1, 0 ], [ 0, 0 ]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:=LieAlgebra( Rationals, mats );;
gap> LieLowerCentralSeries( L );
[ <Lie algebra of dimension 3 over Rationals>,
  <Lie algebra of dimension 1 over Rationals> ]
```

64.4.3 LieUpperCentralSeries

▷ LieUpperCentralSeries(*L*) (attribute)

is the (Lie) upper central series of the Lie algebra *L*.

Example

```
gap> mats:= [ [[ 1, 0 ], [ 0, 0 ]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:=LieAlgebra( Rationals, mats );;
gap> LieUpperCentralSeries( L );
[ <two-sided ideal in <Lie algebra of dimension 3 over Rationals>,
      (dimension 1)>, <Lie algebra over Rationals, with 0 generators>
 ]
```

64.5 Properties of a Lie Algebra

64.5.1 IsLieAbelian

▷ IsLieAbelian(*L*) (property)

returns `true` if *L* is a Lie algebra such that each product of elements in *L* is zero, and `false` otherwise.

Example

```
gap>  T:= EmptySCTable( 5, 0, "antisymmetric" );;
gap>  L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 5 over Rationals>
gap> IsLieAbelian( L );
true
```

64.5.2 IsLieNilpotent

▷ IsLieNilpotent(*L*) (property)

A Lie algebra *L* is defined to be (Lie) *nilpotent* when its (Lie) lower central series reaches the trivial subalgebra.

Example

```
gap> T:= EmptySCTable( 5, 0, "antisymmetric" );;
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
```

```
<Lie algebra of dimension 5 over Rationals>
gap> IsLieNilpotent( L );
true
```

64.5.3 IsLieSolvable

▷ `IsLieSolvable(L)` (property)

A Lie algebra L is defined to be (Lie) *solvable* when its (Lie) derived series reaches the trivial subalgebra.

Example

```
gap> T:= EmptySCTable( 5, 0, "antisymmetric" );;
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 5 over Rationals>
gap> IsLieSolvable( L );
true
```

64.6 Semisimple Lie Algebras and Root Systems

This section contains some functions for dealing with semisimple Lie algebras and their root systems.

64.6.1 SemiSimpleType

▷ `SemiSimpleType(L)` (attribute)

Let L be a semisimple Lie algebra, i.e., a direct sum of simple Lie algebras. Then `SemiSimpleType` returns the type of L, i.e., a string containing the types of the simple summands of L.

Example

```
gap> L:= SimpleLieAlgebra( "E", 8, Rationals );;
gap> b:= BasisVectors( Basis( L ) );;
gap> K:= LieCentralizer(L, Subalgebra(L, [ b[61]+b[79]+b[101]+b[102] ]));
<Lie algebra of dimension 102 over Rationals>
gap> lev:= LeviMalcevDecomposition(K);;
gap> SemiSimpleType( lev[1] );
"B3 A1"
```

64.6.2 ChevalleyBasis

▷ `ChevalleyBasis(L)` (attribute)

Here L must be a semisimple Lie algebra with a split Cartan subalgebra. Then `ChevalleyBasis(L)` returns a list consisting of three sublists. Together these sublists form a Chevalley basis of L. The first list contains the positive root vectors, the second list contains the negative root vectors, and the third list the Cartan elements of the Chevalley basis.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
```

```
gap> ChevalleyBasis( L );
[ [ v.1, v.2, v.3, v.4, v.5, v.6 ],
  [ v.7, v.8, v.9, v.10, v.11, v.12 ], [ v.13, v.14 ] ]
```

64.6.3 IsRootSystem

▷ IsRootSystem(*obj*) (Category)

Category of root systems.

64.6.4 IsRootSystemFromLieAlgebra

▷ IsRootSystemFromLieAlgebra(*obj*) (Category)

Category of root systems that come from (semisimple) Lie algebras. They often have special attributes such as UnderlyingLieAlgebra (64.6.6), PositiveRootVectors (64.6.9), NegativeRootVectors (64.6.10), CanonicalGenerators (64.6.14).

64.6.5 RootSystem

▷ RootSystem(*L*) (attribute)

RootSystem calculates the root system of the semisimple Lie algebra *L* with a split Cartan subalgebra.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> R:= RootSystem( L );
<root system of rank 2>
gap> IsRootSystem( R );
true
gap> IsRootSystemFromLieAlgebra( R );
true
```

64.6.6 UnderlyingLieAlgebra

▷ UnderlyingLieAlgebra(*R*) (attribute)

For a root system *R* coming from a semisimple Lie algebra L, returns the Lie algebra L.

64.6.7 PositiveRoots

▷ PositiveRoots(*R*) (attribute)

The list of positive roots of the root system *R*.

64.6.8 NegativeRoots

▷ `NegativeRoots(R)` (attribute)

The list of negative roots of the root system `R`.

64.6.9 PositiveRootVectors

▷ `PositiveRootVectors(R)` (attribute)

A list of positive root vectors of the root system `R` that comes from a Lie algebra `L`. This is a list in bijection with the list `PositiveRoots( L )` (see `PositiveRoots` (64.6.7)). The root vector is a non-zero element of the root space (in `L`) of the corresponding root.

64.6.10 NegativeRootVectors

▷ `NegativeRootVectors(R)` (attribute)

A list of negative root vectors of the root system `R` that comes from a Lie algebra `L`. This is a list in bijection with the list `NegativeRoots( L )` (see `NegativeRoots` (64.6.8)). The root vector is a non-zero element of the root space (in `L`) of the corresponding root.

64.6.11 SimpleSystem

▷ `SimpleSystem(R)` (attribute)

A list of simple roots of the root system `R`.

64.6.12 CartanMatrix

▷ `CartanMatrix(R)` (attribute)

The Cartan matrix of the root system `R`, relative to the simple roots in `SimpleSystem( R )` (see `SimpleSystem` (64.6.11)).

64.6.13 BilinearFormMat

▷ `BilinearFormMat(R)` (attribute)

The matrix of the bilinear form of the root system `R`. If we denote this matrix by B, then we have $B(i,j) = (\alpha_i, \alpha_j)$, where the α_i are the simple roots of `R`.

64.6.14 CanonicalGenerators

▷ `CanonicalGenerators(R)` (attribute)

Here `R` must be a root system coming from a semisimple Lie algebra `L`. This function returns $3l$ generators of `L`, $x_1, \ldots, x_l, y_1, \ldots, y_l, h_1, \ldots, h_l$, where x_i lies in the root space corresponding to the i-th simple root of the root system of `L`, y_i lies in the root space corresponding to $-$ the i-th simple root,

and the h_i are elements of the Cartan subalgebra. These elements satisfy the relations $h_i * h_j = 0$, $x_i * y_j = \delta_{ij} h_i$, $h_j * x_i = c_{ij} x_i$, $h_j * y_i = -c_{ij} y_i$, where c_{ij} is the entry of the Cartan matrix on position ij.

Also if a is a root of the root system `R` (so a is a list of numbers), then we have the relation $h_i * x = a[i]x$, where x is a root vector corresponding to a.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> R:= RootSystem( L );;
gap> UnderlyingLieAlgebra( R );
<Lie algebra of dimension 14 over Rationals>
gap> PositiveRoots( R );
[ [ 2, -1 ], [ -3, 2 ], [ -1, 1 ], [ 1, 0 ], [ 3, -1 ], [ 0, 1 ] ]
gap> x:= PositiveRootVectors( R );
[ v.1, v.2, v.3, v.4, v.5, v.6 ]
gap> g:=CanonicalGenerators( R );
[ [ v.1, v.2 ], [ v.7, v.8 ], [ v.13, v.14 ] ]
gap> g[3][1]*x[1];
(2)*v.1
gap> g[3][2]*x[1];
(-1)*v.1
gap> # i.e., x[1] is the root vector belonging to the root [ 2, -1 ]
gap> BilinearFormMat( R );
[ [ 1/12, -1/8 ], [ -1/8, 1/4 ] ]
```

64.7 Semisimple Lie Algebras and Weyl Groups of Root Systems

This section deals with the Weyl group of a root system. A Weyl group is represented by its action on the weight lattice. A *weight* is by definition a linear function $\lambda : H \to F$ (where F is the ground field), such that the values $\lambda(h_i)$ are all integers (where the h_i are the Cartan elements of the `CanonicalGenerators` (64.6.14)). On the other hand each weight is determined by these values. Therefore we represent a weight by a vector of integers; the i-th entry of this vector is the value $\lambda(h_i)$. Now the elements of the Weyl group are represented by matrices, and if `g` is an element of a Weyl group and `w` a weight, then `w*g` gives the result of applying `g` to `w`. Another way of applying the i-th simple reflection to a weight is by using the function `ApplySimpleReflection` (64.7.4).

A Weyl group is generated by the simple reflections. So `GeneratorsOfGroup` (39.2.4) for a Weyl group `W` gives a list of matrices and the i-th entry of this list is the simple reflection corresponding to the i-th simple root of the corresponding root system.

64.7.1 IsWeylGroup

▷ `IsWeylGroup(G)` (property)

A Weyl group is a group generated by reflections, with the attribute `SparseCartanMatrix` (64.7.2) set.

64.7.2 SparseCartanMatrix

▷ `SparseCartanMatrix(W)` (attribute)

This is a sparse form of the Cartan matrix of the corresponding root system. If we denote the Cartan matrix by C, then the sparse Cartan matrix of *W* is a list (of length equal to the length of the Cartan matrix), where the i-th entry is a list consisting of elements `[ j, C[i][j] ]`, where j is such that `C[i][j]` is non-zero.

64.7.3 WeylGroup

▷ `WeylGroup(`*R*`)` (attribute)

The Weyl group of the root system *R*. It is generated by the simple reflections. A simple reflection is represented by a matrix, and the result of letting a simple reflection m act on a weight w is obtained by `w*m`.

Example

```
gap> L:= SimpleLieAlgebra( "F", 4, Rationals );;
gap> R:= RootSystem( L );;
gap> W:= WeylGroup( R );
<matrix group with 4 generators>
gap> IsWeylGroup( W );
true
gap> SparseCartanMatrix( W );
[ [ [ 1, 2 ], [ 3, -1 ] ], [ [ 2, 2 ], [ 4, -1 ] ],
  [ [ 1, -1 ], [ 3, 2 ], [ 4, -1 ] ],
  [ [ 2, -1 ], [ 3, -2 ], [ 4, 2 ] ] ]
gap> g:= GeneratorsOfGroup( W );;
gap> [ 1, 1, 1, 1 ]*g[2];
[ 1, -1, 1, 2 ]
```

64.7.4 ApplySimpleReflection

▷ `ApplySimpleReflection(`*SC, i, wt*`)` (operation)

Here *SC* is the sparse Cartan matrix of a Weyl group. This function applies the *i*-th simple reflection to the weight *wt*, thus changing *wt*.

Example

```
gap> L:= SimpleLieAlgebra( "F", 4, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> C:= SparseCartanMatrix( W );;
gap> w:= [ 1, 1, 1, 1 ];;
gap> ApplySimpleReflection( C, 2, w );
gap> w;
[ 1, -1, 1, 2 ]
```

64.7.5 LongestWeylWordPerm

▷ `LongestWeylWordPerm(`*W*`)` (attribute)

Let g_0 be the longest element in the Weyl group *W*, and let $\{\alpha_1, \ldots, \alpha_l\}$ be a simple system of the corresponding root system. Then g_0 maps α_i to $-\alpha_{\sigma(i)}$, where σ is a permutation of $(1, \ldots, l)$. This function returns that permutation.

Example

```
gap> L:= SimpleLieAlgebra( "E", 6, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> LongestWeylWordPerm( W );
(1,6)(3,5)
```

64.7.6 ConjugateDominantWeight

▷ `ConjugateDominantWeight(`*W, wt*`)` (operation)
▷ `ConjugateDominantWeightWithWord(`*W, wt*`)` (operation)

Here *W* is a Weyl group and *wt* a weight (i.e., a list of integers). `ConjugateDominantWeight` returns the unique dominant weight conjugate to *wt* under *W*.

`ConjugateDominantWeightWithWord` returns a list of two elements. The first of these is the dominant weight conjugate to *wt*. The second element is a list of indices of simple reflections that have to be applied to *wt* in order to get the dominant weight conjugate to it.

Example

```
gap> L:= SimpleLieAlgebra( "E", 6, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> C:= SparseCartanMatrix( W );;
gap> w:= [ 1, -1, 2, -2, 3, -3 ];;
gap> ConjugateDominantWeight( W, w );
[ 2, 1, 0, 0, 0, 0 ]
gap> c:= ConjugateDominantWeightWithWord( W, w );
[ [ 2, 1, 0, 0, 0, 0 ], [ 2, 4, 2, 3, 6, 5, 4, 2, 3, 1 ] ]
gap> for i in [1..Length(c[2])] do
> ApplySimpleReflection( C, c[2][i], w );
> od;
gap> w;
[ 2, 1, 0, 0, 0, 0 ]
```

64.7.7 WeylOrbitIterator

▷ `WeylOrbitIterator(`*W, wt*`)` (operation)

Returns an iterator for the orbit of the weight *wt* under the action of the Weyl group *W*.

Example

```
gap> L:= SimpleLieAlgebra( "E", 6, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> orb:= WeylOrbitIterator( W, [ 1, 1, 1, 1, 1, 1 ] );
<iterator>
gap> NextIterator( orb );
[ 1, 1, 1, 1, 1, 1 ]
gap> NextIterator( orb );
[ -1, -1, -1, -1, -1, -1 ]
gap> orb:= WeylOrbitIterator( W, [ 1, 1, 1, 1, 1, 1 ] );
<iterator>
gap> k:= 0;
0
gap> while not IsDoneIterator( orb ) do
```

```
> w:= NextIterator( orb ); k:= k+1;
> od;
gap> k;  # this is the size of the Weyl group of E6
51840
```

64.8 Restricted Lie algebras

A Lie algebra L over a field of characteristic $p > 0$ is called restricted if there is a map $x \mapsto x^p$ from L into L (called a p-map) such that $\mathrm{ad}\, x^p = (\mathrm{ad}x)^p$, $(\alpha x)^p = \alpha^p x^p$ and $(x+y)^p = x^p + y^p + \sum_{i=1}^{p-1} s_i(x,y)$, where $s_i : L \times L \to L$ are certain Lie polynomials in two variables. Using these relations we can calculate y^p for all $y \in L$, once we know x^p for x in a basis of L. Therefore a p-map is represented in GAP by a list containing the images of the basis vectors of a basis B of L. For this reason this list is an attribute of the basis B.

64.8.1 IsRestrictedLieAlgebra

▷ `IsRestrictedLieAlgebra(`*L*`)` (property)

Test whether *L* is restricted.

Example

```
gap> L:= SimpleLieAlgebra( "W", [2], GF(5));
<Lie algebra of dimension 25 over GF(5)>
gap> IsRestrictedLieAlgebra( L );
false
gap> L:= SimpleLieAlgebra( "W", [1], GF(5));
<Lie algebra of dimension 5 over GF(5)>
gap> IsRestrictedLieAlgebra( L );
true
```

64.8.2 PthPowerImages

▷ `PthPowerImages(`*B*`)` (attribute)

Here *B* is a basis of a restricted Lie algebra. This function returns the list of the images of the basis vectors of *B* under the p-map.

Example

```
gap> L:= SimpleLieAlgebra( "W", [1], GF(11) );
<Lie algebra of dimension 11 over GF(11)>
gap> B:= Basis( L );
CanonicalBasis( <Lie algebra of dimension 11 over GF(11)> )
gap> PthPowerImages( B );
[ 0*v.1, v.2, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1,
  0*v.1 ]
```

64.8.3 PthPowerImage (for basis and element)

▷ `PthPowerImage(`*B*`, `*x*`)` (operation)

▷ `PthPowerImage(`*x*`)` (operation)

▷ PthPowerImage(*x*, *n*) (operation)

This function computes the image of an element *x* of a restricted Lie algebra under its p-map.

In the first form, a basis of the Lie algebra is provided; this basis stores the pth powers of its elements. It is the traditional form, provided for backwards compatibility.

In its second form, only the element *x* is provided. It is the only form for elements of Lie algebras with no predetermined basis, such as those constructed by LieObject (64.1.1).

In its third form, an extra non-negative integer *n* is specified; the p-mapping is iterated *n* times on the element *x*.

Example

```
gap> L:= SimpleLieAlgebra( "W", [1], GF(11) );;
gap> B:= Basis( L );;
gap> x:= B[1]+B[11];
v.1+v.11
gap> PthPowerImage( B, x );
v.1+v.11
gap> PthPowerImage( x, 2 );
v.1+v.11
gap> f := FreeAssociativeAlgebra(GF(2),"x","y");
<algebra over GF(2), with 2 generators>
gap> x := LieObject(f.1);; y := LieObject(f.2);;
gap> x*y; x^2; PthPowerImage(x);
LieObject( (Z(2)^0)*x*y+(Z(2)^0)*y*x )
LieObject( <zero> of ... )
LieObject( (Z(2)^0)*x^2 )
```

64.8.4 JenningsLieAlgebra

▷ JenningsLieAlgebra(*G*) (attribute)

Let *G* be a nontrivial p-group, and let $G = G_1 \supset G_2 \supset \cdots \supset G_m = 1$ be its Jennings series (see JenningsSeries (39.17.14)). Then the quotients G_i/G_{i+1} are elementary abelian p-groups, i.e., they can be viewed as vector spaces over GF(p). Now the Jennings-Lie algebra L of *G* is the direct sum of those vector spaces. The Lie bracket on L is induced by the commutator in *G*. Furthermore, the map $g \mapsto g^p$ in *G* induces a p-map in L making L into a restricted Lie algebra. In the canonical basis of L this p-map is added as an attribute. A Lie algebra created by JenningsLieAlgebra is naturally graded. The attribute Grading (62.9.20) is set.

64.8.5 PCentralLieAlgebra

▷ PCentralLieAlgebra(*G*) (attribute)

Here *G* is a nontrivial p-group. PCentralLieAlgebra(*G*) does the same as JenningsLieAlgebra (64.8.4) except that the p-central series is used instead of the Jennings series (see PCentralSeries (39.17.13)). This function also returns a graded Lie algebra. However, it is not necessarily restricted.

Example

```
gap> G:= SmallGroup( 3^6, 123 );
<pc group of size 729 with 6 generators>
```

```
gap> L:= JenningsLieAlgebra( G );
<Lie algebra of dimension 6 over GF(3)>
gap> HasPthPowerImages( Basis( L ) );
true
gap> PthPowerImages( Basis( L ) );
[ v.6, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1 ]
gap> g:= Grading( L );
rec( hom_components := function( d ) ... end, max_degree := 3,
  min_degree := 1, source := Integers )
gap> List( [1,2,3], g.hom_components );
[ <vector space over GF(3), with 3 generators>,
  <vector space over GF(3), with 2 generators>,
  <vector space over GF(3), with 1 generators> ]
```

64.8.6 NaturalHomomorphismOfLieAlgebraFromNilpotentGroup

▷ NaturalHomomorphismOfLieAlgebraFromNilpotentGroup(*L*) (attribute)

This is an attribute of Lie algebras created by JenningsLieAlgebra (64.8.4) or PCentralLieAlgebra (64.8.5). Then *L* is the direct sum of quotients of successive terms of the Jennings, or *p*-central series of a *p*-group G. Let Gi be the *i*-th term in this series, and let f = NaturalHomomorphismOfLieAlgebraFromNilpotentGroup(*L*), then for g in Gi, f(*g*, *i*) returns the element of *L* (lying in the *i*-th homogeneous component) corresponding to g.

64.9 The Adjoint Representation

In this section we show functions for calculating with the adjoint representation of a Lie algebra (and the corresponding trace form, called the Killing form) (see also AdjointBasis (62.9.5) and IndicesOfAdjointBasis (62.9.6)).

64.9.1 AdjointMatrix

▷ AdjointMatrix(*B*, *x*) (operation)

is the matrix of the adjoint representation of the element *x* w.r.t. the basis *B*. The adjoint map is the left multiplication by *x*. The *i*-th column of the resulting matrix represents the image of the *i*-th basis vector of *B* under left multiplication by *x*.

Example

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> AdjointMatrix( Basis( L ), Basis( L )[1] );
[ [ 0, 0, -2 ], [ 0, 0, 0 ], [ 0, 1, 0 ] ]
```

64.9.2 AdjointAssociativeAlgebra

▷ AdjointAssociativeAlgebra(*L*, *K*) (attribute)

is the associative matrix algebra (with 1) generated by the matrices of the adjoint representation of the subalgebra *K* on the Lie algebra *L*.

Example
```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> AdjointAssociativeAlgebra( L, L );
<algebra of dimension 9 over Rationals>
gap> AdjointAssociativeAlgebra( L, CartanSubalgebra( L ) );
<algebra of dimension 3 over Rationals>
```

64.9.3 KillingMatrix

▷ `KillingMatrix(B)` (attribute)

is the matrix of the Killing form κ with respect to the basis `B`, i.e., the matrix $(\kappa(b_i,b_j))$ where $b_1,b_2,\ldots$ are the basis vectors of `B`.

Example
```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> KillingMatrix( Basis( L ) );
[ [ 0, 4, 0 ], [ 4, 0, 0 ], [ 0, 0, 8 ] ]
```

64.9.4 KappaPerp

▷ `KappaPerp(L, U)` (operation)

is the orthogonal complement of the subspace `U` of the Lie algebra `L` with respect to the Killing form κ, that is, the set $U^\perp = \{x \in L; \kappa(x,y) = 0 \text{ for all } y \in L\}$.

$U^\perp$ is a subspace of `L`, and if `U` is an ideal of `L` then $U^\perp$ is a subalgebra of `L`.

Example
```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> b:= BasisVectors( Basis( L ) );;
gap> V:= VectorSpace( Rationals, [b[1],b[2]] );;
gap> KappaPerp( L, V );
<vector space of dimension 1 over Rationals>
```

64.9.5 IsNilpotentElement

▷ `IsNilpotentElement(L, x)` (operation)

`x` is nilpotent in `L` if its adjoint matrix is a nilpotent matrix.

Example
```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> IsNilpotentElement( L, Basis( L )[1] );
true
```

64.9.6 NonNilpotentElement

▷ `NonNilpotentElement(L)` (attribute)

A non-nilpotent element of a Lie algebra `L` is an element x such that ad$x$ is not nilpotent. If `L` is not nilpotent, then by Engel's theorem non-nilpotent elements exist in `L`. In this case this function returns a non-nilpotent element of `L`, otherwise (if `L` is nilpotent) `fail` is returned.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> NonNilpotentElement( L );
v.13
gap> IsNilpotentElement( L, last );
false
```

64.9.7 FindSl2

▷ `FindSl2(`*L, x*`)` (operation)

This function tries to find a subalgebra S of the Lie algebra L with S isomorphic to sl_2 and such that the nilpotent element x of L is contained in S. If such an algebra exists then it is returned, otherwise `fail` is returned.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> b:= BasisVectors( Basis( L ) );;
gap> IsNilpotentElement( L, b[1] );
true
gap> FindSl2( L, b[1] );
<Lie algebra of dimension 3 over Rationals>
```

64.10 Universal Enveloping Algebras

64.10.1 UniversalEnvelopingAlgebra

▷ `UniversalEnvelopingAlgebra(`*L[, B]*`)` (attribute)

Returns the universal enveloping algebra of the Lie algebra `L`. The elements of this algebra are written on a Poincare-Birkhoff-Witt basis.

If a second argument `B` is given, it must be a basis of `L`, and an isomorphic copy of the universal enveloping algebra is returned, generated by the images (in the universal enveloping algebra) of the elements of `B`.

Example

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> UL:= UniversalEnvelopingAlgebra( L );
<algebra-with-one of dimension infinity over Rationals>
gap> g:= GeneratorsOfAlgebraWithOne( UL );
[ [(1)*x.1], [(1)*x.2], [(1)*x.3] ]
gap> g[3]^2*g[2]^2*g[1]^2;
[(-4)*x.1*x.2*x.3^3+(1)*x.1^2*x.2^2*x.3^2+(2)*x.3^3+(2)*x.3^4]
```

64.11 Finitely Presented Lie Algebras

Finitely presented Lie algebras can be constructed from free Lie algebras by using the `/` constructor, i.e., `FL/[r1, ..., rk]` is the quotient of the free Lie algebra `FL` by the ideal generated by the elements `r1, ..., rk` of `FL`. If the finitely presented Lie algebra `K` happens to be finite dimensional then an isomorphic structure constants Lie algebra can be constructed by `NiceAlgebraMonomorphism(K)`

(see NiceAlgebraMonomorphism (62.10.9)), which returns a surjective homomorphism. The structure constants Lie algebra can then be accessed by calling Range (32.3.7) for this map. Also limited computations with elements of the finitely presented Lie algebra are possible.

Example

```
gap> L:= FreeLieAlgebra( Rationals, "s", "t" );
<Lie algebra over Rationals, with 2 generators>
gap> gL:= GeneratorsOfAlgebra( L );; s:= gL[1];; t:= gL[2];;
gap> K:= L/[ s*(s*t), t*(t*(s*t)), s*(t*(s*t))-t*(s*t) ];
<Lie algebra over Rationals, with 2 generators>
gap> h:= NiceAlgebraMonomorphism( K );
[ [(1)*s], [(1)*t] ] -> [ v.1, v.2 ]
gap> U:= Range( h );
<Lie algebra of dimension 3 over Rationals>
gap> IsLieNilpotent( U );
true
gap> gK:= GeneratorsOfAlgebra( K );
[ [(1)*s], [(1)*t] ]
gap> gK[1]*(gK[2]*gK[1]) = Zero( K );
true
```

64.11.1 FpLieAlgebraByCartanMatrix

▷ FpLieAlgebraByCartanMatrix(*C*) (function)

Here *C* must be a Cartan matrix. The function returns the finitely-presented Lie algebra over the field of rational numbers defined by this Cartan matrix. By Serre's theorem, this Lie algebra is a semisimple Lie algebra, and its root system has Cartan matrix *C*.

Example

```
gap> C:= [ [ 2, -1 ], [ -3, 2 ] ];;
gap> K:= FpLieAlgebraByCartanMatrix( C );
<Lie algebra over Rationals, with 6 generators>
gap> h:= NiceAlgebraMonomorphism( K );
[ [(1)*x1], [(1)*x2], [(1)*x3], [(1)*x4], [(1)*x5], [(1)*x6] ] ->
[ v.1, v.2, v.3, v.4, v.5, v.6 ]
gap> SemiSimpleType( Range( h ) );
"G2"
```

64.11.2 NilpotentQuotientOfFpLieAlgebra

▷ NilpotentQuotientOfFpLieAlgebra(*FpL, max[, weights]*) (function)

Here *FpL* is a finitely presented Lie algebra. Let K be the quotient of *FpL* by the *max*+1-th term of its lower central series. This function calculates a surjective homomorphism from *FpL* onto K. When called with the third argument *weights*, the k-th generator of *FpL* gets assigned the k-th element of the list *weights*. In that case a quotient is calculated of *FpL* by the ideal generated by all elements of weight *max*+1. If the list *weights* only consists of 1's then the two calls are equivalent. The default value of *weights* is a list (of length equal to the number of generators of *FpL*) consisting of 1's.

If the relators of *FpL* are homogeneous, then the resulting algebra is naturally graded.

Example

```
gap> L:= FreeLieAlgebra( Rationals, "x", "y" );;
gap> g:= GeneratorsOfAlgebra(L);; x:= g[1]; y:= g[2];
(1)*x
(1)*y
gap> rr:=[ ((y*x)*x)*x-6*(y*x)*y,
>          3*((((y*x)*x)*x)*x)*x-20*(((y*x)*x)*x)*y ];
[ (-1)*(x*(x*(x*y)))+(6)*((x*y)*y),
  (-3)*(x*(x*(x*(x*(x*y)))))+(20)*(x*(x*((x*y)*y)))+(
    -20)*((x*(x*y))*(x*y)) ]
gap> K:= L/rr;
<Lie algebra over Rationals, with 2 generators>
gap> h:=NilpotentQuotientOfFpLieAlgebra(K, 50, [1,2] );
[ [(1)*x], [(1)*y] ] -> [ v.1, v.2 ]
gap> L:= Range( h );
<Lie algebra of dimension 50 over Rationals>
gap> Grading( L );
rec( hom_components := function( d ) ... end, max_degree := 50,
  min_degree := 1, source := Integers )
```

64.12 Modules over Lie Algebras and Their Cohomology

Representations of Lie algebras are dealt with in the same way as representations of ordinary algebras (see 62.11). In this section we mainly deal with modules over general Lie algebras and their cohomology. The next section is devoted to modules over semisimple Lie algebras. An s-cochain of a module V over a Lie algebra L is an s-linear map

$$c : L \times \cdots \times L \to V,$$

with s factors L, that is skew-symmetric (meaning that if any of the arguments are interchanged, c changes to $-c$).

Let $(x_1,\ldots,x_n)$ be a basis of L. Then any s-cochain is determined by the values $c(x_{i_1},\ldots,x_{i_s})$, where $1 \le i_1 < i_2 < \cdots < i_s \le \dim L$. Now this value again is a linear combination of basis elements of V: $c(x_{i_1},\ldots,x_{i_s}) = \sum \lambda^k_{i_1,\ldots,i_s} v_k$. Denote the dimension of V by r. Then we represent an s-cocycle by a list of r lists. The j-th of those lists consists of entries of the form

$$[[i_1,i_2,\ldots,i_s],\lambda^j_{i_1,\ldots,i_s}]$$

where the coefficient on the second position is non-zero. (We only store those entries for which this coefficient is non-zero.) It follows that every s-tuple $(i_1,\ldots,i_s)$ gives rise to r basis elements.

So the zero cochain is represented by a list of the form `[ [ ], [ ], \ldots, [ ] ]`. Furthermore, if V is, e.g., 4-dimensional, then the 2-cochain represented by

Example

```
[ [ [ [1,2], 2] ], [ ], [ [ [1,2], 1/2 ] ], [ ] ]
```

maps the pair (x_1,x_2) to $2v_1 + 1/2v_3$ (where v_1 is the first basis element of V, and v_3 the third), and all other pairs to zero.

By definition, 0-cochains are constant maps $c(x) = v_c \in V$ for all $x \in L$. So 0-cochains have a different representation: they are just represented by the list `[ v_c ]`.

Cochains are constructed using the function `Cochain` (64.12.2), if `c` is a cochain, then its corresponding list is returned by `ExtRepOfObj( c )`.

64.12.1 IsCochain

▷ IsCochain(*obj*) (Category)
▷ IsCochainCollection(*obj*) (Category)

Categories of cochains and of collections of cochains.

64.12.2 Cochain

▷ Cochain(*V*, *s*, *obj*) (operation)

Constructs a *s*-cochain given by the data in *obj*, with respect to the Lie algebra module *V*. If *s* is non-zero, then *obj* must be a list.

Example

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );
<3-dimensional left-module over <Lie algebra of dimension
3 over Rationals>>
gap> c1:= Cochain( V, 2,
>               [ [ [ [ 1, 3 ], -1 ] ], [ ], [ [ [ 2, 3 ], 1/2 ] ] ]);
<2-cochain>
gap> ExtRepOfObj( c1 );
[ [ [ [ 1, 3 ], -1 ] ], [  ], [ [ [ 2, 3 ], 1/2 ] ] ]
gap> c2:= Cochain( V, 0, Basis( V )[1] );
<0-cochain>
gap> ExtRepOfObj( c2 );
v.1
gap> IsCochain( c2 );
true
```

64.12.3 CochainSpace

▷ CochainSpace(*V*, *s*) (operation)

Returns the space of all *s*-cochains with respect to *V*.

Example

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );;
gap> C:=CochainSpace( V, 2 );
<vector space of dimension 9 over Rationals>
gap> BasisVectors( Basis( C ) );
[ <2-cochain>, <2-cochain>, <2-cochain>, <2-cochain>, <2-cochain>,
  <2-cochain>, <2-cochain>, <2-cochain>, <2-cochain> ]
gap> ExtRepOfObj( last[1] );
[ [ [ [ 1, 2 ], 1 ] ], [  ], [  ] ]
```

64.12.4 ValueCochain

▷ ValueCochain(*c*, *y1*, *y2*, ..., *ys*) (function)

Here *c* is an s-cochain. This function returns the value of *c* when applied to the s elements *y1* to *ys* (that lie in the Lie algebra acting on the module corresponding to *c*). It is also possible to call this function with two arguments: first *c* and then the list containing *y1*,...,*ys*.

Example

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );;
gap> C:= CochainSpace( V, 2 );;
gap> c:= Basis( C )[1];
<2-cochain>
gap>  ValueCochain( c, Basis(L)[2], Basis(L)[1] );
(-1)*v.1
```

64.12.5 LieCoboundaryOperator

▷ LieCoboundaryOperator(*c*) (function)

This is a function that takes an s-cochain *c*, and returns an s+1-cochain. The coboundary operator is applied.

Example

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );;
gap> C:= CochainSpace( V, 2 );;
gap> c:= Basis( C )[1];;
gap> c1:= LieCoboundaryOperator( c );
<3-cochain>
gap> c2:= LieCoboundaryOperator( c1 );
<4-cochain>
```

64.12.6 Cocycles (for Lie algebra module)

▷ Cocycles(*V*, *s*) (operation)

is the space of all *s*-cocycles with respect to the Lie algebra module *V*. That is the kernel of the coboundary operator when restricted to the space of *s*-cochains.

64.12.7 Coboundaries

▷ Coboundaries(*V*, *s*) (operation)

is the space of all *s*-coboundaries with respect to the Lie algebra module *V*. That is the image of the coboundary operator, when applied to the space of *s*-1-cochains. By definition the space of all 0-coboundaries is zero.

Example

```
gap> T:= EmptySCTable( 3, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 3 ] );
gap> L:= LieAlgebraByStructureConstants( Rationals, T );;
gap> V:= FaithfulModule( L );
<left-module over <Lie algebra of dimension 3 over Rationals>>
gap> Cocycles( V, 2 );
<vector space of dimension 7 over Rationals>
```

```
gap> Coboundaries( V, 2 );
<vector space over Rationals, with 9 generators>
gap> Dimension( last );
5
```

64.13 Modules over Semisimple Lie Algebras

This section contains functions for calculating information on representations of semisimple Lie algebras. First we have some functions for calculating some combinatorial data (set of dominant weights, the dominant character, the decomposition of a tensor product, the dimension of a highest-weight module). Then there is a function for creating an admissible lattice in the universal enveloping algebra of a semisimple Lie algebra. Finally we have a function for constructing a highest-weight module over a semisimple Lie algebra.

64.13.1 DominantWeights

▷ `DominantWeights(`*R, maxw*`)` (operation)

Returns a list consisting of two lists. The first of these contains the dominant weights (written on the basis of fundamental weights) of the irreducible highest-weight module, with highest weight *maxw*, over the Lie algebra with the root system *R*. The i-th element of the second list is the level of the i-th dominant weight. (Where the level is defined as follows. For a weight μ we write $\mu = \lambda - \sum_i k_i \alpha_i$, where the α_i are the simple roots, and λ the highest weight. Then the level of μ is $\sum_i k_i$.)

64.13.2 DominantCharacter (for a semisimple Lie algebra and a highest weight)

▷ `DominantCharacter(`*L, maxw*`)` (operation)
▷ `DominantCharacter(`*R, maxw*`)` (operation)

For a highest weight *maxw* and a semisimple Lie algebra *L*, this returns the dominant weights of the highest-weight module over *L*, with highest weight *maxw*. The output is a list of two lists, the first list contains the dominant weights; the second list contains their multiplicities.

The first argument can also be a root system, in which case the dominant character of the highest-weight module over the corresponding semisimple Lie algebra is returned.

64.13.3 DecomposeTensorProduct

▷ `DecomposeTensorProduct(`*L, w1, w2*`)` (operation)

Here *L* is a semisimple Lie algebra and *w1*, *w2* are dominant weights. Let V_i be the irreducible highest-weight module over *L* with highest weight w_i for $i = 1, 2$. Let $W = V_1 \otimes V_2$. Then in general W is a reducible *L*-module. Now this function returns a list of two lists. The first of these is the list of highest weights of the irreducible modules occurring in the decomposition of W as a direct sum of irreducible modules. The second list contains the multiplicities of these weights (i.e., the number of copies of the irreducible module with the corresponding highest weight that occur in W). The algorithm uses Klimyk's formula (see [Kli68] or [Kli66] for the original Russian version).

64.13.4 DimensionOfHighestWeightModule

▷ `DimensionOfHighestWeightModule(L, w)` (operation)

Here `L` is a semisimple Lie algebra, and `w` a dominant weight. This function returns the dimension of the highest-weight module over `L` with highest weight `w`. The algorithm uses Weyl's dimension formula.

Example

```
gap> L:= SimpleLieAlgebra( "F", 4, Rationals );;
gap> R:= RootSystem( L );;
gap> DominantWeights( R, [ 1, 1, 0, 0 ] );
[ [ [ 1, 1, 0, 0 ], [ 2, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
      [ 1, 0, 0, 0 ], [ 0, 0, 0, 0 ] ], [ 0, 3, 4, 8, 11, 19 ] ]
gap> DominantCharacter( L, [ 1, 1, 0, 0 ] );
[ [ [ 1, 1, 0, 0 ], [ 2, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
      [ 1, 0, 0, 0 ], [ 0, 0, 0, 0 ] ], [ 1, 1, 4, 6, 14, 21 ] ]
gap> DecomposeTensorProduct( L, [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ] );
[ [ [ 1, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ], [ 0, 1, 0, 0 ],
      [ 2, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 1, 0, 0 ] ],
  [ 1, 1, 1, 1, 1, 1, 1 ] ]
gap> DimensionOfHighestWeightModule( L, [ 1, 2, 3, 4 ] );
79316832731136
```

64.14 Admissible Lattices in UEA

Let L be a semisimple Lie algebra over a field of characteristic 0, and let R be its root system. For a positive root α we let x_α and y_α be positive and negative root vectors, respectively, both from a fixed Chevalley basis of L. Furthermore, $h_1,\ldots,h_l$ are the Cartan elements from the same Chevalley basis. Also we set

$$x_\alpha^{(n)} = \frac{x_\alpha^n}{n!}, y_\alpha^{(n)} = \frac{y_\alpha^n}{n!}.$$

Furthermore, let $\alpha_1,\ldots,\alpha_s$ denote the positive roots of R. For multi-indices $N=(n_1,\ldots,n_s)$, $M=(m_1,\ldots,m_s)$ and $K=(k_1,\ldots,k_s)$ (where $n_i,m_i,k_i \geq 0$) set

$$\begin{aligned} x^N &= x_{\alpha_1}^{(n_1)} \cdots x_{\alpha_s}^{(n_s)}, \\ y^M &= y_{\alpha_1}^{(m_1)} \cdots y_{\alpha_s}^{(m_s)}, \\ h^K &= \binom{h_1}{k_1} \cdots \binom{h_l}{k_l} \end{aligned}$$

Then by a theorem of Kostant, the $x_\alpha^{(n)}$ and $y_\alpha^{(n)}$ generate a subring of the universal enveloping algebra $U(L)$ spanned (as a free Z-module) by the elements

$$y^M h^K x^N$$

(see, e.g., [Hum72] or [Hum78, Section 26]) So by the Poincare-Birkhoff-Witt theorem this subring is a lattice in $U(L)$. Furthermore, this lattice is invariant under the $x_\alpha^{(n)}$ and $y_\alpha^{(n)}$. Therefore, it is called an admissible lattice in $U(L)$.

The next functions enable us to construct the generators of such an admissible lattice.

64.14.1 IsUEALatticeElement

▷ IsUEALatticeElement(*obj*) (Category)
▷ IsUEALatticeElementCollection(*obj*) (Category)
▷ IsUEALatticeElementFamily(*fam*) (Category)

is the category of elements of an admissible lattice in the universal enveloping algebra of a semisimple Lie algebra L.

64.14.2 LatticeGeneratorsInUEA

▷ LatticeGeneratorsInUEA(*L*) (attribute)

Here *L* must be a semisimple Lie algebra of characteristic 0. This function returns a list of generators of an admissible lattice in the universal enveloping algebra of *L*, relative to the Chevalley basis contained in ChevalleyBasis(*L*) (see ChevalleyBasis (64.6.2)). First are listed the negative root vectors (denoted by $y_1,\ldots,y_s$), then the positive root vectors (denoted by $x_1,\ldots,x_s$). At the end of the list there are the Cartan elements. They are printed as (hi/1), which means

$$\binom{h_i}{1}.$$

In general the printed form (hi/ k) means

$$\binom{h_i}{k}.$$

Also $y_i^{(m)}$ is printed as yi^(m), which means that entering yi^m at the GAP prompt results in the output m!*yi^(m).

Products of lattice generators are collected using the following order: first come the $y_i^{(m_i)}$ (in the same order as the positive roots), then the $\binom{h_i}{k_i}$, and then the $x_i^{(n_i)}$ (in the same order as the positive roots).

64.14.3 ObjByExtRep (for creating a UEALattice element)

▷ ObjByExtRep(*F, descr*) (function)

An UEALattice element is represented by a list of the form [m1, c1, m2, c2, ...], where the c1, c2 etc. are coefficients, and the m1, m2 etc. are monomials. A monomial is a list of the form [ind1, e1, ind2, e2, ...] where ind1, ind2 are indices, and e1, e2 etc. are exponents. Let N be the number of positive roots of the underlying Lie algebra L. The indices lie between 1 and $dim(L)$. If an index lies between 1 and N, then it represents a negative root vector (corresponding to the root NegativeRoots(R)[ind], where R is the root system of L; see NegativeRoots (64.6.8)). This leads to a factor yind1^(e1) in the printed form of the monomial (which equals z^e1/e1!, where z is a basis element of L). If an index lies between $N+1$ and $2N$, then it represents a positive root vector. Finally, if ind lies between $2N+1$ and $2N+rank$, then it represents an element of the Cartan subalgebra. This is printed as (h_1/e_1), meaning $\binom{h_1}{e_1}$, where $h_1,\ldots,h_{rank}$ are the canonical Cartan generators.

The zero element is represented by the empty list, the identity element by the list [[], 1].

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> g:=LatticeGeneratorsInUEA( L );
[ y1, y2, y3, y4, y5, y6, x1, x2, x3, x4, x5, x6, ( h13/1 ),
  ( h14/1 ) ]
gap> IsUEALatticeElement( g[1] );
true
gap> g[1]^3;
6*y1^(3)
gap> q:= g[7]*g[1]^2;
-2*y1+2*y1*( h13/1 )+2*y1^(2)*x1
gap> ExtRepOfObj( q );
[ [ 1, 1 ], -2, [ 1, 1, 13, 1 ], 2, [ 1, 2, 7, 1 ], 2 ]
```

64.14.4 IsWeightRepElement

▷ IsWeightRepElement(*obj*) (Category)
▷ IsWeightRepElementCollection(*obj*) (Category)
▷ IsWeightRepElementFamily(*fam*) (Category)

Is a category of vectors, that is used to construct elements of highest-weight modules (by HighestWeightModule (64.14.5)).

WeightRepElements are represented by a list of the form [v1, c1, v2, c2, ...], where the vi are basis vectors, and the ci are coefficients. Furthermore a basis vector v is a weight vector. It is represented by a list of the form [k, mon, wt], where k is an integer (the basis vectors are numbered from 1 to $\dim V$, where V is the highest weight module), mon is an UEALatticeElement (which means that the result of applying mon to a highest weight vector is v; see IsUEALatticeElement (64.14.1)) and wt is the weight of v. A WeightRepElement is printed as mon*v0, where v0 denotes a fixed highest weight vector.

If v is a WeightRepElement, then ExtRepOfObj(v) returns the corresponding list, and if list is such a list and *fam* a WeightRepElementFamily, then ObjByExtRep(*list*, *fam*) returns the corresponding WeightRepElement.

64.14.5 HighestWeightModule

▷ HighestWeightModule(*L*, *wt*) (function)

returns the highest weight module with highest weight *wt* of the semisimple Lie algebra *L* of characteristic 0.

Note that the elements of such a module lie in the category IsLeftAlgebraModuleElement (62.11.9) (and in particular they do not lie in the category IsWeightRepElement (64.14.4)). However, if v is an element of such a module, then ExtRepOfObj(v) is a WeightRepElement.

Note that for the following examples of this chapter we increase the line length limit from its default value 80 to 81 in order to make some long output expressions fit into the lines.

Example

```
gap> K1:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> K2:= SimpleLieAlgebra( "B", 2, Rationals );;
gap> L:= DirectSumOfAlgebras( K1, K2 );
<Lie algebra of dimension 24 over Rationals>
```

```
gap> V:= HighestWeightModule( L, [ 0, 1, 1, 1 ] );
<224-dimensional left-module over <Lie algebra of dimension
24 over Rationals>>
gap> vv:= GeneratorsOfLeftModule( V );;
gap> vv[100];
y5*y7*y10*v0
gap> e:= ExtRepOfObj( vv[100] );
y5*y7*y10*v0
gap> ExtRepOfObj( e );
[ [ 100, y5*y7*y10, [ -3, 2, -1, 1 ] ], 1 ]
gap> Basis(L)[17]^vv[100];
-1*y5*y7*y8*v0-1*y5*y9*v0
```

64.15 Tensor Products and Exterior and Symmetric Powers

64.15.1 TensorProductOfAlgebraModules (for a list of algebra modules)

▷ `TensorProductOfAlgebraModules(`*list*`)` (operation)
▷ `TensorProductOfAlgebraModules(`*V, W*`)` (operation)

Here the elements of *list* must be algebra modules. The tensor product is returned as an algebra module. The two-argument version works in the same way and returns the tensor product of its arguments.

Example

```
gap> L:= SimpleLieAlgebra("G",2,Rationals);;
gap> V:= HighestWeightModule( L, [ 1, 0 ] );;
gap> W:= TensorProductOfAlgebraModules( [ V, V, V ] );
<343-dimensional left-module over <Lie algebra of dimension
14 over Rationals>>
gap> w:= Basis(W)[1];
1*(1*v0<x>1*v0<x>1*v0)
gap> Basis(L)[1]^w;
<0-tensor>
gap> Basis(L)[7]^w;
1*(1*v0<x>1*v0<x>y1*v0)+1*(1*v0<x>y1*v0<x>1*v0)+1*(y
1*v0<x>1*v0<x>1*v0)
```

64.15.2 ExteriorPowerOfAlgebraModule

▷ `ExteriorPowerOfAlgebraModule(`*V, k*`)` (operation)

Here *V* must be an algebra module, defined over a Lie algebra. This function returns the *k*-th exterior power of *V* as an algebra module.

Example

```
gap> L:= SimpleLieAlgebra("G",2,Rationals);;
gap> V:= HighestWeightModule( L, [ 1, 0 ] );;
gap> W:= ExteriorPowerOfAlgebraModule( V, 3 );
<35-dimensional left-module over <Lie algebra of dimension
14 over Rationals>>
gap> w:= Basis(W)[1];
```

```
1*(1*v0/\y1*v0/\y3*v0)
gap> Basis(L)[10]^w;
1*(1*v0/\y1*v0/\y6*v0)+1*(1*v0/\y3*v0/\y5*v0)+1*(y1*v0/\y3*v0/\y4*v0)
```

64.15.3 SymmetricPowerOfAlgebraModule

▷ SymmetricPowerOfAlgebraModule(*V*, *k*) (operation)

Here *V* must be an algebra module. This function returns the *k*-th symmetric power of *V* (as an algebra module).

Example

```
gap> L:= SimpleLieAlgebra("G",2,Rationals);;
gap> V:= HighestWeightModule( L, [ 1, 0 ] );;
gap> W:= SymmetricPowerOfAlgebraModule( V, 3 );
<84-dimensional left-module over <Lie algebra of dimension
14 over Rationals>>
gap> w:= Basis(W)[1];
1*(1*v0.1*v0.1*v0)
gap> Basis(L)[2]^w;
<0-symmetric element>
gap> Basis(L)[7]^w;
3*(1*v0.1*v0.y1*v0)
```

Chapter 65

Magma Rings

Given a magma M then the *free magma ring* (or *magma ring* for short) RM of M over a ring-with-one R is the set of finite sums $\sum_{i \in I} r_i m_i$ with $r_i \in R$, and $m_i \in M$. With the obvious addition and R-action from the left, RM is a free R-module with R-basis M, and with the usual convolution product, RM is a ring.

Typical examples of free magma rings are

- (multivariate) polynomial rings (see 66.15), where the magma is a free abelian monoid generated by the indeterminates,
- group rings (see `IsGroupRing` (65.1.5)), where the magma is a group,
- Laurent polynomial rings, which are group rings of the free abelian groups generated by the indeterminates,
- free algebras and free associative algebras, with or without one, where the magma is a free magma or a free semigroup, or a free magma-with-one or a free monoid, respectively.

Note that formally, polynomial rings in GAP are not constructed as free magma rings.

Furthermore, a free Lie algebra is *not* a magma ring, because of the additional relations given by the Jacobi identity; see 65.4 for a generalization of magma rings that covers such structures.

The coefficient ring R and the magma M cannot be regarded as subsets of RM, hence the natural *embeddings* of R and M into RM must be handled via explicit embedding maps (see 65.3). Note that in a magma ring, the addition of elements is in general different from an addition that may be defined already for the elements of the magma; for example, the addition in the group ring of a matrix group does in general *not* coincide with the addition of matrices.

Example

```
gap> a:= Algebra( GF(2), [ [ [ Z(2) ] ] ] );;  Size( a );
2
gap> rm:= FreeMagmaRing( GF(2), a );;
gap> emb:= Embedding( a, rm );;
gap> z:= Zero( a );;  o:= One( a );;
gap> imz:= z ^ emb;  IsZero( imz );
(Z(2)^0)*[ [ 0*Z(2) ] ]
false
gap> im1:= ( z + o ) ^ emb;
(Z(2)^0)*[ [ Z(2)^0 ] ]
```

```
gap> im2:= z ^ emb + o ^ emb;
(Z(2)^0)*[ [ 0*Z(2) ] ]+(Z(2)^0)*[ [ Z(2)^0 ] ]
gap> im1 = im2;
false
```

65.1 Free Magma Rings

65.1.1 FreeMagmaRing

▷ `FreeMagmaRing(`*R*`, `*M*`)` (function)

is a free magma ring over the ring *R*, free on the magma *M*.

65.1.2 GroupRing

▷ `GroupRing(`*R*`, `*G*`)` (function)

is the group ring of the group *G*, over the ring *R*.

65.1.3 IsFreeMagmaRing

▷ `IsFreeMagmaRing(`*D*`)` (Category)

A domain lies in the category `IsFreeMagmaRing` if it has been constructed as a free magma ring. In particular, if *D* lies in this category then the operations `LeftActingDomain` (57.1.11) and `UnderlyingMagma` (65.1.6) are applicable to *D*, and yield the ring *R* and the magma *M* such that *D* is the magma ring *RM*.

So being a magma ring in GAP includes the knowledge of the ring and the magma. Note that a magma ring *RM* may abstractly be generated as a magma ring by a magma different from the underlying magma *M*. For example, the group ring of the dihedral group of order 8 over the field with 3 elements is also spanned by a quaternion group of order 8 over the same field.

Example

```
gap> d8:= DihedralGroup( 8 );
<pc group of size 8 with 3 generators>
gap> rm:= FreeMagmaRing( GF(3), d8 );
<algebra-with-one over GF(3), with 3 generators>
gap> emb:= Embedding( d8, rm );;
gap> gens:= List( GeneratorsOfGroup( d8 ), x -> x^emb );;
gap> x1:= gens[1] + gens[2];;
gap> x2:= ( gens[1] - gens[2] ) * gens[3];;
gap> x3:= gens[1] * gens[2] * ( One( rm ) - gens[3] );;
gap> g1:= x1 - x2 + x3;;
gap> g2:= x1 + x2;;
gap> q8:= Group( g1, g2 );;
gap> Size( q8 );
8
gap> ForAny( [ d8, q8 ], IsAbelian );
false
gap> List( [ d8, q8 ], g -> Number( AsList( g ), x -> Order( x ) = 2 ) );
```

```
[ 5, 1 ]
gap> Dimension( Subspace( rm, q8 ) );
8
```

65.1.4 IsFreeMagmaRingWithOne

▷ IsFreeMagmaRingWithOne(*obj*) (Category)

IsFreeMagmaRingWithOne is just a synonym for the meet of IsFreeMagmaRing (65.1.3) and IsMagmaWithOne (35.1.2).

65.1.5 IsGroupRing

▷ IsGroupRing(*obj*) (property)

A *group ring* is a magma ring where the underlying magma is a group.

65.1.6 UnderlyingMagma

▷ UnderlyingMagma(*RM*) (attribute)

stores the underlying magma of a free magma ring.

65.1.7 AugmentationIdeal

▷ AugmentationIdeal(*RG*) (attribute)

is the augmentation ideal of the group ring *RG*, i.e., the kernel of the trivial representation of *RG*.

65.2 Elements of Free Magma Rings

In order to treat elements of free magma rings uniformly, also without an external representation, the attributes CoefficientsAndMagmaElements (65.2.4) and ZeroCoefficient (65.2.5) were introduced that allow one to "take an element of an arbitrary magma ring into pieces".

Conversely, for constructing magma ring elements from coefficients and magma elements, ElementOfMagmaRing (65.2.6) can be used. (Of course one can also embed each magma element into the magma ring, see 65.3, and then form the linear combination, but many unnecessary intermediate elements are created this way.)

65.2.1 IsMagmaRingObjDefaultRep

▷ IsMagmaRingObjDefaultRep(*obj*) (Representation)

The default representation of a magma ring element is a list of length 2, at first position the zero coefficient, at second position a list with the coefficients at the even positions, and the magma elements at the odd positions, with the ordering as defined for the magma elements.

It is assumed that arithmetic operations on magma rings produce only normalized elements.

65.2.2 IsElementOfFreeMagmaRing

▷ IsElementOfFreeMagmaRing(*obj*) (Category)
▷ IsElementOfFreeMagmaRingCollection(*obj*) (Category)

The category of elements of a free magma ring (See IsFreeMagmaRing (65.1.3)).

65.2.3 IsElementOfFreeMagmaRingFamily

▷ IsElementOfFreeMagmaRingFamily(*Fam*) (Category)

Elements of families in this category have trivial normalisation, i.e., efficient methods for \= and \<.

65.2.4 CoefficientsAndMagmaElements

▷ CoefficientsAndMagmaElements(*elm*) (attribute)

is a list that contains at the odd positions the magma elements, and at the even positions their coefficients in the element *elm*.

65.2.5 ZeroCoefficient

▷ ZeroCoefficient(*elm*) (attribute)

For an element *elm* of a magma ring (modulo relations) *RM*, ZeroCoefficient returns the zero element of the coefficient ring *R*.

65.2.6 ElementOfMagmaRing

▷ ElementOfMagmaRing(*Fam, zerocoeff, coeffs, mgmelms*) (operation)

ElementOfMagmaRing returns the element $\sum_{i=1}^{n} c_i m_i'$, where *coeffs* $= [c_1, c_2, \ldots, c_n]$ is a list of coefficients, *mgmelms* $= [m_1, m_2, \ldots, m_n]$ is a list of magma elements, and m_i' is the image of m_i under an embedding of a magma containing m_i into a magma ring whose elements lie in the family *Fam*. *zerocoeff* must be the zero of the coefficient ring containing the c_i.

65.3 Natural Embeddings related to Magma Rings

Neither the coefficient ring *R* nor the magma *M* are regarded as subsets of the magma ring *RM*, so one has to use *embeddings* (see Embedding (32.2.10)) explicitly whenever one needs for example the magma ring element corresponding to a given magma element.

Example

```
gap> f:= Rationals;;  g:= SymmetricGroup( 3 );;
gap> fg:= FreeMagmaRing( f, g );
<algebra-with-one over Rationals, with 2 generators>
gap> Dimension( fg );
6
```

```
gap> gens:= GeneratorsOfAlgebraWithOne( fg );
[ (1)*(1,2,3), (1)*(1,2) ]
gap> ( 3*gens[1] - 2*gens[2] ) * ( gens[1] + gens[2] );
(-2)*()+(3)*(2,3)+(3)*(1,3,2)+(-2)*(1,3)
gap> One( fg );
(1)*()
gap> emb:= Embedding( g, fg );;
gap> elm:= (1,2,3)^emb;  elm in fg;
(1)*(1,2,3)
true
gap> new:= elm + One( fg );
(1)*()+(1)*(1,2,3)
gap> new^2;
(1)*()+(2)*(1,2,3)+(1)*(1,3,2)
gap> emb2:= Embedding( f, fg );;
gap> elm:= One( f )^emb2;  elm in fg;
(1)*()
true
```

65.4 Magma Rings modulo Relations

A more general construction than that of free magma rings allows one to create rings that are not free R-modules on a given magma M but arise from the magma ring RM by factoring out certain identities. Examples for such structures are finitely presented (associative) algebras and free Lie algebras (see `FreeLieAlgebra` (64.2.4)).

In GAP, the use of magma rings modulo relations is limited to situations where a normal form of the elements is known and where one wants to guarantee that all elements actually constructed are in normal form. (In particular, the computation of the normal form must be cheap.) This is because the methods for comparing elements in magma rings modulo relations via \= and \< just compare the involved coefficients and magma elements, and also the vector space functions regard those monomials as linearly independent over the coefficients ring that actually occur in the representation of an element of a magma ring modulo relations.

Thus only very special finitely presented algebras will be represented as magma rings modulo relations, in general finitely presented algebras are dealt with via the mechanism described in Chapter 63.

65.4.1 IsElementOfMagmaRingModuloRelations

▷ `IsElementOfMagmaRingModuloRelations(`*obj*`)` (Category)
▷ `IsElementOfMagmaRingModuloRelationsCollection(`*obj*`)` (Category)

This category is used, e. g., for elements of free Lie algebras.

65.4.2 IsElementOfMagmaRingModuloRelationsFamily

▷ `IsElementOfMagmaRingModuloRelationsFamily(`*Fam*`)` (Category)

The family category for the category `IsElementOfMagmaRingModuloRelations` (65.4.1).

65.4.3 NormalizedElementOfMagmaRingModuloRelations

▷ NormalizedElementOfMagmaRingModuloRelations(*F, descr*) (operation)

Let *F* be a family of magma ring elements modulo relations, and *descr* the description of an element in a magma ring modulo relations. NormalizedElementOfMagmaRingModuloRelations returns a description of the same element, but normalized w.r.t. the relations. So two elements are equal if and only if the result of NormalizedElementOfMagmaRingModuloRelations is equal for their internal data, that is, CoefficientsAndMagmaElements (65.2.4) will return the same for the corresponding two elements.

NormalizedElementOfMagmaRingModuloRelations is allowed to return *descr* itself, it need not make a copy. This is the case for example in the case of free magma rings.

65.4.4 IsMagmaRingModuloRelations

▷ IsMagmaRingModuloRelations(*obj*) (Category)

A GAP object lies in the category IsMagmaRingModuloRelations if it has been constructed as a magma ring modulo relations. Each element of such a ring has a unique normal form, so CoefficientsAndMagmaElements (65.2.4) is well-defined for it.

This category is not inherited to factor structures, which are in general best described as finitely presented algebras, see Chapter 63.

65.5 Magma Rings modulo the Span of a Zero Element

65.5.1 IsElementOfMagmaRingModuloSpanOfZeroFamily

▷ IsElementOfMagmaRingModuloSpanOfZeroFamily(*Fam*) (Category)

We need this for the normalization method, which takes a family as first argument.

65.5.2 IsMagmaRingModuloSpanOfZero

▷ IsMagmaRingModuloSpanOfZero(*RM*) (Category)

The category of magma rings modulo the span of a zero element.

65.5.3 MagmaRingModuloSpanOfZero

▷ MagmaRingModuloSpanOfZero(*R, M, z*) (function)

Let *R* be a ring, *M* a magma, and *z* an element of *M* with the property that $z * m = z$ holds for all $m \in M$. The element *z* could be called a "zero element" of *M*, but note that in general *z* cannot be obtained as Zero(*m*) for each $m \in M$, so this situation does not match the definition of Zero (31.10.3).

MagmaRingModuloSpanOfZero returns the magma ring *RM* modulo the relation given by the identification of *z* with zero. This is an example of a magma ring modulo relations, see 65.4.

65.6 Technical Details about the Implementation of Magma Rings

The *family* containing elements in the magma ring *RM* in fact contains all elements with coefficients in the family of elements of *R* and magma elements in the family of elements of *M*. So arithmetic operations with coefficients outside *R* or with magma elements outside *M* might create elements outside *RM*.

It should be mentioned that each call of `FreeMagmaRing` (65.1.1) creates a new family of elements, so for example the elements of two group rings of permutation groups over the same ring lie in different families and therefore are regarded as different.

Example

```
gap> g:= SymmetricGroup( 3 );;
gap> h:= AlternatingGroup( 3 );;
gap> IsSubset( g, h );
true
gap> f:= GF(2);;
gap> fg:= GroupRing( f, g );
<algebra-with-one over GF(2), with 2 generators>
gap> fh:= GroupRing( f, h );
<algebra-with-one over GF(2), with 1 generators>
gap> IsSubset( fg, fh );
false
gap> o1:= One( fh );  o2:= One( fg );  o1 = o2;
(Z(2)^0)*()
(Z(2)^0)*()
false
gap> emb:= Embedding( g, fg );;
gap> im:= Image( emb, h );
<group of size 3 with 1 generators>
gap> IsSubset( fg, im );
true
```

There is *no* generic *external representation* for elements in an arbitrary free magma ring. For example, polynomials are elements of a free magma ring, and they have an external representation relying on the special form of the underlying monomials. On the other hand, elements in a group ring of a permutation group do not admit such an external representation.

For convenience, magma rings constructed with `FreeAlgebra` (62.3.1), `FreeAssociativeAlgebra` (62.3.3), `FreeAlgebraWithOne` (62.3.2), and `FreeAssociativeAlgebraWithOne` (62.3.4) support an external representation of their elements, which is defined as a list of length 2, the first entry being the zero coefficient, the second being a list with the external representations of the magma elements at the odd positions and the corresponding coefficients at the even positions.

As the above examples show, there are several possible representations of magma ring elements, the representations used for polynomials (see Chapter 66) as well as the default representation `IsMagmaRingObjDefaultRep` (65.2.1) of magma ring elements. The latter simply stores the zero coefficient and a list containing the coefficients of the element at the even positions and the corresponding magma elements at the odd positions, where the succession is compatible with the ordering of magma elements via `\<`.

Chapter 66

Polynomials and Rational Functions

Let R be a commutative ring-with-one. We call a free associative algebra A over R a *polynomial ring* over R. The free generators of A are called *indeterminates* (to avoid naming conflicts with the word *variables* which will be used to denote GAP variables only) , they are usually denoted by $x_1, x_2, \ldots$. The number of indeterminates is called the *rank* of A. The elements of A are called *polynomials*. Products of indeterminates are called *monomials*, every polynomial can be expressed as a finite sum of products of monomials with ring elements in a form like $r_{1,0}x_1 + r_{1,1}x_1x_2 + r_{0,1}x_2 + \cdots$ with $r_{i,j} \in R$.

A polynomial ring of rank 1 is called an *univariate* polynomial ring, its elements are *univariate polynomials*.

Polynomial rings of smaller rank naturally embed in rings of higher rank; if S is a subring of R then a polynomial ring over S naturally embeds in a polynomial ring over R of the same rank. Note however that GAP does not consider R as a subset of a polynomial ring over R; for example the zero of R (0) and the zero of the polynomial ring ($0x^0$) are different objects.

Internally, indeterminates are represented by positive integers, but it is possible to give names to them to have them printed in a nicer way. Beware, however that there is not necessarily any relation between the way an indeterminate is called and the way it is printed. See section 66.1 for details.

If R is an integral domain, the polynomial ring A over R is an integral domain as well and one can therefore form its quotient field Q. This field is called a *field of rational functions*. Again A embeds naturally into Q and GAP will perform this embedding implicitly. (In fact it implements the ring of rational functions over R.) To avoid problems with leading coefficients, however, R must be a unique factorization domain.

66.1 Indeterminates

Internally, indeterminates are created for a *family* of objects (for example all elements of finite fields in characteristic 3 are in one family). Thus a variable "x" over the rationals is also an "x" over the integers, while an "x" over `GF(3)` is different.

Within one family, every indeterminate has a number *nr* and as long as no other names have been assigned, this indeterminate will be displayed as "`x_`*nr*". Indeterminate numbers can be arbitrary nonnegative integers.

It is possible to assign names to indeterminates; these names are strings and only provide a means for printing the indeterminates in a nice way. Indeterminates that have not been assigned a name will be printed as "`x_`*nr*".

(Because of this printing convention, the name x_*nr* is interpreted specially to always denote the variable with internal number *nr*.)

The indeterminate names have not necessarily any relations to variable names: this means that an indeterminate whose name is, say, "x" cannot be accessed using the variable x, unless x was defined to be that indeterminate.

When asking for indeterminates with certain names, GAP usually will take the first (with respect to the internal numbering) indeterminates that are not yet named, name these accordingly and return them. Thus when asking for named indeterminates, no relation between names and indeterminate numbers can be guaranteed. The attribute IndeterminateNumberOfLaurentPolynomial(*indet*) will return the number of the indeterminate *indet*.

When asked to create an indeterminate with a name that exists already for the family, GAP will by default return this existing indeterminate. If you explicitly want a *new* indeterminate, distinct from the already existing one with the *same* name, you can add the new option to the function call. (This is in most cases not a good idea.)

Example

```
gap> R:=PolynomialRing(GF(3),["x","y","z"]);
GF(3)[x,y,z]
gap> List(IndeterminatesOfPolynomialRing(R),
>   IndeterminateNumberOfLaurentPolynomial);
[ 1, 2, 3 ]
gap> R:=PolynomialRing(GF(3),["z"]);
GF(3)[z]
gap> List(IndeterminatesOfPolynomialRing(R),
>   IndeterminateNumberOfLaurentPolynomial);
[ 3 ]
gap> R:=PolynomialRing(GF(3),["x","y","z"]:new);
GF(3)[x,y,z]
gap> List(IndeterminatesOfPolynomialRing(R),
>   IndeterminateNumberOfLaurentPolynomial);
[ 4, 5, 6 ]
gap> R:=PolynomialRing(GF(3),["z"]);
GF(3)[z]
gap> List(IndeterminatesOfPolynomialRing(R),
>   IndeterminateNumberOfLaurentPolynomial);
[ 3 ]
```

66.1.1 Indeterminate

▷ Indeterminate(*R[, nr]*) (operation)
▷ Indeterminate(*R[, name][, avoid]*) (operation)
▷ Indeterminate(*fam, nr*) (operation)
▷ X(*R[, nr]*) (operation)
▷ X(*R[, name][, avoid]*) (operation)
▷ X(*fam, nr*) (operation)

returns the indeterminate number *nr* over the ring *R*. If *nr* is not given it defaults to 1. If the number is not specified a list *avoid* of indeterminates may be given. The function will return an indeterminate that is guaranteed to be different from all the indeterminates in the list *avoid*. The third usage returns an indeterminate called *name* (also avoiding the indeterminates in *avoid* if given).

X is simply a synonym for Indeterminate. However, we do not recommend to use this synonym which is supported only for the backwards compatibility.

Example

```
gap> x:=Indeterminate(GF(3),"x");
x
gap> y:=X(GF(3),"y");z:=X(GF(3),"X");
y
X
gap> X(GF(3),2);
y
gap> X(GF(3),"x_3");
X
gap> X(GF(3),[y,z]);
x
```

66.1.2 IndeterminateNumberOfUnivariateRationalFunction

▷ IndeterminateNumberOfUnivariateRationalFunction(*rfun*) (attribute)

returns the number of the indeterminate in which the univariate rational function *rfun* is expressed. (This also provides a way to obtain the number of a given indeterminate.)

A constant rational function might not possess an indeterminate number. In this case IndeterminateNumberOfUnivariateRationalFunction will default to a value of 1. Therefore two univariate polynomials may be considered to be in the same univariate polynomial ring if their indeterminates have the same number or one if of them is constant. (see also CIUnivPols (66.1.5) and IsLaurentPolynomialDefaultRep (66.21.7)).

66.1.3 IndeterminateOfUnivariateRationalFunction

▷ IndeterminateOfUnivariateRationalFunction(*rfun*) (attribute)

returns the indeterminate in which the univariate rational function *rfun* is expressed. (cf. IndeterminateNumberOfUnivariateRationalFunction (66.1.2).)

Example

```
gap> IndeterminateNumberOfUnivariateRationalFunction(z);
3
gap> IndeterminateOfUnivariateRationalFunction(z^5+z);
X
```

66.1.4 IndeterminateName

▷ IndeterminateName(*fam, nr*) (operation)
▷ HasIndeterminateName(*fam, nr*) (operation)
▷ SetIndeterminateName(*fam, nr, name*) (operation)

SetIndeterminateName assigns the name *name* to indeterminate *nr* in the rational functions family *fam*. It issues an error if the indeterminate was already named.

IndeterminateName returns the name of the *nr*-th indeterminate (and returns fail if no name has been assigned).

HasIndeterminateName tests whether indeterminate *nr* has already been assigned a name.

Example

```
gap> IndeterminateName(FamilyObj(x),2);
"y"
gap> HasIndeterminateName(FamilyObj(x),4);
false
gap> SetIndeterminateName(FamilyObj(x),10,"bla");
gap> Indeterminate(GF(3),10);
bla
```

As a convenience there is a special method installed for SetName that will assign a name to an indeterminate.

Example

```
gap> a:=Indeterminate(GF(3),5);
x_5
gap> SetName(a,"ah");
gap> a^5+a;
ah^5+ah
```

66.1.5 CIUnivPols

▷ CIUnivPols(*upol1, upol2*) (function)

This function (whose name stands for "common indeterminate of univariate polynomials") takes two univariate polynomials as arguments. If both polynomials are given in the same indeterminate number *indnum* (in this case they are "compatible" as univariate polynomials) it returns *indnum*. In all other cases it returns fail. CIUnivPols also accepts if either polynomial is constant but formally expressed in another indeterminate, in this situation the indeterminate of the other polynomial is selected.

66.2 Operations for Rational Functions

The rational functions form a field, therefore all arithmetic operations are applicable to rational functions.

f + *g*
f - *g*
f * *g*
f / *g*

Example

```
gap> x:=Indeterminate(Rationals,1);;y:=Indeterminate(Rationals,2);;
gap> f:=3+x*y+x^5;;g:=5+x^2*y+x*y^2;;
gap> a:=g/f;
(x_1^2*x_2+x_1*x_2^2+5)/(x_1^5+x_1*x_2+3)
```

Note that the quotient *f*/*g* of two polynomials might be represented as a rational function again. If *g* is known to divide *f* the call Quotient(*f*,*g*) (see Quotient (56.1.9)) should be used instead.

f mod *g*

For two Laurent polynomials *f* and *g*, *f* `mod` *g* is the Euclidean remainder (see `EuclideanRemainder` (56.6.4)) of *f* modulo *g*.

As calculating a multivariate Gcd can be expensive, it is not guaranteed that rational functions will always be represented as a quotient of coprime polynomials. In certain unfortunate situations this might lead to a degree explosion. To ensure cancellation you can use `Gcd` (56.7.1) on the `NumeratorOfRationalFunction` (66.4.2) and `DenominatorOfRationalFunction` (66.4.3) values of a given rational function.

All polynomials as well as all the univariate polynomials in the same indeterminate form subrings of this field. If two rational functions are known to be in the same subring, the result will be expressed as element in this subring.

66.3 Comparison of Rational Functions

f `=` *g*

Two rational functions *f* and *g* are equal if the product `Numerator(`*f*`) * Denominator(`*g*`)` equals `Numerator(`*g*`) * Denominator(`*f*`)`.

Example

```
gap> x:=Indeterminate(Rationals,"x");;y:=Indeterminate(Rationals,"y");;
gap> f:=3+x*y+x^5;;g:=5+x^2*y+x*y^2;;
gap> a:=g/f;
(x^2*y+x*y^2+5)/(x^5+x*y+3)
gap> b:=(g*f)/(f^2);
(x^7*y+x^6*y^2+5*x^5+x^3*y^2+x^2*y^3+3*x^2*y+3*x*y^2+5*x*y+15)/(x^10+2\
*x^6*y+6*x^5+x^2*y^2+6*x*y+9)
gap> a=b;
true
```

f `<` *g*

The ordering of rational functions is defined in several steps. Monomials (products of indeterminates) are sorted first by degree, then lexicographically (with $x_1 > x_2$) (see `MonomialGrlexOrdering` (66.17.8)). Products of monomials with ring elements ("terms") are compared first by their monomials and then by their coefficients.

Example

```
gap> x>y;
true
gap> x^2*y<x*y^2;
false
gap> x*y<x^2*y;
true
gap> x^2*y < 5* y*x^2;
true
```

Polynomials are compared by comparing the largest terms in turn until they differ.

Example

```
gap> x+y<y;
false
gap> x<x+1;
true
```

Rational functions are compared by comparing the polynomial `Numerator(`*f*`) * Denominator(`*g*`)` with the polynomial `Numerator(`*g*`) * Denominator(`*f*`)`. (As the ordering of monomials used by GAP is invariant under multiplication this is independent of common factors in numerator and denominator.)

Example

```
gap> f/g<g/f;
false
gap> f/g<(g*g)/(f*g);
false
```

For univariate polynomials this reduces to an ordering first by total degree and then lexicographically on the coefficients.

66.4 Properties and Attributes of Rational Functions

All these tests are applicable to *every* rational function. Depending on the internal representation of the rational function, however some of these tests (in particular, univariateness) might be expensive in some cases.

For reasons of performance within algorithms it can be useful to use other attributes, which give a slightly more technical representation. See section 66.20 for details.

66.4.1 IsPolynomialFunction

▷ `IsPolynomialFunction(`*obj*`)` (Category)
▷ `IsRationalFunction(`*obj*`)` (Category)

A rational function is an element of the quotient field of a polynomial ring over an UFD. It is represented as a quotient of two polynomials, its numerator (see `NumeratorOfRationalFunction` (66.4.2)) and its denominator (see `DenominatorOfRationalFunction` (66.4.3))

A polynomial function is an element of a polynomial ring (not necessarily an UFD), or a rational function.

GAP considers `IsRationalFunction` as a subcategory of `IsPolynomialFunction`.

66.4.2 NumeratorOfRationalFunction

▷ `NumeratorOfRationalFunction(`*ratfun*`)` (attribute)

returns the numerator of the rational function *ratfun*.

As no proper multivariate gcd has been implemented yet, numerators and denominators are not guaranteed to be reduced!

66.4.3 DenominatorOfRationalFunction

▷ `DenominatorOfRationalFunction(`*ratfun*`)` (attribute)

returns the denominator of the rational function *ratfun*.

As no proper multivariate gcd has been implemented yet, numerators and denominators are not guaranteed to be reduced!

Example

```
gap> x:=Indeterminate(Rationals,1);;y:=Indeterminate(Rationals,2);;
gap> DenominatorOfRationalFunction((x*y+x^2)/y);
y
gap> NumeratorOfRationalFunction((x*y+x^2)/y);
x^2+x*y
```

66.4.4 IsPolynomial

▷ IsPolynomial(*ratfun*) (property)

A polynomial is a rational function whose denominator is one. (If the coefficients family forms a field this is equivalent to the denominator being constant.)

If the base family is not a field, it may be impossible to represent the quotient of a polynomial by a ring element as a polynomial again, but it will have to be represented as a rational function.

Example

```
gap> IsPolynomial((x*y+x^2*y^3)/y);
true
gap> IsPolynomial((x*y+x^2)/y);
false
```

66.4.5 AsPolynomial

▷ AsPolynomial(*poly*) (attribute)

If *poly* is a rational function that is a polynomial this attribute returns an equal rational function p such that p is equal to its numerator and the denominator of p is one.

Example

```
gap> AsPolynomial((x*y+x^2*y^3)/y);
x^2*y^2+x
```

66.4.6 IsUnivariateRationalFunction

▷ IsUnivariateRationalFunction(*ratfun*) (property)

A rational function is univariate if its numerator and its denominator are both polynomials in the same one indeterminate. The attribute IndeterminateNumberOfUnivariateRationalFunction (66.1.2) can be used to obtain the number of this common indeterminate.

66.4.7 CoefficientsOfUnivariateRationalFunction

▷ CoefficientsOfUnivariateRationalFunction(*rfun*) (attribute)

if *rfun* is a univariate rational function, this attribute returns a list [*ncof*, *dcof*, *val*] where *ncof* and *dcof* are coefficient lists of univariate polynomials *n* and *d* and a valuation *val* such that $rfun = x^{val} \cdot n/d$ where x is the variable with the number given by IndeterminateNumberOfUnivariateRationalFunction (66.1.2). Numerator and denominator are guaranteed to be cancelled.

66.4.8 IsUnivariatePolynomial

▷ `IsUnivariatePolynomial(`*ratfun*`)` (property)

A univariate polynomial is a polynomial in only one indeterminate.

66.4.9 CoefficientsOfUnivariatePolynomial

▷ `CoefficientsOfUnivariatePolynomial(`*pol*`)` (attribute)

`CoefficientsOfUnivariatePolynomial` returns the coefficient list of the polynomial *pol*, sorted in ascending order. (It returns the empty list if *pol* is 0.)

66.4.10 IsLaurentPolynomial

▷ `IsLaurentPolynomial(`*ratfun*`)` (property)

A Laurent polynomial is a univariate rational function whose denominator is a monomial. Therefore every univariate polynomial is a Laurent polynomial.

The attribute `CoefficientsOfLaurentPolynomial` (66.13.2) gives a compact representation as Laurent polynomial.

66.4.11 IsConstantRationalFunction

▷ `IsConstantRationalFunction(`*ratfun*`)` (property)

A constant rational function is a function whose numerator and denominator are polynomials of degree 0.

66.4.12 IsPrimitivePolynomial

▷ `IsPrimitivePolynomial(`*F*`, `*pol*`)` (operation)

For a univariate polynomial *pol* of degree d in the indeterminate X, with coefficients in a finite field *F* with q elements, say, `IsPrimitivePolynomial` returns `true` if

1. *pol* divides $X^{q^d-1}-1$, and
2. for each prime divisor p of q^d-1, *pol* does not divide $X^{(q^d-1)/p}-1$,

and `false` otherwise.

66.4.13 SplittingField

▷ `SplittingField(`*f*`)` (attribute)

returns the smallest field which contains the coefficients of *f* and the roots of *f*.

66.5 Univariate Polynomials

Some of the operations are actually defined on the larger domain of Laurent polynomials (see 66.13). For this section you can simply ignore the word "Laurent" if it occurs in a description.

66.5.1 UnivariatePolynomial

▷ `UnivariatePolynomial(`*`ring, cofs[, ind]`*`)` (operation)

constructs an univariate polynomial over the ring *`ring`* in the indeterminate *`ind`* with the coefficients given by *`coefs`*.

66.5.2 UnivariatePolynomialByCoefficients

▷ `UnivariatePolynomialByCoefficients(`*`fam, cofs, ind`*`)` (operation)

constructs an univariate polynomial over the coefficients family *`fam`* and in the indeterminate *`ind`* with the coefficients given by *`coefs`*. This function should be used in algorithms to create polynomials as it avoids overhead associated with `UnivariatePolynomial` (66.5.1).

66.5.3 DegreeOfLaurentPolynomial

▷ `DegreeOfLaurentPolynomial(`*`pol`*`)` (attribute)

The degree of a univariate (Laurent) polynomial *`pol`* is the largest exponent n of a monomial x^n of *`pol`*. The degree of a zero polynomial is defined to be `-infinity`.

Example

```
gap> p:=UnivariatePolynomial(Rationals,[1,2,3,4],1);
4*x^3+3*x^2+2*x+1
gap> UnivariatePolynomialByCoefficients(FamilyObj(1),[9,2,3,4],73);
4*x_73^3+3*x_73^2+2*x_73+9
gap> CoefficientsOfUnivariatePolynomial(p);
[ 1, 2, 3, 4 ]
gap> DegreeOfLaurentPolynomial(p);
3
gap> DegreeOfLaurentPolynomial(Zero(p));
-infinity
gap> IndeterminateNumberOfLaurentPolynomial(p);
1
gap> IndeterminateOfLaurentPolynomial(p);
x
```

66.5.4 RootsOfPolynomial

▷ `RootsOfPolynomial(`*`[R, ]p`*`)` (function)

For a univariate polynomial *`p`*, this function returns all roots of *`p`* over the ring *`R`*. If the ring is not specified, it defaults to the ring specified by the coefficients of *`p`* via `DefaultRing` (56.1.3)).

Example

```
gap> x:=X(Rationals,"x");;p:=x^4-1;
x^4-1
gap> RootsOfPolynomial(p);
[ 1, -1 ]
gap> RootsOfPolynomial(CF(4),p);
[ 1, -1, E(4), -E(4) ]
```

66.5.5 RootsOfUPol

▷ RootsOfUPol(*[field,]upol*) (function)

This function returns a list of all roots of the univariate polynomial *upol* in its default domain. If the optional argument *field* is a field then the roots in this field are computed. If *field* is the string "split" then the splitting field of the polynomial is taken.

Example

```
gap> RootsOfUPol(50-45*x-6*x^2+x^3);
[ 10, 1, -5 ]
```

66.5.6 QuotRemLaurpols

▷ QuotRemLaurpols(*left, right, mode*) (function)

This internal function for euclidean division of polynomials takes two polynomials *left* and *right* and computes their quotient. No test is performed whether the arguments indeed are polynomials. Depending on the integer variable *mode*, which may take values in a range from 1 to 4, it returns respectively:

1. the quotient (there might be some remainder),
2. the remainder,
3. a list [*q*,*r*] of quotient and remainder,
4. the quotient if there is no remainder and fail otherwise.

66.5.7 UnivariatenessTestRationalFunction

▷ UnivariatenessTestRationalFunction(*f*) (function)

takes a rational function *f* and tests whether it is univariate rational function (or even a Laurent polynomial). It returns a list [isunivariate, indet, islaurent, cofs].

If *f* is a univariate rational function then isunivariate is true and indet is the number of the appropriate indeterminate.

Furthermore, if *f* is a Laurent polynomial, then islaurent is also true. In this case the fourth entry, cofs, is the value of the attribute CoefficientsOfLaurentPolynomial (66.13.2) for *f*.

If isunivariate is true but islaurent is false, then cofs is the value of the attribute CoefficientsOfUnivariateRationalFunction (66.4.7) for *f*.

Otherwise, each entry of the returned list is equal to fail. As there is no proper multivariate gcd, this may also happen for the rational function which may be reduced to univariate (see example).

Example
```
gap> UnivariatenessTestRationalFunction( 50-45*x-6*x^2+x^3 );
[ true, 1, true, [ [ 50, -45, -6, 1 ], 0 ] ]
gap> UnivariatenessTestRationalFunction( (-6*y^2+y^3) / (y+1) );
[ true, 2, false, [ [ -6, 1 ], [ 1, 1 ], 2 ] ]
gap> UnivariatenessTestRationalFunction( (-6*y^2+y^3) / (x+1));
[ fail, fail, fail, fail ]
gap> UnivariatenessTestRationalFunction( ((y+2)*(x+1)) / ((y-1)*(x+1)) );
[ fail, fail, fail, fail ]
```

66.5.8 InfoPoly

▷ `InfoPoly` (info class)

is the info class for univariate polynomials.

We remark that some functions for multivariate polynomials (which will be defined in the following sections) permit a different syntax for univariate polynomials which drops the requirement to specify the indeterminate. Examples are `Value` (66.7.1), `Discriminant` (66.6.6), `Derivative` (66.6.5), `LeadingCoefficient` (66.6.3) and `LeadingMonomial` (66.6.4):

Example
```
gap> p:=UnivariatePolynomial(Rationals,[1,2,3,4],1);
4*x^3+3*x^2+2*x+1
gap> Value(p,Z(5));
Z(5)^2
gap> LeadingCoefficient(p);
4
gap> Derivative(p);
12*x^2+6*x+2
```

66.6 Polynomials as Univariate Polynomials in one Indeterminate

66.6.1 DegreeIndeterminate

▷ `DegreeIndeterminate(`*pol*`, `*ind*`)` (operation)

returns the degree of the polynomial *pol* in the indeterminate (or indeterminate number) *ind*.

Example
```
gap> f:=x^5+3*x*y+9*y^7+4*y^5*x+3*y+2;
9*y^7+4*x*y^5+x^5+3*x*y+3*y+2
gap> DegreeIndeterminate(f,1);
5
gap> DegreeIndeterminate(f,y);
7
```

66.6.2 PolynomialCoefficientsOfPolynomial

▷ `PolynomialCoefficientsOfPolynomial(`*pol*`, `*ind*`)` (operation)

PolynomialCoefficientsOfPolynomial returns the coefficient list (whose entries are polynomials not involving the indeterminate *ind*) describing the polynomial *pol* viewed as a polynomial in *ind*. Instead of the indeterminate, *ind* can also be an indeterminate number.

Example

```
gap> PolynomialCoefficientsOfPolynomial(f,2);
[ x^5+2, 3*x+3, 0, 0, 0, 4*x, 0, 9 ]
```

66.6.3 LeadingCoefficient

▷ LeadingCoefficient(*pol*) (operation)

returns the leading coefficient (that is the coefficient of the leading monomial, see LeadingMonomial (66.6.4)) of the polynomial *pol*.

66.6.4 LeadingMonomial

▷ LeadingMonomial(*pol*) (function)

returns the leading monomial (with respect to the ordering given by MonomialExtGrlexLess (66.17.14)) of the polynomial *pol* as a list containing indeterminate numbers and exponents.

Example

```
gap> LeadingCoefficient(f,1);
1
gap> LeadingCoefficient(f,2);
9
gap> LeadingMonomial(f);
[ 2, 7 ]
gap> LeadingCoefficient(f);
9
```

66.6.5 Derivative

▷ Derivative(*ratfun[, ind]*) (attribute)

If *ratfun* is a univariate rational function then Derivative returns the *derivative* of *ufun* by its indeterminate. For a rational function *ratfun*, the derivative by the indeterminate *ind* is returned, regarding *ratfun* as univariate in *ind*. Instead of the desired indeterminate, also the number of this indeterminate can be given as *ind*.

Example

```
gap> Derivative(f,2);
63*y^6+20*x*y^4+3*x+3
```

66.6.6 Discriminant

▷ Discriminant(*pol[, ind]*) (operation)

If *pol* is a univariate polynomial then Discriminant returns the *discriminant* of *pol* by its indeterminate. The two-argument form returns the discriminant of a polynomial *pol* by the indeterminate

number *ind*, regarding *pol* as univariate in this indeterminate. Instead of the indeterminate number, the indeterminate itself can also be given as *ind*.

Example

```
gap> Discriminant(f,1);
20503125*y^28+262144*y^25+27337500*y^22+19208040*y^21+1474560*y^17+136\
68750*y^16+18225000*y^15+6075000*y^14+1105920*y^13+3037500*y^10+648972\
0*y^9+4050000*y^8+900000*y^7+62208*y^5+253125*y^4+675000*y^3+675000*y^\
2+300000*y+50000
gap> Discriminant(f,1) = Discriminant(f,x);
true
```

66.6.7 Resultant

▷ Resultant(*pol1, pol2, ind*) (operation)

computes the resultant of the polynomials *pol1* and *pol2* with respect to the indeterminate *ind*, or indeterminate number *ind*. The resultant considers *pol1* and *pol2* as univariate in *ind* and returns an element of the corresponding base ring (which might be a polynomial ring).

Example

```
gap> Resultant(x^4+y,y^4+x,1);
y^16+y
gap> Resultant(x^4+y,y^4+x,2);
x^16+x
```

66.7 Multivariate Polynomials

66.7.1 Value

▷ Value(*ratfun, indets, vals[, one]*) (operation)
▷ Value(*upol, value[, one]*) (operation)

The first variant takes a rational function *ratfun* and specializes the indeterminates given in *indets* to the values given in *vals*, replacing the *i*-th entry in *indets* by the *i*-th entry in *vals*. If this specialization results in a constant polynomial, an element of the coefficient ring is returned. If the specialization would specialize the denominator of *ratfun* to zero, an error is raised.

A variation is the evaluation at elements of another ring R, for which a multiplication with elements of the coefficient ring of *ratfun* are defined. In this situation the identity element of R may be given by a further argument *one* which will be used for x^0 for any specialized indeterminate x.

The second version takes an univariate rational function and specializes the value of its indeterminate to *val*. Again, an optional argument *one* may be given.

Example

```
gap> Value(x*y+y+x^7,[x,y],[5,7]);
78167
```

Note that the default values for *one* can lead to different results than one would expect: For example for a matrix M, the values $M + M^0$ and $M + 1$ are *different*. As Value defaults to the one of the coefficient ring, when evaluating matrices in polynomials always the correct *one* should be given!

66.8 Minimal Polynomials

66.8.1 MinimalPolynomial

▷ MinimalPolynomial(*R, elm[, ind]*) (operation)

returns the *minimal polynomial* of *elm* over the ring *R*, expressed in the indeterminate number *ind*. If *ind* is not given, it defaults to 1.

The minimal polynomial is the monic polynomial of smallest degree with coefficients in *R* that has value zero at *elm*.

Example

```
gap> MinimalPolynomial(Rationals,[[2,0],[0,2]]);
x-2
```

66.9 Cyclotomic Polynomials

66.9.1 CyclotomicPolynomial

▷ CyclotomicPolynomial(*F, n*) (function)

is the *n*-th cyclotomic polynomial over the ring *F*.

Example

```
gap> CyclotomicPolynomial(Rationals,5);
x^4+x^3+x^2+x+1
```

66.10 Polynomial Factorization

At the moment GAP provides only methods to factorize polynomials over finite fields (see Chapter 59), over subfields of cyclotomic fields (see Chapter 60), and over algebraic extensions of these (see Chapter 67).

66.10.1 Factors (of polynomial)

▷ Factors(*[R,]poly[, opt]*) (method)

returns a list of the irreducible factors of the polynomial *poly* in the polynomial ring *R*. (That is factors over the CoefficientsRing (66.15.3) value of *R*.)

For univariate factorizations, it is possible to pass a record *opt* as a third argument. This record can contain the following components:

onlydegs
: is a set of positive integers. The factorization assumes that all irreducible factors have a degree in this set.

stopdegs
: is a set of positive integers. The factorization will stop once a factor of degree in stopdegs has been found and will return the factorization found so far.

Example

```
gap> f:= CyclotomicPolynomial( GF(2), 7 );
x_1^6+x_1^5+x_1^4+x_1^3+x_1^2+x_1+Z(2)^0
gap> Factors( f );
[ x_1^3+x_1+Z(2)^0, x_1^3+x_1^2+Z(2)^0 ]
gap> Factors( PolynomialRing( GF(8) ), f );
[ x_1+Z(2^3), x_1+Z(2^3)^2, x_1+Z(2^3)^3, x_1+Z(2^3)^4, x_1+Z(2^3)^5,
  x_1+Z(2^3)^6 ]
gap> f:= MinimalPolynomial( Rationals, E(4) );
x^2+1
gap> Factors( f );
[ x^2+1 ]
gap> Factors( PolynomialRing( Rationals ), f );
[ x^2+1 ]
gap> Factors( PolynomialRing( CF(4) ), f );
[ x+(-E(4)), x+E(4) ]
```

66.10.2 FactorsSquarefree

▷ `FactorsSquarefree(`*pring, upol, opt*`)` (operation)

returns a factorization of the squarefree, monic, univariate polynomial *upol* in the polynomial ring *pring*; *opt* must be a (possibly empty) record of options. *upol* must not have zero as a root. This function is used by the factoring algorithms.

The current method for multivariate factorization reduces to univariate factorization by use of a reduction homomorphism of the form $f(x_1,x_2,x_3) \mapsto f(x,x^p,x^{p^2})$. It can be very time intensive for larger degrees.

Example

```
gap> Factors(x^10-y^10);
[ x-y, x+y, x^4-x^3*y+x^2*y^2-x*y^3+y^4, x^4+x^3*y+x^2*y^2+x*y^3+y^4 ]
```

66.11 Polynomials over the Rationals

The following functions are only available to polynomials with rational coefficients:

66.11.1 PrimitivePolynomial

▷ `PrimitivePolynomial(`*f*`)` (function)

takes a polynomial *f* with rational coefficients and computes a new polynomial with integral coefficients, obtained by multiplying with the Lcm of the denominators of the coefficients and casting out the content (the Gcd of the coefficients). The operation returns a list [*newpol*,*coeff*] with rational *coeff* such that *coeff*`*`*newpol*`=`*f*.

66.11.2 PolynomialModP

▷ `PolynomialModP(`*pol, p*`)` (function)

for a rational polynomial *pol* this function returns a polynomial over the field with *p* elements, obtained by reducing the coefficients modulo *p*.

66.11.3 GaloisType

▷ GaloisType(*f[, cand]*) (function)

Let *f* be an irreducible polynomial with rational coefficients. This function returns the type of Gal(*f*) (considered as a transitive permutation group of the roots of *f*). It returns a number *i* if Gal(*f*) is permutation isomorphic to TransitiveGroup(*n*,*i*) where *n* is the degree of *f*.

Identification is performed by factoring appropriate Galois resolvents as proposed in [SM85]. This function is provided for rational polynomials of degree up to 15. However, in some cases the required calculations become unfeasibly large.

For a few polynomials of degree 14, a complete discrimination is not yet possible, as it would require computations, that are not feasible with current factoring methods.

This function requires the transitive groups library to be installed (see 50.6).

66.11.4 ProbabilityShapes

▷ ProbabilityShapes(*f*) (function)

Let *f* be an irreducible polynomial with rational coefficients. This function returns a list of the most likely type(s) of Gal(*f*) (see GaloisType (66.11.3)), based on factorization modulo a set of primes. It is very fast, but the result is only probabilistic.

This function requires the transitive groups library to be installed (see 50.6).

Example

```
gap> f:=x^9-9*x^7+27*x^5-39*x^3+36*x-8;;
gap> GaloisType(f);
25
gap> TransitiveGroup(9,25);
[1/2.S(3)^3]3
gap> ProbabilityShapes(f);
[ 25 ]
```

66.12 Factorization of Polynomials over the Rationals

The following operations are used by GAP inside the factorization algorithm but might be of interest also in other contexts.

66.12.1 BombieriNorm

▷ BombieriNorm(*pol*) (function)

computes weighted Norm $[pol]_2$ of *pol* which is a good measure for factor coefficients (see [BTW93]).

66.12.2 MinimizedBombieriNorm

▷ `MinimizedBombieriNorm(f)` (attribute)

This function applies linear Tschirnhaus transformations ($x \mapsto x+i$) to the polynomial `f`, trying to get the Bombieri norm of `f` small. It returns a list `[new_polynomial, i_of_transformation]`.

66.12.3 HenselBound

▷ `HenselBound(pol[, minpol, den])` (function)

returns the Hensel bound of the polynomial `pol`. If the computation takes place over an algebraic extension, then the minimal polynomial `minpol` and denominator `den` must be given.

66.12.4 OneFactorBound

▷ `OneFactorBound(pol)` (function)

returns the coefficient bound for a single factor of the rational polynomial `pol`.

66.13 Laurent Polynomials

A univariate polynomial can be written in the form $r_0 + r_1 x + \cdots + r_n x^n$, with $r_i \in R$. Formally, there is no reason to start with 0, if m is an integer, we can consider objects of the form $r_m x^m + r_{m+1} x^{m+1} + \cdots + r_n x^n$. We call these *Laurent polynomials*. Laurent polynomials also can be considered as quotients of a univariate polynomial by a power of the indeterminate. The addition and multiplication of univariate polynomials extends to Laurent polynomials (though it might be impossible to interpret a Laurent polynomial as a function) and many functions for univariate polynomials extend to Laurent polynomials (or extended versions for Laurent polynomials exist).

66.13.1 LaurentPolynomialByCoefficients

▷ `LaurentPolynomialByCoefficients(fam, cofs, val[, ind])` (operation)

constructs a Laurent polynomial over the coefficients family `fam` and in the indeterminate `ind` (defaulting to 1) with the coefficients given by `coefs` and valuation `val`.

66.13.2 CoefficientsOfLaurentPolynomial

▷ `CoefficientsOfLaurentPolynomial(laurent)` (attribute)

For a Laurent polynomial `laurent`, this function returns a pair `[cof, val]`, consisting of the coefficient list (in ascending order) `cof` and the valuation `val` of `laurent`.

Example

```
gap> p:=LaurentPolynomialByCoefficients(FamilyObj(1),
> [1,2,3,4,5],-2);
5*x^2+4*x+3+2*x^-1+x^-2
gap> NumeratorOfRationalFunction(p);DenominatorOfRationalFunction(p);
```

```
5*x^4+4*x^3+3*x^2+2*x+1
x^2
gap> CoefficientsOfLaurentPolynomial(p*p);
[ [ 1, 4, 10, 20, 35, 44, 46, 40, 25 ], -4 ]
```

66.13.3 IndeterminateNumberOfLaurentPolynomial

▷ `IndeterminateNumberOfLaurentPolynomial(`*pol*`)` (function)

Is a synonym for `IndeterminateNumberOfUnivariateRationalFunction` (66.1.2).

66.14 Univariate Rational Functions

66.14.1 UnivariateRationalFunctionByCoefficients

▷ `UnivariateRationalFunctionByCoefficients(`*fam, ncof, dcof, val[, ind]*`)` (operation)

constructs a univariate rational function over the coefficients family *fam* and in the indeterminate *ind* (defaulting to 1) with numerator and denominator coefficients given by *ncof* and *dcof* and valuation *val*.

66.15 Polynomial Rings and Function Fields

While polynomials depend only on the family of the coefficients, polynomial rings A are defined over a base ring R. A polynomial is an element of A if and only if all its coefficients are contained in R. Besides providing domains and an easy way to create polynomials, polynomial rings can affect the behavior of operations like factorization into irreducibles.

If you need to work with a polynomial ring and its indeterminates the following two approaches will produce a ring that contains given variables (see section 66.1 for details about the internal numbering): Either, first create the ring and then get the indeterminates with `IndeterminatesOfPolynomialRing` (66.15.2).

Example

```
gap> r := PolynomialRing(Rationals,["a","b"]);;
gap> indets := IndeterminatesOfPolynomialRing(r);;
gap> a := indets[1]; a := indets[2];
a
b
```

Alternatively, first create the indeterminates and then create the ring including these indeterminates.

Example

```
gap> a:=Indeterminate(Rationals,"a":old);;
gap> b:=Indeterminate(Rationals,"b":old);;
gap> PolynomialRing(Rationals,[a,b]);;
```

As a convenient shortcut, intended mainly for interactive working, the i-th indeterminate of a polynomial ring R can be accessed as $R.i$, which corresponds exactly to

`IndeterminatesOfPolynomialRing`$(R)[i]$ or, if it has the name `nam`, as R`.nam`. *Note* that the number i is in general *not* the indeterminate number, but simply an index into the indeterminates list of R.

Example

```
gap> r := PolynomialRing(Rationals, ["a", "b"]:old );;
gap> r.1; r.2; r.a; r.b;
a
b
a
b
gap> IndeterminateNumberOfLaurentPolynomial(r.1);
3
```

Polynomials as GAP objects can exist without a polynomial ring being defined and polynomials cannot be associated to a particular polynomial ring. (For example dividing a polynomial which is in a polynomial ring over the integers by another integer will result in a polynomial over the rationals, not in a rational function over the integers.)

66.15.1 PolynomialRing

▷ `PolynomialRing(`*R, rank[, avoid]*`)` (operation)
▷ `PolynomialRing(`*R, names[, avoid]*`)` (operation)
▷ `PolynomialRing(`*R, indets*`)` (operation)
▷ `PolynomialRing(`*R, indetnums*`)` (operation)

creates a polynomial ring over the ring *R*. If a positive integer *rank* is given, this creates the polynomial ring in *rank* indeterminates. These indeterminates will have the internal index numbers 1 to *rank*. The second usage takes a list *names* of strings and returns a polynomial ring in indeterminates labelled by *names*. These indeterminates have "new" internal index numbers as if they had been created by calls to `Indeterminate` (66.1.1). (If the argument *avoid* is given it contains indeterminates that should be avoided, in this case internal index numbers are incremented to skip these variables.) In the third version, a list of indeterminates *indets* is given. This creates the polynomial ring in the indeterminates *indets*. Finally, the fourth version specifies indeterminates by their index numbers.

To get the indeterminates of a polynomial ring use `IndeterminatesOfPolynomialRing` (66.15.2). (Indeterminates created independently with `Indeterminate` (66.1.1) will usually differ, though they might be given the same name and display identically, see Section 66.1.)

66.15.2 IndeterminatesOfPolynomialRing

▷ `IndeterminatesOfPolynomialRing(`*pring*`)` (attribute)
▷ `IndeterminatesOfFunctionField(`*ffield*`)` (attribute)

returns a list of the indeterminates of the polynomial ring *pring*, respectively the function field *ffield*.

66.15.3 CoefficientsRing

▷ `CoefficientsRing(`*pring*`)` (attribute)

returns the ring of coefficients of the polynomial ring *pring*, that is the ring over which *pring* was defined.

Example

```
gap> r:=PolynomialRing(GF(7));
GF(7)[x_1]
gap> r:=PolynomialRing(GF(7),3);
GF(7)[x_1,x_2,x_3]
gap> IndeterminatesOfPolynomialRing(r);
[ x_1, x_2, x_3 ]
gap> r2:=PolynomialRing(GF(7),[5,7,12]);
GF(7)[x_5,x_7,x_12]
gap> CoefficientsRing(r);
GF(7)
gap> r:=PolynomialRing(GF(7),3);
GF(7)[x_1,x_2,x_3]
gap> r2:=PolynomialRing(GF(7),3,IndeterminatesOfPolynomialRing(r));
GF(7)[x_4,x_5,x_6]
gap> r:=PolynomialRing(GF(7),["x","y","z","z2"]);
GF(7)[x,y,z,z2]
```

66.15.4 IsPolynomialRing

▷ `IsPolynomialRing(`*pring*`)` (Category)

is the category of polynomial rings

66.15.5 IsFiniteFieldPolynomialRing

▷ `IsFiniteFieldPolynomialRing(`*pring*`)` (Category)

is the category of polynomial rings over a finite field (see Chapter 59).

66.15.6 IsAbelianNumberFieldPolynomialRing

▷ `IsAbelianNumberFieldPolynomialRing(`*pring*`)` (Category)

is the category of polynomial rings over a field of cyclotomics (see the chapters 18 and 60).

66.15.7 IsRationalsPolynomialRing

▷ `IsRationalsPolynomialRing(`*pring*`)` (Category)

is the category of polynomial rings over the rationals (see Chapter 17).

Example

```
gap> r := PolynomialRing(Rationals, ["a", "b"] );;
gap> IsPolynomialRing(r);
true
gap> IsFiniteFieldPolynomialRing(r);
false
gap> IsRationalsPolynomialRing(r);
true
```

66.15.8 FunctionField

▷ FunctionField(*R, rank[, avoid]*) (operation)
▷ FunctionField(*R, names[, avoid]*) (operation)
▷ FunctionField(*R, indets*) (operation)
▷ FunctionField(*R, indetnums*) (operation)

creates a function field over the integral ring *R*. If a positive integer *rank* is given, this creates the function field in *rank* indeterminates. These indeterminates will have the internal index numbers 1 to *rank*. The second usage takes a list *names* of strings and returns a function field in indeterminates labelled by *names*. These indeterminates have "new" internal index numbers as if they had been created by calls to Indeterminate (66.1.1). (If the argument *avoid* is given it contains indeterminates that should be avoided, in this case internal index numbers are incremented to skip these variables.) In the third version, a list of indeterminates *indets* is given. This creates the function field in the indeterminates *indets*. Finally, the fourth version specifies indeterminates by their index number.

To get the indeterminates of a function field use IndeterminatesOfFunctionField (66.15.2). (Indeterminates created independently with Indeterminate (66.1.1) will usually differ, though they might be given the same name and display identically, see Section 66.1.)

66.15.9 IsFunctionField

▷ IsFunctionField(*ffield*) (Category)

is the category of function fields

66.16 Univariate Polynomial Rings

66.16.1 UnivariatePolynomialRing

▷ UnivariatePolynomialRing(*R[, nr]*) (operation)
▷ UnivariatePolynomialRing(*R[, name][, avoid]*) (operation)

returns a univariate polynomial ring in the indeterminate *nr* over the base ring *R*. If *nr* is not given it defaults to 1.

If the number is not specified a list *avoid* of indeterminates may be given. Then the function will return a ring in an indeterminate that is guaranteed to be different from all the indeterminates in *avoid*.

Also a string *name* can be prescribed as the name of the indeterminate chosen (also avoiding the indeterminates in the list *avoid* if given).

66.16.2 IsUnivariatePolynomialRing

▷ IsUnivariatePolynomialRing(*pring*) (Category)

is the category of polynomial rings with one indeterminate.

Example

```
gap> r:=UnivariatePolynomialRing(Rationals,"p");
Rationals[p]
```

```
gap> r2:=PolynomialRing(Rationals,["q"]);
Rationals[q]
gap> IsUnivariatePolynomialRing(r);
true
gap> IsUnivariatePolynomialRing(r2);
true
```

66.17 Monomial Orderings

It is often desirable to consider the monomials within a polynomial to be arranged with respect to a certain ordering. Such an ordering is called a *monomial ordering* if it is total, invariant under multiplication with other monomials and admits no infinite descending chains. For details on monomial orderings see [CLO97].

In GAP, monomial orderings are represented by objects that provide a way to compare monomials (as polynomials as well as –for efficiency purposes within algorithms– in the internal representation as lists).

Normally the ordering chosen should be *admissible*, i.e. it must be compatible with products: If $a < b$ then $ca < cb$ for all monomials a, b and c.

Each monomial ordering provides the two functions `MonomialComparisonFunction` (66.17.5) and `MonomialExtrepComparisonFun` (66.17.6) to compare monomials. These functions work as "is less than", i.e. they return `true` if and only if the left argument is smaller.

66.17.1 IsMonomialOrdering

▷ `IsMonomialOrdering(`*obj*`)` (Category)

A monomial ordering is an object representing a monomial ordering. Its attributes `MonomialComparisonFunction` (66.17.5) and `MonomialExtrepComparisonFun` (66.17.6) are actual comparison functions.

66.17.2 LeadingMonomialOfPolynomial

▷ `LeadingMonomialOfPolynomial(`*pol, ord*`)` (function)

returns the leading monomial (with respect to the ordering *ord*) of the polynomial *pol*.

Example

```
gap> x:=Indeterminate(Rationals,"x");;
gap> y:=Indeterminate(Rationals,"y");;
gap> z:=Indeterminate(Rationals,"z");;
gap> lexord:=MonomialLexOrdering();grlexord:=MonomialGrlexOrdering();
MonomialLexOrdering()
MonomialGrlexOrdering()
gap> f:=2*x+3*y+4*z+5*x^2-6*z^2+7*y^3;
7*y^3+5*x^2-6*z^2+2*x+3*y+4*z
gap> LeadingMonomialOfPolynomial(f,lexord);
x^2
gap> LeadingMonomialOfPolynomial(f,grlexord);
y^3
```

66.17.3 LeadingTermOfPolynomial

▷ `LeadingTermOfPolynomial(`*pol, ord*`)` (function)

returns the leading term (with respect to the ordering *ord*) of the polynomial *pol*, i.e. the product of leading coefficient and leading monomial.

66.17.4 LeadingCoefficientOfPolynomial

▷ `LeadingCoefficientOfPolynomial(`*pol, ord*`)` (operation)

returns the leading coefficient (that is the coefficient of the leading monomial, see `LeadingMonomialOfPolynomial` (66.17.2)) of the polynomial *pol*.

Example

```
gap> LeadingTermOfPolynomial(f,lexord);
5*x^2
gap> LeadingTermOfPolynomial(f,grlexord);
7*y^3
gap> LeadingCoefficientOfPolynomial(f,lexord);
5
```

66.17.5 MonomialComparisonFunction

▷ `MonomialComparisonFunction(`*O*`)` (attribute)

If *O* is an object representing a monomial ordering, this attribute returns a *function* that can be used to compare or sort monomials (and polynomials which will be compared by their monomials in decreasing order) in this order.

Example

```
gap> MonomialComparisonFunction(lexord);
function( a, b ) ... end
gap> l:=[f,Derivative(f,x),Derivative(f,y),Derivative(f,z)];;
gap> Sort(l,MonomialComparisonFunction(lexord));l;
[ -12*z+4, 21*y^2+3, 10*x+2, 7*y^3+5*x^2-6*z^2+2*x+3*y+4*z ]
```

66.17.6 MonomialExtrepComparisonFun

▷ `MonomialExtrepComparisonFun(`*O*`)` (attribute)

If *O* is an object representing a monomial ordering, this attribute returns a *function* that can be used to compare or sort monomials *in their external representation* (as lists). This comparison variant is used inside algorithms that manipulate the external representation.

66.17.7 MonomialLexOrdering

▷ `MonomialLexOrdering(`*[vari]*`)` (function)

This function creates a lexicographic ordering for monomials. Monomials are compared first by the exponents of the largest variable, then the exponents of the second largest variable and so on.

The variables are ordered according to their (internal) index, i.e., x_1 is larger than x_2 and so on. If *vari* is given, and is a list of variables or variable indices, instead this arrangement of variables (in descending order; i.e. the first variable is larger than the second) is used as the underlying order of variables.

Example

```
gap> l:=List(Tuples([1..3],3),i->x^(i[1]-1)*y^(i[2]-1)*z^(i[3]-1));
[ 1, z, z^2, y, y*z, y*z^2, y^2, y^2*z, y^2*z^2, x, x*z, x*z^2, x*y,
  x*y*z, x*y*z^2, x*y^2, x*y^2*z, x*y^2*z^2, x^2, x^2*z, x^2*z^2,
  x^2*y, x^2*y*z, x^2*y*z^2, x^2*y^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialLexOrdering()));l;
[ 1, z, z^2, y, y*z, y*z^2, y^2, y^2*z, y^2*z^2, x, x*z, x*z^2, x*y,
  x*y*z, x*y*z^2, x*y^2, x*y^2*z, x*y^2*z^2, x^2, x^2*z, x^2*z^2,
  x^2*y, x^2*y*z, x^2*y*z^2, x^2*y^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialLexOrdering([y,z,x])));l;
[ 1, x, x^2, z, x*z, x^2*z, z^2, x*z^2, x^2*z^2, y, x*y, x^2*y, y*z,
  x*y*z, x^2*y*z, y*z^2, x*y*z^2, x^2*y*z^2, y^2, x*y^2, x^2*y^2,
  y^2*z, x*y^2*z, x^2*y^2*z, y^2*z^2, x*y^2*z^2, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialLexOrdering([z,x,y])));l;
[ 1, y, y^2, x, x*y, x*y^2, x^2, x^2*y, x^2*y^2, z, y*z, y^2*z, x*z,
  x*y*z, x*y^2*z, x^2*z, x^2*y*z, x^2*y^2*z, z^2, y*z^2, y^2*z^2,
  x*z^2, x*y*z^2, x*y^2*z^2, x^2*z^2, x^2*y*z^2, x^2*y^2*z^2 ]
```

66.17.8 MonomialGrlexOrdering

▷ `MonomialGrlexOrdering(`*[vari]*`)` (function)

This function creates a degree/lexicographic ordering. In this ordering monomials are compared first by their total degree, then lexicographically (see `MonomialLexOrdering` (66.17.7)).

The variables are ordered according to their (internal) index, i.e., x_1 is larger than x_2 and so on. If *vari* is given, and is a list of variables or variable indices, instead this arrangement of variables (in descending order; i.e. the first variable is larger than the second) is used as the underlying order of variables.

66.17.9 MonomialGrevlexOrdering

▷ `MonomialGrevlexOrdering(`*[vari]*`)` (function)

This function creates a "grevlex" ordering. In this ordering monomials are compared first by total degree and then backwards lexicographically. (This is different than "grlex" ordering with variables reversed.)

The variables are ordered according to their (internal) index, i.e., x_1 is larger than x_2 and so on. If *vari* is given, and is a list of variables or variable indices, instead this arrangement of variables (in descending order; i.e. the first variable is larger than the second) is used as the underlying order of variables.

Example

```
gap> Sort(l,MonomialComparisonFunction(MonomialGrlexOrdering()));l;
[ 1, z, y, x, z^2, y*z, y^2, x*z, x*y, x^2, y*z^2, y^2*z, x*z^2,
  x*y*z, x*y^2, x^2*z, x^2*y, y^2*z^2, x*y*z^2, x*y^2*z, x^2*z^2,
  x^2*y*z, x^2*y^2, x*y^2*z^2, x^2*y*z^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialGrevlexOrdering()));l;
```

```
[ 1, z, y, x, z^2, y*z, x*z, y^2, x*y, x^2, y*z^2, x*z^2, y^2*z,
  x*y*z, x^2*z, x*y^2, x^2*y, y^2*z^2, x*y*z^2, x^2*z^2, x*y^2*z,
  x^2*y*z, x^2*y^2, x*y^2*z^2, x^2*y*z^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialGrlexOrdering([z,y,x])));l;
[ 1, x, y, z, x^2, x*y, y^2, x*z, y*z, z^2, x^2*y, x*y^2, x^2*z,
  x*y*z, y^2*z, x*z^2, y*z^2, x^2*y^2, x^2*y*z, x*y^2*z, x^2*z^2,
  x*y*z^2, y^2*z^2, x^2*y^2*z, x^2*y*z^2, x*y^2*z^2, x^2*y^2*z^2 ]
```

66.17.10 EliminationOrdering

▷ `EliminationOrdering(`*elim[, rest]*`)` (function)

This function creates an elimination ordering for eliminating the variables in *elim*. Two monomials are compared first by the exponent vectors for the variables listed in *elim* (a lexicographic comparison with respect to the ordering indicated in *elim*). If these submonomial are equal, the submonomials given by the other variables are compared by a graded lexicographic ordering (with respect to the variable order given in *rest*, if called with two parameters).

Both *elim* and *rest* may be a list of variables or a list of variable indices.

66.17.11 PolynomialReduction

▷ `PolynomialReduction(`*poly, gens, order*`)` (function)

reduces the polynomial *poly* by the ideal generated by the polynomials in *gens*, using the order *order* of monomials. Unless *gens* is a Gröbner basis the result is not guaranteed to be unique.

The operation returns a list of length two, the first entry is the remainder after the reduction. The second entry is a list of quotients corresponding to *gens*.

Note that the strategy used by `PolynomialReduction` differs from the standard textbook reduction algorithm, which is provided by `PolynomialDivisionAlgorithm` (66.17.13).

66.17.12 PolynomialReducedRemainder

▷ `PolynomialReducedRemainder(`*poly, gens, order*`)` (function)

this operation does the same way as `PolynomialReduction` (66.17.11) but does not keep track of the actual quotients and returns only the remainder (it is therefore slightly faster).

66.17.13 PolynomialDivisionAlgorithm

▷ `PolynomialDivisionAlgorithm(`*poly, gens, order*`)` (function)

This function implements the division algorithm for multivariate polynomials as given in [CLO97, Theorem 3 in Chapter 2]. (It might be slower than `PolynomialReduction` (66.17.11) but the remainders are guaranteed to agree with the textbook.)

The operation returns a list of length two, the first entry is the remainder after the reduction. The second entry is a list of quotients corresponding to *gens*.

Example

```
gap> bas:=[x^3*y*z,x*y^2*z,z*y*z^3+x];;
gap> pol:=x^7*z*bas[1]+y^5*bas[3]+x*z;;
gap> PolynomialReduction(pol,bas,MonomialLexOrdering());
[ -y*z^5, [ x^7*z, 0, y^5+z ] ]
gap> PolynomialReducedRemainder(pol,bas,MonomialLexOrdering());
-y*z^5
gap> PolynomialDivisionAlgorithm(pol,bas,MonomialLexOrdering());
[ -y*z^5, [ x^7*z, 0, y^5+z ] ]
```

66.17.14 MonomialExtGrlexLess

▷ `MonomialExtGrlexLess(a, b)` (function)

implements comparison of monomial in their external representation by a "grlex" order with $x_1 > x_2$ (This is exactly the same as the ordering by `MonomialGrlexOrdering` (66.17.8), see 66.17). The function takes two monomials `a` and `b` in expanded form and returns whether the first is smaller than the second. (This ordering is also used by GAP internally for representing polynomials as a linear combination of monomials.)

See section 66.21 for details on the expanded form of monomials.

66.18 Groebner Bases

A *Groebner Basis* of an ideal *I*i, in a polynomial ring *R*, with respect to a monomial ordering, is a set of ideal generators `G` such that the ideal generated by the leading monomials of all polynomials in `G` is equal to the ideal generated by the leading monomials of all polynomials in `I`.

For more details on Groebner bases see [CLO97].

66.18.1 GroebnerBasis

▷ `GroebnerBasis(L, O)` (operation)
▷ `GroebnerBasis(I, O)` (operation)
▷ `GroebnerBasisNC(L, O)` (operation)

Let `O` be a monomial ordering and `L` be a list of polynomials that generate an ideal `I`. This operation returns a Groebner basis of `I` with respect to the ordering `O`.

`GroebnerBasisNC` works like `GroebnerBasis` with the only distinction that the first argument has to be a list of polynomials and that no test is performed to check whether the ordering is defined for all occuring variables.

Note that GAP at the moment only includes a naïve implementation of Buchberger's algorithm (which is mainly intended as a teaching tool). It might not be sufficient for serious problems.

Example

```
gap> l:=[x^2+y^2+z^2-1,x^2+z^2-y,x-y];;
gap> GroebnerBasis(l,MonomialLexOrdering());
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1, -z^2+2*y-1,
  1/2*z^4+2*z^2-1/2 ]
gap> GroebnerBasis(l,MonomialLexOrdering([z,x,y]));
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1 ]
```

```
gap> GroebnerBasis(l,MonomialGrlexOrdering());
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1, -z^2+2*y-1 ]
```

66.18.2 ReducedGroebnerBasis

▷ `ReducedGroebnerBasis(L, O)` (operation)
▷ `ReducedGroebnerBasis(I, O)` (operation)

a Groebner basis B (see `GroebnerBasis` (66.18.1)) is *reduced* if no monomial in a polynomial in `B` is divisible by the leading monomial of another polynomial in B. This operation computes a Groebner basis with respect to the monomial ordering `O` and then reduces it.

Example

```
gap> ReducedGroebnerBasis(l,MonomialGrlexOrdering());
[ x-y, z^2-2*y+1, y^2+y-1 ]
gap> ReducedGroebnerBasis(l,MonomialLexOrdering());
[ z^4+4*z^2-1, -1/2*z^2+y-1/2, -1/2*z^2+x-1/2 ]
gap> ReducedGroebnerBasis(l,MonomialLexOrdering([y,z,x]));
[ x^2+x-1, z^2-2*x+1, -x+y ]
```

For performance reasons it can be advantageous to define monomial orderings once and then to reuse them:

Example

```
gap> ord:=MonomialGrlexOrdering();;
gap> GroebnerBasis(l,ord);
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1, -z^2+2*y-1 ]
gap> ReducedGroebnerBasis(l,ord);
[ x-y, z^2-2*y+1, y^2+y-1 ]
```

66.18.3 StoredGroebnerBasis

▷ `StoredGroebnerBasis(I)` (attribute)

For an ideal `I` in a polynomial ring, this attribute holds a list $[B,O]$ where B is a Groebner basis for the monomial ordering O. this can be used to test membership or canonical coset representatives.

66.18.4 InfoGroebner

▷ `InfoGroebner` (info class)

This info class gives information about Groebner basis calculations.

66.19 Rational Function Families

All rational functions defined over a ring lie in the same family, the rational functions family over this ring.

In GAP therefore the family of a polynomial depends only on the family of the coefficients, all polynomials whose coefficients lie in the same family are “compatible”.

66.19.1 RationalFunctionsFamily

▷ RationalFunctionsFamily(*fam*) (attribute)

creates a family containing rational functions with coefficients in *fam*. All elements of the RationalFunctionsFamily are rational functions (see IsRationalFunction (66.4.1)).

66.19.2 IsPolynomialFunctionsFamily

▷ IsPolynomialFunctionsFamily(*obj*) (Category)
▷ IsRationalFunctionsFamily(*obj*) (Category)

IsPolynomialFunctionsFamily is the category of a family of polynomials. For families over an UFD, the category becomes IsRationalFunctionsFamily (as rational functions and quotients are only provided for families over an UFD.)

Example

```
gap> fam:=RationalFunctionsFamily(FamilyObj(1));
NewFamily( "RationalFunctionsFamily(...)", [ 618, 620 ],
[ 82, 85, 89, 93, 97, 100, 103, 107, 111, 618, 620 ] )
```

66.19.3 CoefficientsFamily

▷ CoefficientsFamily(*rffam*) (attribute)

If *rffam* has been created as RationalFunctionsFamily(*cfam*) this attribute holds the coefficients family *cfam*.

GAP does *not* embed the base ring in the polynomial ring. While multiplication and addition of base ring elements to rational functions return the expected results, polynomials and rational functions are not equal.

Example

```
gap> 1=Indeterminate(Rationals)^0;
false
```

66.20 The Representations of Rational Functions

GAP uses four representations of rational functions: Rational functions given by numerator and denominator, polynomials, univariate rational functions (given by coefficient lists for numerator and denominator and valuation) and Laurent polynomials (given by coefficient list and valuation).

These representations do not necessarily reflect mathematical properties: While an object in the Laurent polynomials representation must be a Laurent polynomial it might turn out that a rational function given by numerator and denominator is actually a Laurent polynomial and the property tests in section 66.4 will find this out.

Each representation is associated one or several "defining attributes" that give an "external" representation (see 66.21) of the representation in the form of lists and are the defining information that tells a rational function what it is.

GAP also implements methods to compute these attributes for rational functions in *other* representations, provided it would be possible to express an *mathematically equal* rational function in the

representation associated with the attribute. (That is one can always get a numerator/denominator representation of a polynomial while an arbitrary function of course can compute a polynomial representation only if it is a polynomial.)

Therefore these attributes can be thought of as "conceptual" representations that allow us –as far as possible– to consider an object as a rational function, a polynomial or a Laurent polynomial, regardless of the way it is represented in the computer.

Functions thus usually do not need to care about the representation of a rational function. Depending on its (known in the context or determined) properties, they can access the attribute representing the rational function in the desired way.

Consequentially, methods for rational functions are installed for properties and not for representations.

When *creating* new rational functions however they must be created in one of the three representations. In most cases this will be the representation for which the "conceptual" representation in which the calculation was done is the defining attribute.

Iterated operations (like forming the product over a list) therefore will tend to stay in the most suitable representation and the calculation of another conceptual representation (which may be comparatively expensive in certain circumstances) is not necessary.

66.21 The Defining Attributes of Rational Functions

In general, rational functions are given in terms of monomials. They are represented by lists, using numbers (see 66.1) for the indeterminates.

A monomial is a product of powers of indeterminates. A monomial is stored as a list (we call this the *expanded form* of the monomial) of the form `[`*inum*`,`*exp*`,`*inum*`,`*exp*`,...]` where each *inum* is the number of an indeterminate and *exp* the corresponding exponent. The list must be sorted according to the numbers of the indeterminates. Thus for example, if x, y and z are the first three indeterminates, the expanded form of the monomial $x^5z^8 = z^8x^5$ is `[ 1, 5, 3, 8 ]`. The representation of a polynomials is a list of the form `[`*mon*`,`*coeff*`,`*mon*`,`*coeff*`,...]` where *mon* is a monomial in expanded form (that is given as list) and *coeff* its coefficient. The monomials must be sorted according to the total degree/lexicographic order (This is the same as given by the "grlex" monomial ordering, see `MonomialGrlexOrdering` (66.17.8)). We call this the *external representation* of a polynomial. (The reason for ordering is that addition of polynomials becomes linear in the number of monomials instead of quadratic; the reason for the particular ordering chose is that it is compatible with multiplication and thus gives acceptable performance for quotient calculations.)

The attributes that give a representation of a a rational function as a Laurent polynomial are `CoefficientsOfLaurentPolynomial` (66.13.2) and `IndeterminateNumberOfUnivariateRationalFunction` (66.1.2).

Algorithms should use only the attributes `ExtRepNumeratorRatFun` (66.21.2), `ExtRepDenominatorRatFun` (66.21.3), `ExtRepPolynomialRatFun` (66.21.6), `CoefficientsOfLaurentPolynomial` (66.13.2) and –if the univariate function is not constant– `IndeterminateNumberOfUnivariateRationalFunction` (66.1.2) as the low-level interface to work with a polynomial. They should not refer to the actual representation used.

66.21.1 IsRationalFunctionDefaultRep

▷ IsRationalFunctionDefaultRep(*obj*) (Representation)

is the default representation of rational functions. A rational function in this representation is defined by the attributes ExtRepNumeratorRatFun (66.21.2) and ExtRepDenominatorRatFun (66.21.3), the values of which are external representations of polynomials.

66.21.2 ExtRepNumeratorRatFun

▷ ExtRepNumeratorRatFun(*ratfun*) (attribute)

returns the external representation of the numerator polynomial of the rational function *ratfun*. Numerator and denominator are not guaranteed to be cancelled against each other.

66.21.3 ExtRepDenominatorRatFun

▷ ExtRepDenominatorRatFun(*ratfun*) (attribute)

returns the external representation of the denominator polynomial of the rational function *ratfun*. Numerator and denominator are not guaranteed to be cancelled against each other.

66.21.4 ZeroCoefficientRatFun

▷ ZeroCoefficientRatFun(*ratfun*) (operation)

returns the zero of the coefficient ring. This might be needed to represent the zero polynomial for which the external representation of the numerator is the empty list.

66.21.5 IsPolynomialDefaultRep

▷ IsPolynomialDefaultRep(*obj*) (Representation)

is the default representation of polynomials. A polynomial in this representation is defined by the components and ExtRepNumeratorRatFun (66.21.2) where ExtRepNumeratorRatFun (66.21.2) is the external representation of the polynomial.

66.21.6 ExtRepPolynomialRatFun

▷ ExtRepPolynomialRatFun(*polynomial*) (attribute)

returns the external representation of a polynomial. The difference to ExtRepNumeratorRatFun (66.21.2) is that rational functions might know to be a polynomial but can still have a non-vanishing denominator. In this case ExtRepPolynomialRatFun has to call a quotient routine.

66.21.7 IsLaurentPolynomialDefaultRep

▷ IsLaurentPolynomialDefaultRep(*obj*) (Representation)

This representation is used for Laurent polynomials and univariate polynomials. It represents a Laurent polynomial via the attributes CoefficientsOfLaurentPolynomial (66.13.2) and IndeterminateNumberOfLaurentPolynomial (66.13.3).

66.22 Creation of Rational Functions

The operations LaurentPolynomialByCoefficients (66.13.1), PolynomialByExtRep (66.22.2) and RationalFunctionByExtRep (66.22.1) are used to construct objects in the three basic representations for rational functions.

66.22.1 RationalFunctionByExtRep

▷ RationalFunctionByExtRep(*rfam, num, den*) (function)
▷ RationalFunctionByExtRepNC(*rfam, num, den*) (function)

constructs a rational function (in the representation IsRationalFunctionDefaultRep (66.21.1)) in the rational function family *rfam*, the rational function itself is given by the external representations *num* and *den* for numerator and denominator. No cancellation takes place.

The variant RationalFunctionByExtRepNC does not perform any test of the arguments and thus potentially can create illegal objects. It only should be used if speed is required and the arguments are known to be in correct form.

66.22.2 PolynomialByExtRep

▷ PolynomialByExtRep(*rfam, extrep*) (function)
▷ PolynomialByExtRepNC(*rfam, extrep*) (function)

constructs a polynomial (in the representation IsPolynomialDefaultRep (66.21.5)) in the rational function family *rfam*, the polynomial itself is given by the external representation *extrep*.

The variant PolynomialByExtRepNC does not perform any test of the arguments and thus potentially can create invalid objects. It only should be used if speed is required and the arguments are known to be in correct form.

Example

```
gap> fam:=RationalFunctionsFamily(FamilyObj(1));;
gap> p:=PolynomialByExtRep(fam,[[1,2],1,[2,1,15,7],3]);
3*y*x_15^7+x^2
gap> q:=p/(p+1);
(3*y*x_15^7+x^2)/(3*y*x_15^7+x^2+1)
gap> ExtRepNumeratorRatFun(q);
[ [ 1, 2 ], 1, [ 2, 1, 15, 7 ], 3 ]
gap> ExtRepDenominatorRatFun(q);
[ [  ], 1, [ 1, 2 ], 1, [ 2, 1, 15, 7 ], 3 ]
```

66.22.3 LaurentPolynomialByExtRep

▷ LaurentPolynomialByExtRep(*fam, cofs, val, ind*) (function)
▷ LaurentPolynomialByExtRepNC(*fam, cofs, val, ind*) (function)

creates a Laurent polynomial in the family *fam* with [*cofs*,*val*] as value of CoefficientsOfLaurentPolynomial (66.13.2). No coefficient shifting is performed. This is the lowest level function to create a Laurent polynomial but will rely on the coefficients being shifted properly and will not perform any tests. Unless this is guaranteed for the parameters, LaurentPolynomialByCoefficients (66.13.1) should be used.

66.23 Arithmetic for External Representations of Polynomials

The following operations are used internally to perform the arithmetic for polynomials in their "external" representation (see 66.21) as lists.

Functions to perform arithmetic with the coefficient lists of Laurent polynomials are described in Section 23.4.

66.23.1 ZippedSum

▷ ZippedSum(*z1, z2, czero, funcs*) (operation)

computes the sum of two external representations of polynomials *z1* and *z2*. *czero* is the appropriate coefficient zero and *funcs* a list [*monomial_less*, *coefficient_sum*] containing a monomial comparison and a coefficient addition function. This list can be found in the component *fam*!.zippedSum of the rational functions family.

Note that *coefficient_sum* must be a proper "summation" function, not a function computing differences.

66.23.2 ZippedProduct

▷ ZippedProduct(*z1, z2, czero, funcs*) (operation)

computes the product of two external representations of polynomials *z1* and *z2*. *czero* is the appropriate coefficient zero and *funcs* a list [*monomial_prod*, *monomial_less*, *coefficient_sum*, *coefficient_prod*] containing functions to multiply and compare monomials, to add and to multiply coefficients. This list can be found in the component *fam*!.zippedProduct of the rational functions family.

66.23.3 QuotientPolynomialsExtRep

▷ QuotientPolynomialsExtRep(*fam, a, b*) (function)

Let a and *b* the external representations of two polynomials in the rational functions family *fam*. This function computes the external representation of the quotient of both polynomials, it returns fail if the polynomial described by *b* does not divide the polynomial described by a.

66.24 Cancellation Tests for Rational Functions

The operation Gcd (56.7.1) can be used to test for common factors of two polynomials. This however would be too expensive to be done in the arithmetic, thus uses the following operations internally to try to keep the denominators as small as possible

66.24.1 RationalFunctionByExtRepWithCancellation

▷ RationalFunctionByExtRepWithCancellation(*rfam, num, den*) (function)

constructs a rational function as RationalFunctionByExtRep (66.22.1) does but tries to cancel out common factors of numerator and denominator, calling TryGcdCancelExtRepPolynomials (66.24.2).

66.24.2 TryGcdCancelExtRepPolynomials

▷ TryGcdCancelExtRepPolynomials(*fam, a, b*) (function)

Let a and b be the external representations of two polynomials. This function tries to cancel common factors between the corresponding polynomials and returns a list $[a', b']$ of external representations of cancelled polynomials. As there is no proper multivariate GCD cancellation is not guaranteed to be optimal.

66.24.3 HeuristicCancelPolynomials

▷ HeuristicCancelPolynomials(*fam, ext1, ext2*) (operation)

is called by TryGcdCancelExtRepPolynomials (66.24.2) to perform the actual work. It will return either fail or a new list of of external representations of cancelled polynomials. The cancellation performed is not necessarily optimal.

Chapter 67

Algebraic extensions of fields

If we adjoin a root α of an irreducible polynomial $f \in K[x]$ to the field K we get an *algebraic extension* $K(\alpha)$, which is again a field. We call K the *base field* of $K(\alpha)$.

By Kronecker's construction, we may identify $K(\alpha)$ with the factor ring $K[x]/(f)$, an identification that also provides a method for computing in these extension fields.

It is important to note that different extensions of the same field are entirely different (and its elements lie in different families), even if mathematically one could be embedded in the other one.

Currently GAP only allows extension fields of fields K, when K itself is not an extension field.

67.1 Creation of Algebraic Extensions

67.1.1 AlgebraicExtension

▷ `AlgebraicExtension(K, f)` (operation)

constructs an extension `L` of the field `K` by one root of the irreducible polynomial `f`, using Kronecker's construction. `L` is a field whose `LeftActingDomain` (57.1.11) value is `K`. The polynomial `f` is the `DefiningPolynomial` (58.2.7) value of `L` and the attribute `RootOfDefiningPolynomial` (58.2.8) of `L` holds a root of `f` in `L`.

Example

```
gap> x:=Indeterminate(Rationals,"x");;
gap> p:=x^4+3*x^2+1;;
gap> e:=AlgebraicExtension(Rationals,p);
<algebraic extension over the Rationals of degree 4>
gap> IsField(e);
true
gap> a:=RootOfDefiningPolynomial(e);
a
```

67.1.2 IsAlgebraicExtension

▷ `IsAlgebraicExtension(obj)` (Category)

is the category of algebraic extensions of fields.

Example

```
gap> IsAlgebraicExtension(e);
true
gap> IsAlgebraicExtension(Rationals);
false
```

67.2 Elements in Algebraic Extensions

According to Kronecker's construction, the elements of an algebraic extension are considered to be polynomials in the primitive element. The elements of the base field are represented as polynomials of degree zero. GAP therefore displays elements of an algebraic extension as polynomials in an indeterminate "a", which is a root of the defining polynomial of the extension. Polynomials of degree zero are displayed with a leading exclamation mark to indicate that they are different from elements of the base field.

The usual field operations are applicable to algebraic elements.

Example

```
gap> a^3/(a^2+a+1);
-1/2*a^3+1/2*a^2-1/2*a
gap> a*(1/a);
!1
```

The external representation of algebraic extension elements are the polynomial coefficients in the primitive element a, the operations ExtRepOfObj (79.16.1) and ObjByExtRep (79.16.1) can be used for conversion.

Example

```
gap> ExtRepOfObj(One(a));
[ 1, 0, 0, 0 ]
gap> ExtRepOfObj(a^3+2*a-9);
[ -9, 2, 0, 1 ]
gap> ObjByExtRep(FamilyObj(a),[3,19,-27,433]);
433*a^3-27*a^2+19*a+3
```

GAP does *not* embed the base field in its algebraic extensions and therefore lists which contain elements of the base field and of the extension are not homogeneous and thus cannot be used as polynomial coefficients or to form matrices. The remedy is to multiply the list(s) with the value of the attribute One (31.10.2) of the extension which will embed all entries in the extension.

Example

```
gap> m:=[[1,a],[0,1]];
[ [ 1, a ], [ 0, 1 ] ]
gap> IsMatrix(m);
false
gap> m:=m*One(e);
[ [ !1, a ], [ !0, !1 ] ]
gap> IsMatrix(m);
true
gap> m^2;
[ [ !1, 2*a ], [ !0, !1 ] ]
```

67.2.1 IsAlgebraicElement

▷ `IsAlgebraicElement(`*obj*`)` (Category)

is the category for elements of an algebraic extension.

Chapter 68

p-adic Numbers (preliminary)

In this chapter p is always a (fixed) prime integer.

The p-adic numbers Q_p are the completion of the rational numbers with respect to the valuation $v_p(p^v \cdot a/b) = v$ if p divides neither a nor b. They form a field of characteristic 0 which nevertheless shows some behaviour of the finite field with p elements.

A p-adic numbers can be represented by a "p-adic expansion" which is similar to the decimal expansion used for the reals (but written from left to right). So for example if $p = 2$, the numbers 1, 2, 3, 4, $1/2$, and $4/5$ are represented as $1(2)$, $0.1(2)$, $1.1(2)$, $0.01(2)$, $10(2)$, and the infinite periodic expansion $0.010110011001100...(2)$. p-adic numbers can be approximated by ignoring higher powers of p, so for example with only 2 digits accuracy $4/5$ would be approximated as $0.01(2)$. This is different from the decimal approximation of real numbers in that p-adic approximation is a ring homomorphism on the subrings of p-adic numbers whose valuation is bounded from below so that rounding errors do not increase with repeated calculations.

In GAP, p-adic numbers are always represented by such approximations. A family of approximated p-adic numbers consists of p-adic numbers with a fixed prime p and a certain precision, and arithmetic with these numbers is done with this precision.

68.1 Pure p-adic Numbers

Pure p-adic numbers are the p-adic numbers described so far.

68.1.1 PurePadicNumberFamily

▷ `PurePadicNumberFamily(`*`p, precision`*`)` (operation)

returns the family of pure p-adic numbers over the prime *p* with *precision* "digits". That is to say, the approximate value will differ from the correct value by a multiple of p^{digits}.

68.1.2 PadicNumber (for pure padics)

▷ `PadicNumber(`*`fam, rat`*`)` (function)

returns the element of the p-adic number family *fam* that approximates the rational number *rat*. p-adic numbers allow the usual operations for fields.

Example

```
gap> fam:=PurePadicNumberFamily(2,20);;
gap> a:=PadicNumber(fam,4/5);
0.01011001100110011(2)
gap> fam:=PurePadicNumberFamily(2,3);;
gap> a:=PadicNumber(fam,4/5);
0.0101(2)
gap> 3*a;
0.0111(2)
gap> a/2;
0.101(2)
gap> a*10;
0.001(2)
```

See PadicNumber (68.2.2) for other methods for PadicNumber.

68.1.3 Valuation

▷ Valuation(*obj*) (operation)

The valuation is the p-part of the p-adic number.

68.1.4 ShiftedPadicNumber

▷ ShiftedPadicNumber(*padic, int*) (operation)

ShiftedPadicNumber takes a p-adic number *padic* and an integer *shift* and returns the p-adic number c, that is *padic* * *p^shift*.

68.1.5 IsPurePadicNumber

▷ IsPurePadicNumber(*obj*) (Category)

The category of pure p-adic numbers.

68.1.6 IsPurePadicNumberFamily

▷ IsPurePadicNumberFamily(*fam*) (Category)

The family of pure p-adic numbers.

68.2 Extensions of the p-adic Numbers

The usual Kronecker construction with an irreducible polynomial can be used to construct extensions of the p-adic numbers. Let L be such an extension. Then there is a subfield $K < L$ such that K is an unramified extension of the p-adic numbers and L/K is purely ramified.

(For an explanation of "ramification" see for example [Neu92, Section II.7], or another book on algebraic number theory. Essentially, an extension L of the p-adic numbers generated by a rational

polynomial f is unramified if f remains squarefree modulo p and is completely ramified if modulo p the polynomial f is a power of a linear factor while remaining irreducible over the p-adic numbers.)

The representation of extensions of p-adic numbers in GAP uses the subfield K.

68.2.1 PadicExtensionNumberFamily

▷ `PadicExtensionNumberFamily(p, precision, unram, ram)` (function)

An extended p-adic field L is given by two polynomials h and g with coefficient lists `unram` (for the unramified part) and `ram` (for the ramified part). Then L is isomorphic to $Q_p[x,y]/(h(x),g(y))$.

This function takes the prime number `p` and the two coefficient lists `unram` and `ram` for the two polynomials. The polynomial given by the coefficients in `unram` must be a cyclotomic polynomial and the polynomial given by `ram` must be either an Eisenstein polynomial or $1+x$. *This is not checked by GAP.*

Every number in L is represented as a coefficient list w. r. t. the basis $\{1,x,x^2,\ldots,y,xy,x^2y,\ldots\}$ of L. The integer `precision` is the number of "digits" that all the coefficients have.

A general comment:

The polynomials with which `PadicExtensionNumberFamily` is called define an extension of Q_p. It must be ensured that both polynomials are really irreducible over Q_p! For example x^2+x+1 is *not* irreducible over Q_p. Therefore the "extension" `PadicExtensionNumberFamily(3, 4, [1,1,1], [1,1])` contains non-invertible "pseudo-p-adic numbers". Conversely, if an "extension" contains noninvertible elements then one of the defining polynomials was not irreducible.

68.2.2 PadicNumber (for a p-adic extension family and a rational)

▷ `PadicNumber(fam, rat)` (operation)
▷ `PadicNumber(purefam, list)` (operation)
▷ `PadicNumber(extfam, list)` (operation)

(see also `PadicNumber` (68.1.2)).

`PadicNumber` creates a p-adic number in the p-adic numbers family `fam`. The first form returns the p-adic number corresponding to the rational `rat`.

The second form takes a pure p-adic numbers family `purefam` and a list `list` of length two, and returns the number `p^list[1] * list[2]`. It must be guaranteed that no entry of `list[2]` is divisible by the prime p. (Otherwise precision will get lost.)

The third form creates a number in the family `extfam` of a p-adic extension. The second argument must be a list `list` of length two such that `list[2]` is the list of coefficients w.r.t. the basis $\{1,\ldots,x^{f-1}\cdot y^{e-1}\}$ of the extended p-adic field and `list[1]` is a common p-part of all these coefficients.

p-adic numbers admit the usual field operations.

Example

```
gap> efam:=PadicExtensionNumberFamily(3, 5, [1,1,1], [1,1]);;
gap> PadicNumber(efam,7/9);
padic(120(3),0(3))
```

A word of warning:

Depending on the actual representation of quotients, precision may seem to "vanish". For example in `PadicExtensionNumberFamily(3, 5, [1,1,1], [1,1])` the number `(1.2000,`

`0.1210)(3)` can be represented as `[ 0, [ 1.2000, 0.1210 ] ]` or as `[ -1, [ 12.000, 1.2100 ] ]` (here the coefficients have to be multiplied by p^{-1}).

So there may be a number `(1.2, 2.2)(3)` which seems to have only two digits of precision instead of the declared 5. But internally the number is stored as `[ -3, [ 0.0012, 0.0022 ] ]` and so has in fact maximum precision.

68.2.3 IsPadicExtensionNumber

▷ `IsPadicExtensionNumber(`*obj*`)` (Category)

The category of elements of the extended p-adic field.

Example

```
gap>  efam:=PadicExtensionNumberFamily(3, 5, [1,1,1], [1,1]);;
gap> IsPadicExtensionNumber(PadicNumber(efam,7/9));
true
```

68.2.4 IsPadicExtensionNumberFamily

▷ `IsPadicExtensionNumberFamily(`*fam*`)` (Category)

Family of elements of the extended p-adic field.

Example

```
gap> efam:=PadicExtensionNumberFamily(3, 5, [1,1,1], [1,1]);;
gap> IsPadicExtensionNumberFamily(efam);
true
```

Chapter 69

The MeatAxe

The MeatAxe [Par84] is a tool for the examination of submodules of a group algebra. It is a basic tool for the examination of group actions on finite-dimensional modules.

GAP uses the improved MeatAxe of Derek Holt and Sarah Rees, and also incorporates further improvements of Ivanyos and Lux.

Please note that, consistently with the convention for group actions, the action of the GAP MeatAxe is always that of matrices on row vectors by multiplication on the right. If you want to investigate left modules you will have to transpose the matrices.

69.1 MeatAxe Modules

69.1.1 GModuleByMats

▷ `GModuleByMats(`*`gens, field`*`)` (function)
▷ `GModuleByMats(`*`emptygens, dim, field`*`)` (function)

creates a MeatAxe module over *`field`* from a list of invertible matrices *`gens`* which reflect a group's action. If the list of generators is empty, the dimension must be given as second argument.

MeatAxe routines are on a level with Gaussian elimination. Therefore they do not deal with GAP modules but essentially with lists of matrices. For the MeatAxe, a module is a record with components

`generators`
: A list of matrices which represent a group operation on a finite dimensional row vector space.

`dimension`
: The dimension of the vector space (this is the common length of the row vectors (see `DimensionOfVectors` (61.9.6))).

`field`
: The field over which the vector space is defined.

Once a module has been created its entries may not be changed. A MeatAxe may create a new component *`NameOfMeatAxe`* in which it can store private information. By a MeatAxe "submodule" or "factor module" we denote actually the *induced action* on the submodule, respectively factor module. Therefore the submodules or factor modules are again MeatAxe modules. The arrangement of `generators` is guaranteed to be the same for the induced modules, but to obtain the complete relation to the original module, the bases used are needed as well.

69.2 Module Constructions

69.2.1 PermutationGModule

▷ `PermutationGModule(`*G, F*`)` (function)

Called with a permutation group *G* and a finite field *F*, `PermutationGModule` returns the natural permutation module *M* over *F* for the group of permutation matrices that acts on the canonical basis of *M* in the same way as *G* acts on the points up to its largest moved point (see `LargestMovedPoint` (42.3.2)).

69.2.2 TensorProductGModule

▷ `TensorProductGModule(`*m1, m2*`)` (function)

`TensorProductGModule` calculates the tensor product of the modules *m1* and *m2*. They are assumed to be modules over the same algebra so, in particular, they should have the same number of generators.

69.2.3 WedgeGModule

▷ `WedgeGModule(`*module*`)` (function)

`WedgeGModule` calculates the wedge product of a *G*-module. That is the action on antisymmetric tensors.

69.3 Selecting a Different MeatAxe

69.3.1 MTX

▷ `MTX` (global variable)

All MeatAxe routines are accessed via the global variable `MTX`, which is a record whose components hold the various functions. It is possible to have several implementations of a MeatAxe available. Each MeatAxe represents its routines in an own global variable and assigning `MTX` to this variable selects the corresponding MeatAxe.

69.4 Accessing a Module

Even though a MeatAxe module is a record, its components should never be accessed outside of MeatAxe functions. Instead the following operations should be used:

69.4.1 MTX.Generators

▷ `MTX.Generators(`*module*`)` (function)

returns a list of matrix generators of *module*.

69.4.2 MTX.Dimension

▷ `MTX.Dimension(`*module*`)` (function)

returns the dimension in which the matrices act.

69.4.3 MTX.Field

▷ `MTX.Field(`*module*`)` (function)

returns the field over which *module* is defined.

69.5 Irreducibility Tests

69.5.1 MTX.IsIrreducible

▷ `MTX.IsIrreducible(`*module*`)` (function)

tests whether the module *module* is irreducible (i.e. contains no proper submodules.)

69.5.2 MTX.IsAbsolutelyIrreducible

▷ `MTX.IsAbsolutelyIrreducible(`*module*`)` (function)

A module is absolutely irreducible if it remains irreducible over the algebraic closure of the field. (Formally: If the tensor product $L \otimes_K M$ is irreducible where M is the module defined over K and L is the algebraic closure of K.)

69.5.3 MTX.DegreeSplittingField

▷ `MTX.DegreeSplittingField(`*module*`)` (function)

returns the degree of the splitting field as extension of the prime field.

69.6 Decomposition of modules

A module is *decomposable* if it can be written as the direct sum of two proper submodules (and *indecomposable* if not). Obviously every finite dimensional module is a direct sum of its indecomposable parts. The *homogeneous components* of a module are the direct sums of isomorphic indecomposable components. They are uniquely determined.

69.6.1 MTX.IsIndecomposable

▷ `MTX.IsIndecomposable(`*module*`)` (function)

returns whether *module* is indecomposable.

69.6.2 MTX.Indecomposition

▷ `MTX.Indecomposition(`*module*`)` (function)

returns a decomposition of *module* as a direct sum of indecomposable modules. It returns a list, each entry is a list of form [*B*,*ind*] where *B* is a list of basis vectors for the indecomposable component and *ind* the induced module action on this component. (Such a decomposition is not unique.)

69.6.3 MTX.HomogeneousComponents

▷ `MTX.HomogeneousComponents(`*module*`)` (function)

computes the homogeneous components of *module* given as sums of indecomposable components. The function returns a list, each entry of which is a record corresponding to one isomorphism type of indecomposable components. The record has the following components.

`indices`
: the index numbers of the indecomposable components, as given by `MTX.Indecomposition` (69.6.2), that are in the homogeneous component,

`component`
: one of the indecomposable components,

`images`
: a list of the remaining indecomposable components, each given as a record with the components `component` (the component itself) and `isomorphism` (an isomorphism from the defining component to this one).

69.7 Finding Submodules

69.7.1 MTX.SubmoduleGModule

▷ `MTX.SubmoduleGModule(`*module, subspace*`)` (function)
▷ `MTX.SubGModule(`*module, subspace*`)` (function)

subspace should be a subspace of (or a vector in) the underlying vector space of *module* i.e. the full row space of the same dimension and over the same field as *module*. A normalized basis of the submodule of *module* generated by *subspace* is returned.

69.7.2 MTX.ProperSubmoduleBasis

▷ `MTX.ProperSubmoduleBasis(`*module*`)` (function)

returns the basis of a proper submodule of *module* and `fail` if no proper submodule exists.

69.7.3 MTX.BasesSubmodules

▷ `MTX.BasesSubmodules(`*module*`)` (function)

returns a list containing a basis for every submodule.

69.7.4 MTX.BasesMinimalSubmodules

▷ `MTX.BasesMinimalSubmodules(`*`module`*`)` (function)

returns a list of bases of all minimal submodules.

69.7.5 MTX.BasesMaximalSubmodules

▷ `MTX.BasesMaximalSubmodules(`*`module`*`)` (function)

returns a list of bases of all maximal submodules.

69.7.6 MTX.BasisRadical

▷ `MTX.BasisRadical(`*`module`*`)` (function)

returns a basis of the radical of *`module`*.

69.7.7 MTX.BasisSocle

▷ `MTX.BasisSocle(`*`module`*`)` (function)

returns a basis of the socle of *`module`*.

69.7.8 MTX.BasesMinimalSupermodules

▷ `MTX.BasesMinimalSupermodules(`*`module, sub`*`)` (function)

returns a list of bases of all minimal supermodules of the submodule given by the basis *`sub`*.

69.7.9 MTX.BasesCompositionSeries

▷ `MTX.BasesCompositionSeries(`*`module`*`)` (function)

returns a list of bases of submodules in a composition series in ascending order.

69.7.10 MTX.CompositionFactors

▷ `MTX.CompositionFactors(`*`module`*`)` (function)

returns a list of composition factors of *`module`* in ascending order.

69.7.11 MTX.CollectedFactors

▷ `MTX.CollectedFactors(`*`module`*`)` (function)

returns a list giving all irreducible composition factors with their frequencies.

69.8 Induced Actions

69.8.1 MTX.NormedBasisAndBaseChange

▷ `MTX.NormedBasisAndBaseChange(`*sub*`)` (function)

returns a list `[`*bas*`, `*change*` ]` where *bas* is a normed basis (i.e. in echelon form with pivots normed to 1) for *sub* and *change* is the base change from *bas* to *sub* (the basis vectors of *bas* expressed in coefficients for *sub*).

69.8.2 MTX.InducedActionSubmodule

▷ `MTX.InducedActionSubmodule(`*module, sub*`)` (function)
▷ `MTX.InducedActionSubmoduleNB(`*module, sub*`)` (function)

creates a new module corresponding to the action of *module* on *sub*. In the `NB` version the basis *sub* must be normed. (That is it must be in echelon form with pivots normed to 1, see `MTX.NormedBasisAndBaseChange` (69.8.1).)

69.8.3 MTX.InducedActionFactorModule

▷ `MTX.InducedActionFactorModule(`*module, sub[, compl]*`)` (function)

creates a new module corresponding to the action of *module* on the factor of *sub*. If *compl* is given, it has to be a basis of a (vector space-)complement of *sub*. The action then will correspond to *compl*.

The basis *sub* has to be given in normed form. (That is it must be in echelon form with pivots normed to 1, see `MTX.NormedBasisAndBaseChange` (69.8.1))

69.8.4 MTX.InducedActionMatrix

▷ `MTX.InducedActionMatrix(`*mat, sub*`)` (function)
▷ `MTX.InducedActionMatrixNB(`*mat, sub*`)` (function)
▷ `MTX.InducedActionFactorMatrix(`*mat, sub[, compl]*`)` (function)

work the same way as the above functions for modules, but take as input only a single matrix.

69.8.5 MTX.InducedAction

▷ `MTX.InducedAction(`*module, sub[, type]*`)` (function)

Computes induced actions on submodules or factor modules and also returns the corresponding bases. The action taken is binary encoded in *type*: `1` stands for subspace action, `2` for factor action, and `4` for action of the full module on a subspace adapted basis. The routine returns the computed results in a list in sequence (*sub*,*quot*,*both*,*basis*) where *basis* is a basis for the whole space, extending *sub*. (Actions which are not computed are omitted, so the returned list may be shorter.) If no *type* is given, it is assumed to be `7`. The basis given in *sub* must be normed!

All these routines return `fail` if *sub* is not a proper subspace.

69.9 Module Homomorphisms

69.9.1 MTX.BasisModuleHomomorphisms

▷ `MTX.BasisModuleHomomorphisms(`*module1, module2*`)` (function)

returns a basis of all module homomorphisms from *module1* to *module2*. Homomorphisms are by matrices, whose rows give the images of the standard basis vectors of *module1* in the standard basis of *module2*.

69.9.2 MTX.BasisModuleEndomorphisms

▷ `MTX.BasisModuleEndomorphisms(`*module*`)` (function)

returns a basis of all module homomorphisms from *module* to *module*.

69.9.3 MTX.IsomorphismModules

▷ `MTX.IsomorphismModules(`*module1, module2*`)` (function)

If *module1* and *module2* are isomorphic modules, this function returns an isomorphism from *module1* to *module2* in form of a matrix. It returns `fail` if the modules are not isomorphic.

69.9.4 MTX.ModuleAutomorphisms

▷ `MTX.ModuleAutomorphisms(`*module*`)` (function)

returns the module automorphisms of *module* (the set of all isomorphisms from *module* to itself) as a matrix group.

69.10 Module Homomorphisms for irreducible modules

The following are lower-level functions that provide homomorphism functionality for irreducible modules. Generic code should use the functions in Section 69.9 instead.

69.10.1 MTX.IsEquivalent

▷ `MTX.IsEquivalent(`*module1, module2*`)` (function)

tests two irreducible modules for equivalence.

69.10.2 MTX.IsomorphismIrred

▷ `MTX.IsomorphismIrred(`*module1, module2*`)` (function)

returns an isomorphism from *module1* to *module2* (if one exists), and `fail` otherwise. It requires that one of the modules is known to be irreducible. It implicitly assumes that the same group is acting, otherwise the results are unpredictable. The isomorphism is given by a matrix M, whose rows give

the images of the standard basis vectors of *module1* in the standard basis of *module2*. That is, conjugation of the generators of *module2* with M yields the generators of *module1*.

69.10.3 MTX.Homomorphism

▷ MTX.Homomorphism(*module1, module2, mat*) (function)

mat should be a *dim1* $\times$ *dim2* matrix defining a homomorphism from *module1* to *module2*. This function verifies that *mat* really does define a module homomorphism, and then returns the corresponding homomorphism between the underlying row spaces of the modules. This can be used for computing kernels, images and pre-images.

69.10.4 MTX.Homomorphisms

▷ MTX.Homomorphisms(*module1, module2*) (function)

returns a basis of the space of all homomorphisms from the irreducible module *module1* to *module2*.

69.10.5 MTX.Distinguish

▷ MTX.Distinguish(*cf, nr*) (function)

Let *cf* be the output of MTX.CollectedFactors (69.7.11). This routine tries to find a group algebra element that has nullity zero on all composition factors except number *nr*.

69.11 MeatAxe Functionality for Invariant Forms

The functions in this section can only be applied to an absolutely irreducible MeatAxe module.

69.11.1 MTX.InvariantBilinearForm

▷ MTX.InvariantBilinearForm(*module*) (function)

returns an invariant bilinear form, which may be symmetric or anti-symmetric, of *module*, or fail if no such form exists.

69.11.2 MTX.InvariantSesquilinearForm

▷ MTX.InvariantSesquilinearForm(*module*) (function)

returns an invariant hermitian (= self-adjoint) sesquilinear form of *module*, which must be defined over a finite field whose order is a square, or fail if no such form exists.

69.11.3 MTX.InvariantQuadraticForm

▷ `MTX.InvariantQuadraticForm(`*`module`*`)` (function)

returns an invariant quadratic form of *`module`*, or `fail` if no such form exists. If the characteristic of the field over which *`module`* is defined is not 2, then the invariant bilinear form (if any) divided by two will be returned. In characteristic 2, the form returned will be lower triangular.

69.11.4 MTX.BasisInOrbit

▷ `MTX.BasisInOrbit(`*`module`*`)` (function)

returns a basis of the underlying vector space of *`module`* which is contained in an orbit of the action of the generators of module on that space. This is used by `MTX.InvariantQuadraticForm` (69.11.3) in characteristic 2.

69.11.5 MTX.OrthogonalSign

▷ `MTX.OrthogonalSign(`*`module`*`)` (function)

for an even dimensional module, returns 1 or -1, according as `MTX.InvariantQuadraticForm(`*`module`*`)` is of + or - type. For an odd dimensional module, returns 0. For a module with no invariant quadratic form, returns `fail`. This calculation uses an algorithm due to Jon Thackray.

69.12 The Smash MeatAxe

The standard MeatAxe provided in the GAP library is based on the MeatAxe in the GAP 3 package Smash, originally written by Derek Holt and Sarah Rees [HR94]. It is accessible via the variable `SMTX` to which `MTX` (69.3.1) is assigned by default. For the sake of completeness the remaining sections document more technical functions of this MeatAxe.

69.12.1 SMTX.RandomIrreducibleSubGModule

▷ `SMTX.RandomIrreducibleSubGModule(`*`module`*`)` (function)

returns the module action on a random irreducible submodule.

69.12.2 SMTX.GoodElementGModule

▷ `SMTX.GoodElementGModule(`*`module`*`)` (function)

finds an element with minimal possible nullspace dimension if *`module`* is known to be irreducible.

69.12.3 SMTX.SortHomGModule

▷ `SMTX.SortHomGModule(`*`module1, module2, homs`*`)` (function)

Function to sort the output of Homomorphisms.

69.12.4 SMTX.MinimalSubGModules

▷ SMTX.MinimalSubGModules(*module1, module2[, max]*) (function)

returns (at most *max*) bases of submodules of *module2* which are isomorphic to the irreducible module *module1*.

69.12.5 SMTX.Setter

▷ SMTX.Setter(*string*) (function)

returns a setter function for the component smashMeataxe.(string).

69.12.6 SMTX.Getter

▷ SMTX.Getter(*string*) (function)

returns a getter function for the component smashMeataxe.(string).

69.12.7 SMTX.IrreducibilityTest

▷ SMTX.IrreducibilityTest(*module*) (function)

Tests for irreducibility and sets a subbasis if reducible. It neither sets an irreducibility flag, nor tests it. Thus the routine also can simply be called to obtain a random submodule.

69.12.8 SMTX.AbsoluteIrreducibilityTest

▷ SMTX.AbsoluteIrreducibilityTest(*module*) (function)

Tests for absolute irreducibility and sets splitting field degree. It neither sets an absolute irreducibility flag, nor tests it.

69.12.9 SMTX.MinimalSubGModule

▷ SMTX.MinimalSubGModule(*module, cf, nr*) (function)

returns the basis of a minimal submodule of *module* containing the indicated composition factor. It assumes Distinguish has been called already.

69.12.10 SMTX.MatrixSum

▷ SMTX.MatrixSum(*matrices1, matrices2*) (function)

creates the direct sum of two matrix lists.

69.12.11 SMTX.CompleteBasis

▷ `SMTX.CompleteBasis(`*`module, pbasis`*`)` (function)

extends the partial basis *pbasis* to a basis of the full space by action of *module*. It returns whether it succeeded.

69.13 Smash MeatAxe Flags

The following getter routines access internal flags. For each routine, the appropriate setter's name is prefixed with `Set`.

69.13.1 SMTX.Subbasis

▷ `SMTX.Subbasis(`*`module`*`)` (function)

Basis of a submodule.

69.13.2 SMTX.AlgEl

▷ `SMTX.AlgEl(`*`module`*`)` (function)

list `[newgens,coefflist]` giving an algebra element used for chopping.

69.13.3 SMTX.AlgElMat

▷ `SMTX.AlgElMat(`*`module`*`)` (function)

matrix of `SMTX.AlgEl` (69.13.2).

69.13.4 SMTX.AlgElCharPol

▷ `SMTX.AlgElCharPol(`*`module`*`)` (function)

minimal polynomial of `SMTX.AlgEl` (69.13.2).

69.13.5 SMTX.AlgElCharPolFac

▷ `SMTX.AlgElCharPolFac(`*`module`*`)` (function)

uses factor of `SMTX.AlgEl` (69.13.2).

69.13.6 SMTX.AlgElNullspaceVec

▷ `SMTX.AlgElNullspaceVec(`*`module`*`)` (function)

nullspace of the matrix evaluated under this factor.

69.13.7 SMTX.AlgElNullspaceDimension

▷ `SMTX.AlgElNullspaceDimension(`*module*`)` (function)

dimension of the nullspace.

69.13.8 SMTX.CentMat

▷ `SMTX.CentMat(`*module*`)` (function)

matrix centralising all generators which is computed as a byproduct of `SMTX.AbsoluteIrreducibilityTest` (69.12.8).

69.13.9 SMTX.CentMatMinPoly

▷ `SMTX.CentMatMinPoly(`*module*`)` (function)

minimal polynomial of `SMTX.CentMat` (69.13.8).

Chapter 70

Tables of Marks

The concept of a *table of marks* was introduced by W. Burnside in his book "Theory of Groups of Finite Order", see [Bur55]. Therefore a table of marks is sometimes called a *Burnside matrix*.

The table of marks of a finite group G is a matrix whose rows and columns are labelled by the conjugacy classes of subgroups of G and where for two subgroups A and B the (A,B)-entry is the number of fixed points of B in the transitive action of G on the cosets of A in G. So the table of marks characterizes the set of all permutation representations of G.

Moreover, the table of marks gives a compact description of the subgroup lattice of G, since from the numbers of fixed points the numbers of conjugates of a subgroup B contained in a subgroup A can be derived.

A table of marks of a given group G can be constructed from the subgroup lattice of G (see 70.3). For several groups, the table of marks can be restored from the GAP library of tables of marks (see 70.13).

Given the table of marks of G, one can display it (see 70.4) and derive information about G and its Burnside ring from it (see 70.7, 70.8, 70.9). Moreover, tables of marks in GAP provide an easy access to the classes of subgroups of their underlying groups (see 70.10).

70.1 More about Tables of Marks

Let G be a finite group with n conjugacy classes of subgroups $C_1, C_2, \ldots, C_n$ and representatives $H_i \in C_i$, $1 \leq i \leq n$. The *table of marks* of G is defined to be the $n \times n$ matrix $M = (m_{ij})$ where the *mark* m_{ij} is the number of fixed points of the subgroup H_j in the action of G on the right cosets of H_i in G.

Since H_j can only have fixed points if it is contained in a point stabilizer the matrix M is lower triangular if the classes C_i are sorted according to the condition that if H_i is contained in a conjugate of H_j then $i \leq j$.

Moreover, the diagonal entries m_{ii} are nonzero since m_{ii} equals the index of H_i in its normalizer in G. Hence M is invertible. Since any transitive action of G is equivalent to an action on the cosets of a subgroup of G, one sees that the table of marks completely characterizes the set of all permutation representations of G.

The marks m_{ij} have further meanings. If H_1 is the trivial subgroup of G then each mark m_{i1} in the first column of M is equal to the index of H_i in G since the trivial subgroup fixes all cosets of H_i. If $H_n = G$ then each m_{nj} in the last row of M is equal to 1 since there is only one coset of G in G. In general, m_{ij} equals the number of conjugates of H_i containing H_j, multiplied by the index of H_i in its normalizer in G. Moreover, the number c_{ij} of conjugates of H_j which are contained in H_i can be

derived from the marks m_{ij} via the formula

$$c_{ij} = (m_{ij}m_{j1})/(m_{i1}m_{jj})$$

.

Both the marks m_{ij} and the numbers of subgroups c_{ij} are needed for the functions described in this chapter.

A brief survey of properties of tables of marks and a description of algorithms for the interactive construction of tables of marks using GAP can be found in [Pfe97].

70.2 Table of Marks Objects in GAP

A table of marks of a group G in GAP is represented by an immutable (see 12.6) object *tom* in the category IsTableOfMarks (70.6.2), with defining attributes SubsTom (70.7.1) and MarksTom (70.7.1). These two attributes encode the matrix of marks in a compressed form. The SubsTom (70.7.1) value of *tom* is a list where for each conjugacy class of subgroups the class numbers of its subgroups are stored. These are exactly the positions in the corresponding row of the matrix of marks which have nonzero entries. The marks themselves are stored via the MarksTom (70.7.1) value of *tom*, which is a list that contains for each entry in SubsTom(*tom*) the corresponding nonzero value of the table of marks.

It is possible to create table of marks objects that do not store a group, moreover one can create a table of marks object from a matrix of marks (see TableOfMarks (70.3.1)). So it may happen that a table of marks object in GAP is in fact *not* the table of marks of a group. To some extent, the consistency of a table of marks object can be checked (see 70.9), but GAP knows no general way to prove or disprove that a given matrix of nonnegative integers is the matrix of marks for a group. Many functions for tables of marks work well without access to the group –this is one of the arguments why tables of marks are so useful–, but for example normalizers (see NormalizerTom (70.9.4)) and derived subgroups (see DerivedSubgroupTom (70.9.2)) of subgroups are in general not uniquely determined by the matrix of marks.

GAP tables of marks are assumed to be in lower triangular form, that is, if a subgroup from the conjugacy class corresponding to the i-th row is contained in a subgroup from the class corresponding to the j-th row j then $i \leq j$.

The MarksTom (70.7.1) information can be computed from the values of the attributes NrSubsTom (70.7.2), LengthsTom (70.7.3), OrdersTom (70.7.2), and SubsTom (70.7.1). NrSubsTom (70.7.2) stores a list containing for each entry in the SubsTom (70.7.1) value the corresponding number of conjugates that are contained in a subgroup, LengthsTom (70.7.3) a list containing for each conjugacy class of subgroups its length, and OrdersTom (70.7.2) a list containing for each class of subgroups their order. So the MarksTom (70.7.1) value of *tom* may be missing provided that the values of NrSubsTom (70.7.2), LengthsTom (70.7.3), and OrdersTom (70.7.2) are stored in *tom*.

Additional information about a table of marks is needed by some functions. The class numbers of normalizers in G and the number of the derived subgroup of G can be stored via appropriate attributes (see NormalizersTom (70.9.4), DerivedSubgroupTom (70.9.2)).

If *tom* stores its group G and a bijection from the rows and columns of the matrix of marks of *tom* to the classes of subgroups of G then clearly normalizers, derived subgroup etc. can be computed from this information. But in general a table of marks need not have access to G, for example *tom* might have been constructed from a generic table of marks (see 70.12), or as table of marks of a factor group from a given table of marks (see FactorGroupTom (70.9.11)). Access to the group G is provided by

the attribute UnderlyingGroup (70.7.7) if this value is set. Access to the relevant information about conjugacy classes of subgroups of G –compatible with the ordering of rows and columns of the marks in *tom*– is signalled by the filter IsTableOfMarksWithGens (70.10.3).

Several examples in this chapter require the GAP package TomLib (the GAP Library of Tables of Marks) to be available. If it is not yet loaded then we load it now.

Example

```
gap> LoadPackage( "tomlib" );
true
```

70.3 Constructing Tables of Marks

70.3.1 TableOfMarks (for a group)

▷ TableOfMarks(*G*) (attribute)
▷ TableOfMarks(*string*) (attribute)
▷ TableOfMarks(*matrix*) (attribute)

In the first form, *G* must be a finite group, and TableOfMarks constructs the table of marks of *G*. This computation requires the knowledge of the complete subgroup lattice of *G* (see LatticeSubgroups (39.20.1)). If the lattice is not yet stored then it will be constructed. This may take a while if *G* is large. The result has the IsTableOfMarksWithGens (70.10.3) value true.

In the second form, *string* must be a string, and TableOfMarks gets the table of marks with name *string* from the GAP library (see 70.13). If no table of marks with this name is contained in the library then fail is returned.

In the third form, *matrix* must be a matrix or a list of rows describing a lower triangular matrix where the part above the diagonal is omitted. For such an argument *matrix*, TableOfMarks returns a table of marks object (see 70.2) for which *matrix* is the matrix of marks. Note that not every matrix (containing only nonnegative integers and having lower triangular shape) describes a table of marks of a group. Necessary conditions are checked with IsInternallyConsistent (70.9.1) (see 70.9), and fail is returned if *matrix* is proved not to describe a matrix of marks; but if TableOfMarks returns a table of marks object created from a matrix then it may still happen that this object does not describe the table of marks of a group.

For an overview of operations for table of marks objects, see the introduction to Chapter 70.

Example

```
gap> tom:= TableOfMarks( AlternatingGroup( 5 ) );
TableOfMarks( Alt( [ 1 .. 5 ] ) )
gap> TableOfMarks( "J5" );
fail
gap> a5:= TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> mat:=
> [ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
>   [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
>   [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
>   [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0 ],
>   [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ];;
gap> TableOfMarks( mat );
TableOfMarks( <9 classes> )
```

The following TableOfMarks methods for a group are installed.

- If the group is known to be cyclic then TableOfMarks constructs the table of marks essentially without the group, instead the knowledge about the structure of cyclic groups is used.
- If the lattice of subgroups is already stored in the group then TableOfMarks computes the table of marks from the lattice (see TableOfMarksByLattice (70.3.2)).
- If the group is known to be solvable then TableOfMarks takes the lattice of subgroups (see LatticeSubgroups (39.20.1)) of the group –which means that the lattice is computed if it is not yet stored– and then computes the table of marks from it. This method is also accessible via the global function TableOfMarksByLattice (70.3.2).
- If the group doesn't know its lattice of subgroups or its conjugacy classes of subgroups then the table of marks and the conjugacy classes of subgroups are computed at the same time by the cyclic extension method. Only the table of marks is stored because the conjugacy classes of subgroups or the lattice of subgroups can be easily read off (see LatticeSubgroupsByTom (70.3.3)).

Conversely, the lattice of subgroups of a group with known table of marks can be computed using the table of marks, via the function LatticeSubgroupsByTom (70.3.3). This is also installed as a method for LatticeSubgroups (39.20.1).

70.3.2 TableOfMarksByLattice

▷ TableOfMarksByLattice(*G*) (function)

TableOfMarksByLattice computes the table of marks of the group *G* from the lattice of subgroups of *G*. This lattice is computed via LatticeSubgroups (39.20.1) if it is not yet stored in *G*. The function TableOfMarksByLattice is installed as a method for TableOfMarks (70.3.1) for solvable groups and groups with stored subgroup lattice, and is available as a global variable only in order to provide explicit access to this method.

70.3.3 LatticeSubgroupsByTom

▷ LatticeSubgroupsByTom(*G*) (function)

LatticeSubgroupsByTom computes the lattice of subgroups of *G* from the table of marks of *G*, using RepresentativeTom (70.10.4).

70.4 Printing Tables of Marks

70.4.1 ViewObj (for a table of marks)

▷ ViewObj(*tom*) (method)

The default ViewObj (6.3.5) method for tables of marks prints the string "TableOfMarks", followed by –if known– the identifier (see Identifier (70.7.9)) or the group of the table of marks enclosed in brackets; if neither group nor identifier are known then just the number of conjugacy classes of subgroups is printed instead.

70.4.2 PrintObj (for a table of marks)

▷ `PrintObj(`*tom*`)` (method)

The default `PrintObj` (6.3.5) method for tables of marks does the same as `ViewObj` (6.3.5), except that `PrintObj` (6.3.5) is used for the group instead of `ViewObj` (6.3.5).

70.4.3 Display (for a table of marks)

▷ `Display(`*tom*`[, `*arec*`])` (method)

The default `Display` (6.3.6) method for a table of marks *tom* produces a formatted output of the marks in *tom*. Each line of output begins with the number of the corresponding class of subgroups. This number is repeated if the output spreads over several pages. The number of columns printed at one time depends on the actual line length, which can be accessed and changed by the function `SizeScreen` (6.12.1).

An interactive variant of `Display` (6.3.6) is the `Browse` (**Browse: Browse**) method for tables of marks that is provided by the GAP package Browse, see `Browse` (**Browse: Browse (for tables of marks)**).

The optional second argument *arec* of `Display` (6.3.6) can be used to change the default style for displaying a table of marks. *arec* must be a record, its relevant components are the following.

`classes`
: a list of class numbers to select only the rows and columns of the matrix that correspond to this list for printing,

`form`
: one of the strings `"subgroups"`, `"supergroups"`; in the former case, at position (i, j) of the matrix the number of conjugates of H_j contained in H_i is printed, and in the latter case, at position (i, j) the number of conjugates of H_i which contain H_j is printed.

Example

```
gap> tom:= TableOfMarks( "A5" );;
gap> Display( tom );
1:  60
2:  30 2
3:  20 . 2
4:  15 3 . 3
5:  12 . . . 2
6:  10 2 1 . . 1
7:   6 2 . . 1 . 1
8:   5 1 2 1 . . . 1
9:   1 1 1 1 1 1 1 1 1

gap> Display( tom, rec( classes:= [ 1, 2, 3, 4, 8 ] ) );
1:  60
2:  30 2
3:  20 . 2
4:  15 3 . 3
8:   5 1 2 1 1
```

```
gap> Display( tom, rec( form:= "subgroups" ) );
1:  1
2:  1  1
3:  1  .  1
4:  1  3  . 1
5:  1  .  . . 1
6:  1  3  1 . .  1
7:  1  5  . . 1  . 1
8:  1  3  4 1 .  . . 1
9:  1 15 10 5 6 10 6 5 1

gap> Display( tom, rec( form:= "supergroups" ) );
1:   1
2:  15 1
3:  10 . 1
4:   5 1 . 1
5:   6 . . . 1
6:  10 2 1 . . 1
7:   6 2 . . 1 . 1
8:   5 1 2 1 . . . 1
9:   1 1 1 1 1 1 1 1 1
```

70.5 Sorting Tables of Marks

70.5.1 SortedTom

▷ SortedTom(*tom*, *perm*) (operation)

SortedTom returns a table of marks where the rows and columns of the table of marks *tom* are reordered according to the permutation *perm*.

Note that in each table of marks in GAP, the matrix of marks is assumed to have lower triangular shape (see 70.2). If the permutation *perm* does *not* have this property then the functions for tables of marks might return wrong results when applied to the output of SortedTom.

The returned table of marks has only those attribute values stored that are known for *tom* and listed in TableOfMarksComponents (70.6.4).

Example

```
gap> tom:= TableOfMarksCyclic( 6 );;  Display( tom );
1:  6
2:  3 3
3:  2 . 2
4:  1 1 1 1

gap> sorted:= SortedTom( tom, (2,3) );;  Display( sorted );
1:  6
2:  2 2
3:  3 . 3
4:  1 1 1 1

gap> wrong:= SortedTom( tom, (1,2) );;  Display( wrong );
```

```
1:  3
2:  . 6
3:  . 2 2
4:  1 1 1 1
```

70.5.2 PermutationTom

▷ PermutationTom(*tom*) (attribute)

For the table of marks *tom* of the group G stored as UnderlyingGroup (70.7.7) value of *tom*, PermutationTom is a permutation π such that the i-th conjugacy class of subgroups of G belongs to the i^{π}-th column and row of marks in *tom*.

This attribute value is bound only if *tom* was obtained from another table of marks by permuting with SortedTom (70.5.1), and there is no default method to compute its value.

The attribute is necessary because the original and the sorted table of marks have the same identifier and the same group, and information computed from the group may depend on the ordering of marks, for example the fusion from the ordinary character table of G into *tom*.

Example

```
gap> MarksTom( tom )[2];
[ 3, 3 ]
gap> MarksTom( sorted )[2];
[ 2, 2 ]
gap> HasPermutationTom( sorted );
true
gap> PermutationTom( sorted );
(2,3)
```

70.6 Technical Details about Tables of Marks

70.6.1 InfoTom

▷ InfoTom (info class)

is the info class for computations concerning tables of marks.

70.6.2 IsTableOfMarks

▷ IsTableOfMarks(*obj*) (Category)

Each table of marks belongs to this category.

70.6.3 TableOfMarksFamily

▷ TableOfMarksFamily (global variable)

Each table of marks belongs to this family.

70.6.4 TableOfMarksComponents

▷ TableOfMarksComponents (global variable)

The list TableOfMarksComponents is used when a table of marks object is created from a record via ConvertToTableOfMarks (70.6.5). TableOfMarksComponents contains at position $2i-1$ a name of an attribute and at position $2i$ the corresponding attribute getter function.

70.6.5 ConvertToTableOfMarks

▷ ConvertToTableOfMarks(*record*) (function)

ConvertToTableOfMarks converts a record with components from TableOfMarksComponents (70.6.4) into a table of marks object with the corresponding attributes.

Example

```
gap> record:= rec( MarksTom:= [ [ 4 ], [ 2, 2 ], [ 1, 1, 1 ] ],
>  SubsTom:= [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] );;
gap> ConvertToTableOfMarks( record );;
gap> record;
TableOfMarks( <3 classes> )
```

70.7 Attributes of Tables of Marks

70.7.1 MarksTom

▷ MarksTom(*tom*) (attribute)
▷ SubsTom(*tom*) (attribute)

The matrix of marks (see 70.1) of the table of marks *tom* is stored in a compressed form where zeros are omitted, using the attributes MarksTom and SubsTom. If M is the square matrix of marks of *tom* (see MatTom (70.7.10)) then the SubsTom value of *tom* is a list that contains at position i the list of all positions of nonzero entries of the i-th row of M, and the MarksTom value of *tom* is a list that contains at position i the list of the corresponding marks.

MarksTom and SubsTom are defining attributes of tables of marks (see 70.2). There is no default method for computing the SubsTom value, and the default MarksTom method needs the values of NrSubsTom (70.7.2) and OrdersTom (70.7.2).

Example

```
gap> a5:= TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> MarksTom( a5 );
[ [ 60 ], [ 30, 2 ], [ 20, 2 ], [ 15, 3, 3 ], [ 12, 2 ],
  [ 10, 2, 1, 1 ], [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
gap> SubsTom( a5 );
[ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 5 ], [ 1, 2, 3, 6 ],
  [ 1, 2, 5, 7 ], [ 1, 2, 3, 4, 8 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ]
```

70.7.2 NrSubsTom

▷ NrSubsTom(*tom*) (attribute)
▷ OrdersTom(*tom*) (attribute)

Instead of storing the marks (see MarksTom (70.7.1)) of the table of marks *tom* one can use a matrix which contains at position (i, j) the number of subgroups of conjugacy class j that are contained in one member of the conjugacy class i. These values are stored in the NrSubsTom value in the same way as the marks in the MarksTom (70.7.1) value.

OrdersTom returns a list that contains at position i the order of a representative of the i-th conjugacy class of subgroups of *tom*.

One can compute the NrSubsTom and OrdersTom values from the MarksTom (70.7.1) value of *tom* and vice versa.

Example

```
gap> NrSubsTom( a5 );
[ [ 1 ], [ 1, 1 ], [ 1, 1 ], [ 1, 3, 1 ], [ 1, 1 ], [ 1, 3, 1, 1 ],
  [ 1, 5, 1, 1 ], [ 1, 3, 4, 1, 1 ], [ 1, 15, 10, 5, 6, 10, 6, 5, 1 ]
 ]
gap> OrdersTom( a5 );
[ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]
```

70.7.3 LengthsTom

▷ LengthsTom(*tom*) (attribute)

For a table of marks *tom*, LengthsTom returns a list of the lengths of the conjugacy classes of subgroups.

Example

```
gap> LengthsTom( a5 );
[ 1, 15, 10, 5, 6, 10, 6, 5, 1 ]
```

70.7.4 ClassTypesTom

▷ ClassTypesTom(*tom*) (attribute)

ClassTypesTom distinguishes isomorphism types of the classes of subgroups of the table of marks *tom* as far as this is possible from the SubsTom (70.7.1) and MarksTom (70.7.1) values of *tom*.

Two subgroups are clearly not isomorphic if they have different orders. Moreover, isomorphic subgroups must contain the same number of subgroups of each type.

Each type is represented by a positive integer. ClassTypesTom returns the list which contains for each class of subgroups its corresponding type.

Example

```
gap> a6:= TableOfMarks( "A6" );;
gap> ClassTypesTom( a6 );
[ 1, 2, 3, 3, 4, 5, 6, 6, 7, 7, 8, 9, 10, 11, 11, 12, 13, 13, 14, 15,
  15, 16 ]
```

70.7.5 ClassNamesTom

▷ ClassNamesTom(*tom*) (attribute)

ClassNamesTom constructs generic names for the conjugacy classes of subgroups of the table of marks *tom*. In general, the generic name of a class of non-cyclic subgroups consists of three parts and has the form "(*o*)_{*t*}*l*", where *o* indicates the order of the subgroup, *t* is a number that distinguishes different types of subgroups of the same order, and *l* is a letter that distinguishes classes of subgroups of the same type and order. The type of a subgroup is determined by the numbers of its subgroups of other types (see ClassTypesTom (70.7.4)). This is slightly weaker than isomorphism.

The letter is omitted if there is only one class of subgroups of that order and type, and the type is omitted if there is only one class of that order. Moreover, the braces {} around the type are omitted if the type number has only one digit.

For classes of cyclic subgroups, the parentheses round the order and the type are omitted. Hence the most general form of their generic names is "*o*,*l*". Again, the letter is omitted if there is only one class of cyclic subgroups of that order.

Example

```
gap> ClassNamesTom( a6 );
[ "1", "2", "3a", "3b", "5", "4", "(4)_2a", "(4)_2b", "(6)a", "(6)b",
  "(8)", "(9)", "(10)", "(12)a", "(12)b", "(18)", "(24)a", "(24)b",
  "(36)", "(60)a", "(60)b", "(360)" ]
```

70.7.6 FusionsTom

▷ FusionsTom(*tom*) (attribute)

For a table of marks *tom*, FusionsTom is a list of fusions into other tables of marks. Each fusion is a list of length two, the first entry being the Identifier (70.7.9)) value of the image table, the second entry being the list of images of the class positions of *tom* in the image table.

This attribute is mainly used for tables of marks in the GAP library (see 70.13).

Example

```
gap> fus:= FusionsTom( a6 );;
gap> fus[1];
[ "L3(4)",
  [ 1, 2, 3, 3, 14, 5, 9, 7, 15, 15, 24, 26, 27, 32, 33, 50, 57, 55,
      63, 73, 77, 90 ] ]
```

70.7.7 UnderlyingGroup (for tables of marks)

▷ UnderlyingGroup(*tom*) (attribute)

UnderlyingGroup is used to access an underlying group that is stored on the table of marks *tom*. There is no default method to compute an underlying group if it is not stored.

Example

```
gap> UnderlyingGroup( a6 );
Group([ (1,2)(3,4), (1,2,4,5)(3,6) ])
```

70.7.8 IdempotentsTom

▷ IdempotentsTom(*tom*) (attribute)
▷ IdempotentsTomInfo(*tom*) (attribute)

IdempotentsTom encodes the idempotents of the integral Burnside ring described by the table of marks *tom*. The return value is a list l of positive integers such that each row vector describing a primitive idempotent has value 1 at all positions with the same entry in l, and 0 at all other positions.

According to A. Dress [Dre69] (see also [Pfe97]), these idempotents correspond to the classes of perfect subgroups, and each such idempotent is the characteristic function of all those subgroups that arise by cyclic extension from the corresponding perfect subgroup (see CyclicExtensionsTom (70.9.7)).

IdempotentsTomInfo returns a record with components fixpointvectors and primidems, both bound to lists. The i-th entry of the fixpointvectors list is the $0-1$-vector describing the i-th primitive idempotent, and the i-th entry of primidems is the decomposition of this idempotent in the rows of *tom*.

Example

```
gap> IdempotentsTom( a5 );
[ 1, 1, 1, 1, 1, 1, 1, 1, 9 ]
gap> IdempotentsTomInfo( a5 );
rec(
  fixpointvectors := [ [ 1, 1, 1, 1, 1, 1, 1, 1, 0 ],
      [ 0, 0, 0, 0, 0, 0, 0, 0, 1 ] ],
  primidems := [ [ 1, -2, -1, 0, 0, 1, 1, 1 ],
      [ -1, 2, 1, 0, 0, -1, -1, -1, 1 ] ] )
```

70.7.9 Identifier (for tables of marks)

▷ Identifier(*tom*) (attribute)

The identifier of a table of marks *tom* is a string. It is used for printing the table of marks (see 70.4) and in fusions between tables of marks (see FusionsTom (70.7.6)).

If *tom* is a table of marks from the GAP library of tables of marks (see 70.13) then it has an identifier, and if *tom* was constructed from a group with Name (12.8.2) then this name is chosen as Identifier value. There is no default method to compute an identifier in all other cases.

Example

```
gap> Identifier( a5 );
"A5"
```

70.7.10 MatTom

▷ MatTom(*tom*) (attribute)

MatTom returns the square matrix of marks (see 70.1) of the table of marks *tom* which is stored in a compressed form using the attributes MarksTom (70.7.1) and SubsTom (70.7.1) This may need substantially more space than the values of MarksTom (70.7.1) and SubsTom (70.7.1).

Example

```
gap> MatTom( a5 );
[ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
```

```
    [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
    [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
    [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

70.7.11 MoebiusTom

▷ MoebiusTom(*tom*) (attribute)

MoebiusTom computes the Möbius values both of the subgroup lattice of the group G with table of marks *tom* and of the poset of conjugacy classes of subgroups of G. It returns a record where the component mu contains the Möbius values of the subgroup lattice, and the component nu contains the Möbius values of the poset.

Moreover, according to an observation of Isaacs et al. (see [HIÖ89], [Pah93]), the values on the subgroup lattice often can be derived from those of the poset of conjugacy classes. These "expected values" are returned in the component ex, and the list of numbers of those subgroups where the expected value does not coincide with the actual value are returned in the component hyp. For the computation of these values, the position of the derived subgroup of G is needed (see DerivedSubgroupTom (70.9.2)). If it is not uniquely determined then the result does not have the components ex and hyp.

Example

```
gap> MoebiusTom( a5 );
rec( ex := [ -60, 4, 2,,, -1, -1, -1, 1 ], hyp := [  ],
  mu := [ -60, 4, 2,,, -1, -1, -1, 1 ],
  nu := [ -1, 2, 1,,, -1, -1, -1, 1 ] )
gap> tom:= TableOfMarks( "M12" );;
gap> moebius:= MoebiusTom( tom );;
gap> moebius.hyp;
[ 1, 2, 4, 16, 39, 45, 105 ]
gap> moebius.mu[1];  moebius.ex[1];
95040
190080
```

70.7.12 WeightsTom

▷ WeightsTom(*tom*) (attribute)

WeightsTom extracts the *weights* from the table of marks *tom*, i.e., the diagonal entries of the matrix of marks (see MarksTom (70.7.1)), indicating the index of a subgroup in its normalizer.

Example

```
gap> wt:= WeightsTom( a5 );
[ 60, 2, 2, 3, 2, 1, 1, 1, 1 ]
```

This information may be used to obtain the numbers of conjugate supergroups from the marks.

Example

```
gap> marks:= MarksTom( a5 );;
gap> List( [ 1 .. 9 ], x -> marks[x] / wt[x] );
[ [ 1 ], [ 15, 1 ], [ 10, 1 ], [ 5, 1, 1 ], [ 6, 1 ], [ 10, 2, 1, 1 ],
  [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ], [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

70.8 Properties of Tables of Marks

For a table of marks *tom* of a group G, the following properties have the same meaning as the corresponding properties for G. Additionally, if a positive integer *sub* is given as the second argument then the value of the corresponding property for the *sub*-th class of subgroups of *tom* is returned.

70.8.1 IsAbelianTom

▷ IsAbelianTom(*tom[, sub]*) (property)
▷ IsCyclicTom(*tom[, sub]*) (property)
▷ IsNilpotentTom(*tom[, sub]*) (property)
▷ IsPerfectTom(*tom[, sub]*) (property)
▷ IsSolvableTom(*tom[, sub]*) (property)

Example

```
gap> tom:= TableOfMarks( "A5" );;
gap> IsAbelianTom( tom );  IsPerfectTom( tom );
false
true
gap> IsAbelianTom( tom, 3 );  IsNilpotentTom( tom, 7 );
true
false
gap> IsPerfectTom( tom, 7 );  IsSolvableTom( tom, 7 );
false
true
gap> for i in [ 1 .. 6 ] do
> Print( i, ": ", IsCyclicTom(a5, i), "  " );
> od;  Print( "\n" );
1: true  2: true  3: true  4: false  5: true  6: false
```

70.9 Other Operations for Tables of Marks

70.9.1 IsInternallyConsistent (for tables of marks)

▷ IsInternallyConsistent(*tom*) (method)

For a table of marks *tom*, IsInternallyConsistent decomposes all tensor products of rows of *tom*. It returns true if all decomposition numbers are nonnegative integers, and false otherwise. This provides a strong consistency check for a table of marks.

70.9.2 DerivedSubgroupTom

▷ DerivedSubgroupTom(*tom, sub*) (operation)
▷ DerivedSubgroupsTom(*tom*) (function)

For a table of marks *tom* and a positive integer *sub*, DerivedSubgroupTom returns either a positive integer i or a list l of positive integers. In the former case, the result means that the derived subgroups of the subgroups in the *sub*-th class of *tom* lie in the i-th class. In the latter case, the class

of the derived subgroups could not be uniquely determined, and the position of the class of derived subgroups is an entry of *l*.

Values computed with DerivedSubgroupTom are stored using the attribute DerivedSubgroupsTomPossible (70.9.3).

DerivedSubgroupsTom is just the list of DerivedSubgroupTom values for all values of *sub*.

70.9.3 DerivedSubgroupsTomPossible

▷ DerivedSubgroupsTomPossible(*tom*) (attribute)
▷ DerivedSubgroupsTomUnique(*tom*) (attribute)

Let *tom* be a table of marks. The value of the attribute DerivedSubgroupsTomPossible is a list in which the value at position *i* –if bound– is a positive integer or a list; the meaning of the entry is the same as in DerivedSubgroupTom (70.9.2).

If the value of the attribute DerivedSubgroupsTomUnique is known for *tom* then it is a list of positive integers, the value at position *i* being the position of the class of derived subgroups of the *i*-th class of subgroups in *tom*.

The derived subgroups are in general not uniquely determined by the table of marks if no UnderlyingGroup (70.7.7) value is stored, so there is no default method for DerivedSubgroupsTomUnique. But in some cases the derived subgroups are explicitly set when the table of marks is constructed. In this case, DerivedSubgroupTom (70.9.2) does not set values in the DerivedSubgroupsTomPossible list.

The DerivedSubgroupsTomUnique value is automatically set when the last missing unique value is entered in the DerivedSubgroupsTomPossible list by DerivedSubgroupTom (70.9.2).

70.9.4 NormalizerTom

▷ NormalizerTom(*tom, sub*) (operation)
▷ NormalizersTom(*tom*) (attribute)

Let *tom* be the table of marks of a group *G*, say. NormalizerTom tries to find the conjugacy class of the normalizer *N* in *G* of a subgroup *U* in the *sub*-th class of *tom*. The return value is either the list of class numbers of those subgroups that have the right size and contain the subgroup and all subgroups that clearly contain it as a normal subgroup, or the class number of the normalizer if it is uniquely determined by these conditions. If *tom* knows the subgroup lattice of *G* (see IsTableOfMarksWithGens (70.10.3)) then all normalizers are uniquely determined. NormalizerTom should never return an empty list.

NormalizersTom returns the list of positions of the classes of normalizers of subgroups in *tom*. In addition to the criteria for a single class of subgroup used by NormalizerTom, the approximations of normalizers for several classes are used and thus NormalizersTom may return better approximations than NormalizerTom.

Example

```
gap> NormalizerTom( a5, 4 );
8
gap> NormalizersTom( a5 );
[ 9, 4, 6, 8, 7, 6, 7, 8, 9 ]
```

The example shows that a subgroup with class number 4 in A_5 (which is a Kleinian four group) is normalized by a subgroup in class 8. This class contains the subgroups of A_5 which are isomorphic to A_4.

70.9.5 ContainedTom

▷ ContainedTom(*tom, sub1, sub2*) (operation)

ContainedTom returns the number of subgroups in class *sub1* of the table of marks *tom* that are contained in one fixed member of the class *sub2*.

Example

```
gap> ContainedTom( a5, 3, 5 );  ContainedTom( a5, 3, 8 );
0
4
```

70.9.6 ContainingTom

▷ ContainingTom(*tom, sub1, sub2*) (operation)

ContainingTom returns the number of subgroups in class *sub2* of the table of marks *tom* that contain one fixed member of the class *sub1*.

Example

```
gap> ContainingTom( a5, 3, 5 );  ContainingTom( a5, 3, 8 );
0
2
```

70.9.7 CyclicExtensionsTom (for a prime)

▷ CyclicExtensionsTom(*tom, p*) (operation)
▷ CyclicExtensionsTom(*tom[, list]*) (attribute)

According to A. Dress [Dre69], two columns of the table of marks *tom* are equal modulo the prime *p* if and only if the corresponding subgroups are connected by a chain of normal extensions of order *p*.

Called with *tom* and *p*, CyclicExtensionsTom returns the classes of this equivalence relation.

In the second form, *list* must be a list of primes, and the return value is the list of classes of the relation obtained by considering chains of normal extensions of prime order where all primes are in *list*. The default value for *list* is the set of prime divisors of the order of the group of *tom*.

(This information is *not* used by NormalizerTom (70.9.4) although it might give additional restrictions in the search of normalizers.)

Example

```
gap> CyclicExtensionsTom( a5, 2 );
[ [ 1, 2, 4 ], [ 3, 6 ], [ 5, 7 ], [ 8 ], [ 9 ] ]
```

70.9.8 DecomposedFixedPointVector

▷ DecomposedFixedPointVector(*tom, fix*) (operation)

Let *tom* be the table of marks of the group G, say, and let *fix* be a vector of fixed point numbers w.r.t. an action of G, i.e., a vector which contains for each class of subgroups the number of fixed points under the given action. DecomposedFixedPointVector returns the decomposition of *fix* into rows of the table of marks. This decomposition corresponds to a decomposition of the action into transitive constituents. Trailing zeros in *fix* may be omitted.

Example

```
gap> DecomposedFixedPointVector( a5, [ 16, 4, 1, 0, 1, 1, 1 ] );
[ 0, 0, 0, 0, 0, 1, 1 ]
```

The vector *fix* may be any vector of integers. The resulting decomposition, however, will not be integral, in general.

Example

```
gap> DecomposedFixedPointVector( a5, [ 0, 0, 0, 0, 1, 1 ] );
[ 2/5, -1, -1/2, 0, 1/2, 1 ]
```

70.9.9 EulerianFunctionByTom

▷ EulerianFunctionByTom(*tom, n[, sub]*) (operation)

Called with two arguments, EulerianFunctionByTom computes the Eulerian function (see EulerianFunction (39.16.3)) of the underlying group G of the table of marks *tom*, that is, the number of *n*-tuples of elements in G that generate G. If the optional argument *sub* is given then EulerianFunctionByTom computes the Eulerian function of each subgroup in the *sub*-th class of subgroups of *tom*.

For a group G whose table of marks is known, EulerianFunctionByTom is installed as a method for EulerianFunction (39.16.3).

Example

```
gap> EulerianFunctionByTom( a5, 2 );
2280
gap> EulerianFunctionByTom( a5, 3 );
200160
gap> EulerianFunctionByTom( a5, 2, 3 );
8
```

70.9.10 IntersectionsTom

▷ IntersectionsTom(*tom, sub1, sub2*) (operation)

The intersections of the groups in the *sub1*-th conjugacy class of subgroups of the table of marks *tom* with the groups in the *sub2*-th conjugacy classes of subgroups of *tom* are determined up to conjugacy by the decomposition of the tensor product of their rows of marks. IntersectionsTom returns a list l that describes this decomposition. The i-th entry in l is the multiplicity of groups in the i-th conjugacy class as an intersection.

Example

```
gap> IntersectionsTom( a5, 8, 8 );
[ 0, 0, 1, 0, 0, 0, 0, 1 ]
```

Any two subgroups of class number 8 (A_4) of A_5 are either equal and their intersection has again class number 8, or their intersection has class number 3, and is a cyclic subgroup of order 3.

70.9.11 FactorGroupTom

▷ FactorGroupTom(*tom, n*) (operation)

For a table of marks *tom* of the group G, say, and the normal subgroup N of G corresponding to the *n*-th class of subgroups of *tom*, FactorGroupTom returns the table of marks of the factor group G/N.

Example

```
gap> s4:= TableOfMarks( SymmetricGroup( 4 ) );
TableOfMarks( Sym( [ 1 .. 4 ] ) )
gap> LengthsTom( s4 );
[ 1, 3, 6, 4, 1, 3, 3, 4, 3, 1, 1 ]
gap> OrdersTom( s4 );
[ 1, 2, 2, 3, 4, 4, 4, 6, 8, 12, 24 ]
gap> s3:= FactorGroupTom( s4, 5 );
TableOfMarks( Group([ f1, f2 ]) )
gap> Display( s3 );
1:  6
2:  3 1
3:  2 . 2
4:  1 1 1 1
```

70.9.12 MaximalSubgroupsTom

▷ MaximalSubgroupsTom(*tom[, sub]*) (attribute)

Called with a table of marks *tom*, MaximalSubgroupsTom returns a list of length two, the first entry being the list of positions of the classes of maximal subgroups of the whole group of *tom*, the second entry being the list of class lengths of these groups.

Called with a table of marks *tom* and a position *sub*, the same information for the *sub*-th class of subgroups is returned.

Example

```
gap> MaximalSubgroupsTom( s4 );
[ [ 10, 9, 8 ], [ 1, 3, 4 ] ]
gap> MaximalSubgroupsTom( s4, 10 );
[ [ 5, 4 ], [ 1, 4 ] ]
```

70.9.13 MinimalSupergroupsTom

▷ MinimalSupergroupsTom(*tom, sub*) (operation)

For a table of marks *tom*, MinimalSupergroupsTom returns a list of length two, the first entry being the list of positions of the classes containing the minimal supergroups of the groups in the *sub*-th class of subgroups of *tom*, the second entry being the list of class lengths of these groups.

Example

```
gap> MinimalSupergroupsTom( s4, 5 );
[ [ 9, 10 ], [ 3, 1 ] ]
```

70.10 Accessing Subgroups via Tables of Marks

Let *tom* be the table of marks of the group G, and assume that *tom* has access to G via the UnderlyingGroup (70.7.7) value. Then it makes sense to use *tom* and its ordering of conjugacy classes of subgroups of G for storing information for constructing representatives of these classes. The group G is in general not sufficient for this, *tom* needs more information; this is available if and only if the IsTableOfMarksWithGens (70.10.3) value of *tom* is true. In this case, RepresentativeTom (70.10.4) can be used to get a subgroup of the i-th class, for all i.

GAP provides two different possibilities to store generators of the representatives of classes of subgroups. The first is implemented by the attribute GeneratorsSubgroupsTom (70.10.1), which uses explicit generators of the subgroups. The second, more general, possibility is implemented by the attribute StraightLineProgramsTom (70.10.2), which encodes the generators as straight line programs (see 37.8) that evaluate to the generators in question when applied to *standard generators* of G. This means that on the one hand, standard generators of G must be known in order to use StraightLineProgramsTom (70.10.2). On the other hand, the straight line programs allow one to compute easily generators not only of a subgroup U of G but also generators of the image of U in any representation of G, provided that one knows standard generators of the image of G under this representation. See the manual of the package TomLib for details and an example.

70.10.1 GeneratorsSubgroupsTom

▷ GeneratorsSubgroupsTom(*tom*) (attribute)

Let *tom* be a table of marks with IsTableOfMarksWithGens (70.10.3) value true. Then GeneratorsSubgroupsTom returns a list of length two, the first entry being a list l of elements of the group stored as UnderlyingGroup (70.7.7) value of *tom*, the second entry being a list that contains at position i a list of positions in l of generators of a representative of a subgroup in class i.

The GeneratorsSubgroupsTom value is known for all tables of marks that have been computed with TableOfMarks (70.3.1) from a group, and there is a method to compute the value for a table of marks that admits RepresentativeTom (70.10.4).

70.10.2 StraightLineProgramsTom

▷ StraightLineProgramsTom(*tom*) (attribute)

For a table of marks *tom* with IsTableOfMarksWithGens (70.10.3) value true, StraightLineProgramsTom returns a list that contains at position i either a list of straight line programs or a straight line program (see 37.8), encoding the generators of a representative of the i-th conjugacy class of subgroups of UnderlyingGroup(*tom*); in the former case, each straight line program returns a generator, in the latter case, the program returns the list of generators.

There is no default method to compute the StraightLineProgramsTom value of a table of marks if they are not yet stored. The value is known for all tables of marks that belong to the GAP library of tables of marks (see 70.13).

70.10.3 IsTableOfMarksWithGens

▷ IsTableOfMarksWithGens(*tom*) (function)

This filter shall express the union of the filters IsTableOfMarks and HasStraightLineProgramsTom and IsTableOfMarks and HasGeneratorsSubgroupsTom. If a table of marks *tom* has this filter set then *tom* can be asked to compute information that is in general not uniquely determined by a table of marks, for example the positions of derived subgroups or normalizers of subgroups (see DerivedSubgroupTom (70.9.2), NormalizerTom (70.9.4)).

Example

```
gap> a5:= TableOfMarks( "A5" );;  IsTableOfMarksWithGens( a5 );
true
gap> HasGeneratorsSubgroupsTom( a5 );  HasStraightLineProgramsTom( a5 );
false
true
gap> alt5:= TableOfMarks( AlternatingGroup( 5 ) );;
gap> IsTableOfMarksWithGens( alt5 );
true
gap> HasGeneratorsSubgroupsTom(alt5); HasStraightLineProgramsTom(alt5);
true
false
gap> progs:= StraightLineProgramsTom( a5 );;
gap> OrdersTom( a5 );
[ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]
gap> IsCyclicTom( a5, 4 );
false
gap> Length( progs[4] );
2
gap> progs[4][1];
<straight line program>
gap> # first generator of an el. ab group of order 4:
gap> Display( progs[4][1] );
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[2]*r[1];
r[4]:= r[3]*r[2]^-1*r[1]*r[3]*r[2]^-1*r[1]*r[2];
# return value:
r[4]
gap> x:= [ [ Z(2)^0, 0*Z(2) ], [ Z(2^2), Z(2)^0 ] ];;
gap> y:= [ [ Z(2^2), Z(2)^0 ], [ 0*Z(2), Z(2^2)^2 ] ];;
gap> res1:= ResultOfStraightLineProgram( progs[4][1], [ x, y ] );
[ [ Z(2)^0, 0*Z(2) ], [ Z(2^2)^2, Z(2)^0 ] ]
gap> res2:= ResultOfStraightLineProgram( progs[4][2], [ x, y ] );
[ [ Z(2)^0, 0*Z(2) ], [ Z(2^2), Z(2)^0 ] ]
gap> w:= y*x;;
gap> res1 = w*y^-1*x*w*y^-1*x*y;
```

```
true
gap> subgrp:= Group( res1, res2 );;  Size( subgrp );  IsCyclic( subgrp );
4
false
```

70.10.4 RepresentativeTom

▷ RepresentativeTom(*tom, sub*) (operation)
▷ RepresentativeTomByGenerators(*tom, sub, gens*) (operation)
▷ RepresentativeTomByGeneratorsNC(*tom, sub, gens*) (operation)

Let *tom* be a table of marks with IsTableOfMarksWithGens (70.10.3) value true, and *sub* a positive integer. RepresentativeTom returns a representative of the *sub*-th conjugacy class of subgroups of *tom*.

If the attribute StraightLineProgramsTom (70.10.2) is set in *tom* then methods for the operations RepresentativeTomByGenerators and RepresentativeTomByGeneratorsNC are available, which return a representative of the *sub*-th conjugacy class of subgroups of *tom*, as a subgroup of the group generated by *gens*. This means that the standard generators of *tom* are replaced by *gens*.

RepresentativeTomByGenerators checks whether mapping the standard generators of *tom* to *gens* extends to a group isomorphism, and returns fail if not. RepresentativeTomByGeneratorsNC omits all checks. So RepresentativeTomByGenerators is thought mainly for debugging purposes; note that when several representatives are constructed, it is cheaper to construct (and check) the isomorphism once, and to map the groups returned by RepresentativeTom under this isomorphism. The idea behind RepresentativeTomByGeneratorsNC, however, is to avoid the overhead of using isomorphisms when *gens* are known to be standard generators. In order to proceed like this, the attribute StraightLineProgramsTom (70.10.2) is needed.

Example

```
gap> RepresentativeTom( a5, 4 );
Group([ (2,3)(4,5), (2,4)(3,5) ])
```

70.11 The Interface between Tables of Marks and Character Tables

The following examples require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

Example

```
gap> LoadPackage( "ctbllib" );
true
```

70.11.1 FusionCharTableTom

▷ FusionCharTableTom(*tbl, tom*) (operation)
▷ PossibleFusionsCharTableTom(*tbl, tom[, options]*) (operation)

Let *tbl* be the ordinary character table of the group G, say, and *tom* the table of marks of G. FusionCharTableTom determines the fusion of the classes of elements from *tbl* to the classes of

cyclic subgroups on *tom*, that is, a list that contains at position *i* the position of the class of cyclic subgroups in *tom* that are generated by elements in the *i*-th conjugacy class of elements in *tbl*.

Three cases are handled differently.

1. The fusion is explicitly stored on *tbl*. Then nothing has to be done. This happens only if both *tbl* and *tom* are tables from the GAP library (see 70.13 and the manual of the GAP Character Table Library).

2. The UnderlyingGroup (70.7.7) values of *tbl* and *tom* are known and equal. Then the group is used to compute the fusion.

3. There is neither fusion nor group information available. In this case only necessary conditions can be checked, and if they are not sufficient to determine the fusion uniquely then fail is returned by FusionCharTableTom.

PossibleFusionsCharTableTom computes the list of possible fusions from *tbl* to *tom*, according to the criteria that have been checked. So if FusionCharTableTom returns a unique fusion then the list returned by PossibleFusionsCharTableTom for the same arguments contains exactly this fusion, and if FusionCharTableTom returns fail then the length of this list is different from 1.

The optional argument *options* must be a record that may have the following components.

fusionmap
: a parametrized map which is an approximation of the desired map,

quick
: a Boolean; if true then as soon as only one possibility remains this possibility is returned immediately; the default value is false.

Example

```
gap> a5c:= CharacterTable( "A5" );;
gap> fus:= FusionCharTableTom( a5c, a5 );
[ 1, 2, 3, 5, 5 ]
```

70.11.2 PermCharsTom (via fusion map)

▷ PermCharsTom(*fus, tom*) (operation)
▷ PermCharsTom(*tbl, tom*) (operation)

PermCharsTom returns the list of transitive permutation characters from the table of marks *tom*. In the first form, *fus* must be the fusion map from the ordinary character table of the group of *tom* to *tom* (see FusionCharTableTom (70.11.1)). In the second form, *tbl* must be the character table of the group of which *tom* is the table of marks. If the fusion map is not uniquely determined (see FusionCharTableTom (70.11.1)) then fail is returned.

If the fusion map *fus* is given as first argument then each transitive permutation character is represented by its values list. If the character table *tbl* is given then the permutation characters are class function objects (see Chapter 72).

Example

```
gap> PermCharsTom( a5c, a5 );
[ Character( CharacterTable( "A5" ), [ 60, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 30, 2, 0, 0, 0 ] ),
```

```
  Character( CharacterTable( "A5" ), [ 20, 0, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 15, 3, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 12, 0, 0, 2, 2 ] ),
  Character( CharacterTable( "A5" ), [ 10, 2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 6, 2, 0, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ) ]
gap> PermCharsTom( fus, a5 )[1];
[ 60, 0, 0, 0, 0 ]
```

70.12 Generic Construction of Tables of Marks

The following three operations construct a table of marks only from the data given, i.e., without underlying group.

70.12.1 TableOfMarksCyclic

▷ `TableOfMarksCyclic(n)` (operation)

`TableOfMarksCyclic` returns the table of marks of the cyclic group of order `n`.

A cyclic group of order `n` has as its subgroups for each divisor d of `n` a cyclic subgroup of order d.

Example

```
gap> Display( TableOfMarksCyclic( 6 ) );
1:  6
2:  3 3
3:  2 . 2
4:  1 1 1 1
```

70.12.2 TableOfMarksDihedral

▷ `TableOfMarksDihedral(n)` (operation)

`TableOfMarksDihedral` returns the table of marks of the dihedral group of order `m`.

For each divisor d of `m`, a dihedral group of order $m = 2n$ contains subgroups of order d according to the following rule. If d is odd and divides n then there is only one cyclic subgroup of order d. If d is even and divides n then there are a cyclic subgroup of order d and two classes of dihedral subgroups of order d (which are cyclic, too, in the case $d = 2$, see the example below). Otherwise (i.e., if d does not divide n) there is just one class of dihedral subgroups of order d.

Example

```
gap> Display( TableOfMarksDihedral( 12 ) );
 1:  12
 2:   6 6
 3:   6 . 2
 4:   6 . . 2
 5:   4 . . . 4
 6:   3 3 1 1 . 1
```

```
 7:   2 2 . . 2 . 2
 8:   2 . 2 . 2 . . 2
 9:   2 . . 2 2 . . . 2
10:   1 1 1 1 1 1 1 1 1 1
```

70.12.3 TableOfMarksFrobenius

▷ `TableOfMarksFrobenius(`*p*`, `*q*`)` (operation)

`TableOfMarksFrobenius` computes the table of marks of a Frobenius group of order pq, where p is a prime and q divides $p-1$.

Example

```
gap> Display( TableOfMarksFrobenius( 5, 4 ) );
1:  20
2:  10 2
3:   5 1 1
4:   4 . . 4
5:   2 2 . 2 2
6:   1 1 1 1 1 1
```

70.13 The Library of Tables of Marks

The GAP package TomLib provides access to several hundred tables of marks of almost simple groups and their maximal subgroups. If this package is installed then the tables from this database can be accessed via `TableOfMarks` (70.3.1) with argument a string. If also the GAP Character Table Library is installed and contains the ordinary character table of the group for which one wants to fetch the table of marks then one can also call `TableOfMarks` (70.3.1) with argument the character table.

A list of all names of tables of marks that are provided by the TomLib package can be obtained via `AllLibTomNames` (**TomLib: AllLibTomNames**).

Example

```
gap> names:= AllLibTomNames();;
gap> "A5" in names;
true
```

Chapter 71

Character Tables

This chapter describes operations for *character tables of finite groups*.

Operations for *characters* (or, more general, *class functions*) are described in Chapter 72.

For a description of the GAP Library of Character Tables, see the separate manual for the GAP package CTblLib.

Several examples in this chapter require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

Example

```
gap> LoadPackage( "ctbllib" );
true
```

71.1 Some Remarks about Character Theory in GAP

It seems to be necessary to state some basic facts –and maybe warnings– at the beginning of the character theory package. This holds for people who are familiar with character theory because there is no global reference on computational character theory, although there are many papers on this topic, such as [NPP84] or [LP91]. It holds, however, also for people who are familiar with GAP because the general concept of domains (see Chapter 12.4) plays no important role here –we will justify this later in this section.

Intuitively, *characters* (or more generally, *class functions*) of a finite group G can be thought of as certain mappings defined on G, with values in the complex number field; the set of all characters of G forms a semiring, with both addition and multiplication defined pointwise, which is naturally embedded into the ring of *generalized* (or *virtual*) *characters* in the natural way. A $\mathbb{Z}$-basis of this ring, and also a vector space basis of the complex vector space of class functions of G, is given by the irreducible characters of G.

At this stage one could ask where there is a problem, since all these algebraic structures are supported by GAP. But in practice, these structures are of minor importance, compared to individual characters and the *character tables* themselves (which are not domains in the sense of GAP).

For computations with characters of a finite group G with n conjugacy classes, say, we fix an ordering of the classes, and then identify each class with its position according to this ordering. Each character of G can be represented by a list of length n in which the character value for elements of the i-th class is stored at the i-th position. Note that we need not know the conjugacy classes of G physically, even our knowledge of G may be implicit in the sense that, e.g., we know how many classes of involutions G has, and which length these classes have, but we never have seen an element

of G, or a presentation or representation of G. This allows us to work with the character tables of very large groups, e.g., of the so-called monster, where GAP has (currently) no chance to deal with the group.

As a consequence, also other information involving characters is given implicitly. For example, we can talk about the kernel of a character not as a group but as a list of classes (more exactly: a list of their positions according to the chosen ordering of classes) forming this kernel; we can deduce the group order, the contained cyclic subgroups and so on, but we do not get the group itself.

So typical calculations with characters involve loops over lists of character values. For example, the scalar product of two characters χ, ψ of G given by

$$[\chi,\psi] = \left(\sum_{g \in G} \chi(g)\psi(g^{-1}) \right) / |G|$$

can be written as

```
Sum( [ 1 .. n ], i -> SizesConjugacyClasses( t )[i] * chi[i]
                          * ComplexConjugate( psi[i] ) ) / Size( t );
```

where `t` is the character table of G, and `chi`, `psi` are the lists of values of χ, ψ, respectively.

It is one of the advantages of character theory that after one has translated a problem concerning groups into a problem concerning only characters, the necessary calculations are mostly simple. For example, one can often prove that a group is a Galois group over the rationals using calculations with structure constants that can be computed from the character table, and information about (the character tables of) maximal subgroups. When one deals with such questions, the translation back to groups is just an interpretation by the user, it does not take place in GAP.

GAP uses character *tables* to store information such as class lengths, element orders, the irreducible characters of G etc. in a consistent way; in the example above, we have seen that `SizesConjugacyClasses` (71.9.3) returns the list of class lengths of its argument. Note that the values of these attributes rely on the chosen ordering of conjugacy classes, a character table is not determined by something similar to generators of groups or rings in GAP where knowledge could in principle be recovered from the generators but is stored mainly for the sake of efficiency.

Note that the character table of a group G in GAP must *not* be mixed up with the list of complex irreducible characters of G. The irreducible characters are stored in a character table via the attribute `Irr` (71.8.2).

Two further important instances of information that depends on the ordering of conjugacy classes are *power maps* and *fusion maps*. Both are represented as lists of integers in GAP. The k-th power map maps each class to the class of k-th powers of its elements, the corresponding list contains at each position the position of the image. A class fusion map between the classes of a subgroup H of G and the classes of G maps each class c of H to that class of G that contains c, the corresponding list contains again the positions of image classes; if we know only the character tables of H and G but not the groups themselves, this means with respect to a fixed embedding of H into G. More about power maps and fusion maps can be found in Chapter 73.

So class functions, power maps, and fusion maps are represented by lists in GAP. If they are plain lists then they are regarded as class functions etc. of an appropriate character table when they are passed to GAP functions that expect class functions etc. For example, a list with all entries equal to 1 is regarded as the trivial character if it is passed to a function that expects a character. Note that this approach requires the character table as an argument for such a function.

One can construct class function objects that store their underlying character table and other attribute values (see Chapter 72). This allows one to omit the character table argument in many functions, and it allows one to use infix operations for tensoring or inducing class functions.

71.2 History of Character Theory Stuff in GAP

GAP provides functions for dealing with group characters since the version GAP 3.1, which was released in March 1992. The reason for adding this branch of mathematics to the topics of GAP was (apart from the usefulness of character theoretic computations in general) the insight that GAP provides an ideal environment for developing the algorithms needed. In particular, it had been decided at Lehrstuhl D für Mathematik that the CAS system (a standalone Fortran program together with a database of character tables, see [NPP84]) should not be developed further and the functionality of CAS should be made available in GAP. The background was that extending CAS (by new Fortran code) had turned out to be much less flexible than writing analogous GAP library code.

For integrating the existing character theory algorithms, GAP's memory management and long integer arithmetic were useful as well as the list handling –it is an important feature of character theoretic methods that questions about groups are translated into manipulations of lists; on the other hand, the datatype of cyclotomics (see Chapter `Cyclotomics` (18.1.2)) was added to the GAP kernel because of the character theory algorithms. For developing further code, also other areas of GAP's library became interesting, such as permutation groups, finite fields, and polynomials.

The development of character theory code for GAP has been supported by several DFG grants, in particular the project "Representation Theory of Finite Groups and Finite Dimensional Algebras" (until 1991), and the Schwerpunkt "Algorithmische Zahlentheorie und Algebra" (from 1991 until 1997). Besides that, several Diploma theses at Lehrstuhl D were concerned with the development and/or implementation of algorithms dealing with characters in GAP.

The major contributions can be listed as follows.

- The arithmetic for the cyclotomics data type, following [Zum89], was first implemented by Marco van Meegen; an alternative approach was studied in the diploma thesis of Michael Scherner (see [Sch92]) but was not efficient enough; later Martin Schönert replaced the implementation by a better one.
- The basic routines for characters and character tables were written by Thomas Breuer and Götz Pfeiffer.
- The lattice related functions, such as `LLL` (72.10.4), `OrthogonalEmbeddings` (25.6.1), and `DnLattice` (72.10.8), were implemented by Ansgar Kaup (see [Kau92]).
- Functions for computing possible class fusions, possible power maps, and table automorphisms were written by Thomas Breuer (see [Bre91]).
- Functions for computing possible permutation characters were written by Thomas Breuer (see [Bre91]) and Götz Pfeiffer (see [Pfe91]).
- Functions for computing character tables from groups were written by Alexander Hulpke (Dixon-Schneider algorithm, see [Hul93]) and Hans Ulrich Besche (Baum algorithm and Conlon algorithm, see [Bes92]).
- Functions for dealing with Clifford matrices were written by Ute Schiffer (see [Sch94]).

- Functions for monomiality questions were written by Thomas Breuer and Erzsébet Horváth.

Since then, the code has been maintained and extended further by Alexander Hulpke (code related to his implementation of the Dixon-Schneider algorithm) and Thomas Breuer.

Currently GAP does not provide special functionality for computing Brauer character tables, but there is an interface to the MOC system (see [HJLP]), and the GAP Character Table Library contains many known Brauer character tables.

71.3 Creating Character Tables

There are in general five different ways to get a character table in GAP. You can

1. compute the table from a group,
2. read a file that contains the table data,
3. construct the table using generic formulae,
4. derive it from known character tables, or
5. combine partial information about conjugacy classes, power maps of the group in question, and about (character tables of) some subgroups and supergroups.

In 1., the computation of the irreducible characters is the hardest part; the different algorithms available for this are described in 71.14. Possibility 2. is used for the character tables in the GAP Character Table Library, see the manual of this library. Generic character tables –as addressed by 3.– are described in (**CTblLib: Generic Character Tables**). Several occurrences of 4. are described in 71.20. The last of the above possibilities *is currently not supported and will be described in a chapter of its own when it becomes available*.

The operation `CharacterTable` (71.3.1) can be used for the cases 1. to 3.

71.3.1 CharacterTable

▷ `CharacterTable(`*`G[, p]`*`)` (operation)
▷ `CharacterTable(`*`ordtbl, p`*`)` (operation)
▷ `CharacterTable(`*`name[, param]`*`)` (operation)

Called with a group *`G`*, `CharacterTable` calls the attribute `OrdinaryCharacterTable` (71.8.4). Called with first argument a group *`G`* or an ordinary character table *`ordtbl`*, and second argument a prime *`p`*, `CharacterTable` calls the operation `BrauerTable` (71.3.2).

Called with a string *`name`* and perhaps optional parameters *`param`*, `CharacterTable` tries to access a character table from the GAP Character Table Library. See the manual of the GAP package CTblLib for an overview of admissible arguments. An error is signalled if this GAP package is not loaded in this case.

Probably the most interesting information about the character table is its list of irreducibles, which can be accessed as the value of the attribute `Irr` (71.8.2). If the argument of `CharacterTable` is a string *`name`* then the irreducibles are just read from the library file, therefore the returned table stores them already. However, if `CharacterTable` is called with a group *`G`* or with an ordinary character table *`ordtbl`*, the irreducible characters are *not* computed by `CharacterTable`. They are

only computed when the Irr (71.8.2) value is accessed for the first time, for example when Display (6.3.6) is called for the table (see 71.13). This means for example that CharacterTable returns its result very quickly, and the first call of Display (6.3.6) for this table may take some time because the irreducible characters must be computed at that time before they can be displayed together with other information stored on the character table. The value of the filter HasIrr indicates whether the irreducible characters have been computed already.

The reason why CharacterTable does not compute the irreducible characters is that there are situations where one only needs the "table head", that is, the information about class lengths, power maps etc., but not the irreducibles. For example, if one wants to inspect permutation characters of a group then all one has to do is to induce the trivial characters of subgroups one is interested in; for that, only class lengths and the class fusion are needed. Or if one wants to compute the Molien series (see MolienSeries (72.12.1)) for a given complex matrix group, the irreducible characters of this group are in general of no interest.

For details about different algorithms to compute the irreducible characters, see 71.14.

If the group *G* is given as an argument, CharacterTable accesses the conjugacy classes of *G* and therefore causes that these classes are computed if they were not yet stored (see 71.6).

71.3.2 BrauerTable

▷ BrauerTable(*ordtbl*, *p*) (operation)
▷ BrauerTable(*G*, *p*) (operation)
▷ BrauerTableOp(*ordtbl*, *p*) (operation)
▷ ComputedBrauerTables(*ordtbl*) (attribute)

Called with an ordinary character table *ordtbl* or a group *G*, BrauerTable returns its *p*-modular character table if GAP can compute this table, and fail otherwise. The *p*-modular table can be computed for *p*-solvable groups (using the Fong-Swan Theorem) and in the case that *ordtbl* is a table from the GAP character table library for which also the *p*-modular table is contained in the table library.

The default method for a group and a prime delegates to BrauerTable for the ordinary character table of this group. The default method for *ordtbl* uses the attribute ComputedBrauerTables for storing the computed Brauer table at position *p*, and calls the operation BrauerTableOp for computing values that are not yet known.

So if one wants to install a new method for computing Brauer tables then it is sufficient to install it for BrauerTableOp.

The mod operator for a character table and a prime (see 71.7) delegates to BrauerTable.

71.3.3 CharacterTableRegular

▷ CharacterTableRegular(*tbl*, *p*) (function)

For an ordinary character table *tbl* and a prime integer *p*, CharacterTableRegular returns the "table head" of the *p*-modular Brauer character table of *tbl*. This is the restriction of *tbl* to its *p*-regular classes, like the return value of BrauerTable (71.3.2), but without the irreducible Brauer characters. (In general, these characters are hard to compute, and BrauerTable (71.3.2) may return fail for the given arguments, for example if *tbl* is a table from the GAP character table library.)

The returned table head can be used to create p-modular Brauer characters, by restricting ordinary characters, for example when one is interested in approximations of the (unknown) irreducible Brauer characters.

Example

```
gap> g:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> tbl:= CharacterTable( g );;  HasIrr( tbl );
false
gap> tblmod2:= CharacterTable( tbl, 2 );
BrauerTable( Sym( [ 1 .. 4 ] ), 2 )
gap> tblmod2 = CharacterTable( tbl, 2 );
true
gap> tblmod2 = BrauerTable( tbl, 2 );
true
gap> tblmod2 = BrauerTable( g, 2 );
true
gap> libtbl:= CharacterTable( "M" );
CharacterTable( "M" )
gap> CharacterTableRegular( libtbl, 2 );
BrauerTable( "M", 2 )
gap> BrauerTable( libtbl, 2 );
fail
gap> CharacterTable( "Symmetric", 4 );
CharacterTable( "Sym(4)" )
gap> ComputedBrauerTables( tbl );
[ , BrauerTable( Sym( [ 1 .. 4 ] ), 2 ) ]
```

71.3.4 SupportedCharacterTableInfo

▷ SupportedCharacterTableInfo (global variable)

SupportedCharacterTableInfo is a list that contains at position $3i-2$ an attribute getter function, at position $3i-1$ the name of this attribute, and at position $3i$ a list containing one or two of the strings "class", "character", depending on whether the attribute value relies on the ordering of classes or characters. This allows one to set exactly the components with these names in the record that is later converted to the new table, in order to use the values as attribute values. So the record components that shall *not* be regarded as attribute values can be ignored. Also other attributes of the old table are ignored.

SupportedCharacterTableInfo is used when (ordinary or Brauer) character table objects are created from records, using ConvertToCharacterTable (71.3.5).

New attributes and properties can be notified to SupportedCharacterTableInfo by creating them with DeclareAttributeSuppCT and DeclarePropertySuppCT instead of DeclareAttribute (79.18.3) and DeclareProperty (79.18.4).

71.3.5 ConvertToCharacterTable

▷ ConvertToCharacterTable(*record*) (function)
▷ ConvertToCharacterTableNC(*record*) (function)

Let *record* be a record. ConvertToCharacterTable converts *record* into a component object (see 79.10) representing a character table. The values of those components of *record* whose names occur in SupportedCharacterTableInfo (71.3.4) correspond to attribute values of the returned character table. All other components of the record simply become components of the character table object.

If inconsistencies in *record* are detected, fail is returned. *record* must have the component UnderlyingCharacteristic bound (cf. UnderlyingCharacteristic (71.9.5)), since this decides about whether the returned character table lies in IsOrdinaryTable (71.4.1) or in IsBrauerTable (71.4.1).

ConvertToCharacterTableNC does the same except that all checks of *record* are omitted.

An example of a conversion from a record to a character table object can be found in Section PrintCharacterTable (71.13.5).

71.4 Character Table Categories

71.4.1 IsNearlyCharacterTable

▷ IsNearlyCharacterTable(*obj*) (Category)
▷ IsCharacterTable(*obj*) (Category)
▷ IsOrdinaryTable(*obj*) (Category)
▷ IsBrauerTable(*obj*) (Category)
▷ IsCharacterTableInProgress(*obj*) (Category)

Every "character table like object" in GAP lies in the category IsNearlyCharacterTable. There are four important subcategories, namely the *ordinary* tables in IsOrdinaryTable, the *Brauer* tables in IsBrauerTable, the union of these two in IsCharacterTable, and the *incomplete ordinary* tables in IsCharacterTableInProgress.

We want to distinguish ordinary and Brauer tables because a Brauer table may delegate tasks to the ordinary table of the same group, for example the computation of power maps. A Brauer table is constructed from an ordinary table and stores this table upon construction (see OrdinaryCharacterTable (71.8.4)).

Furthermore, IsOrdinaryTable and IsBrauerTable denote character tables that provide enough information to compute all power maps and irreducible characters (and in the case of Brauer tables to get the ordinary table), for example because the underlying group (see UnderlyingGroup (71.6.1)) is known or because the table is a library table (see the manual of the GAP Character Table Library). We want to distinguish these tables from partially known ordinary tables that cannot be asked for all power maps or all irreducible characters.

The character table objects in IsCharacterTable are always immutable (see 12.6). This means mainly that the ordering of conjugacy classes used for the various attributes of the character table cannot be changed; see 71.21 for how to compute a character table with a different ordering of classes.

The GAP objects in IsCharacterTableInProgress represent incomplete ordinary character tables. This means that not all irreducible characters, not all power maps are known, and perhaps even the number of classes and the centralizer orders are known. Such tables occur when the character table of a group G is constructed using character tables of related groups and information about G but for example without explicitly computing the conjugacy classes of G. An object in IsCharacterTableInProgress is first of all *mutable*, so *nothing is stored automatically* on such a table, since otherwise one has no control of side-effects when a hypothesis is changed. Operations

for such tables may return more general values than for other tables, for example class functions may contain unknowns (see Chapter 74) or lists of possible values in certain positions, the same may happen also for power maps and class fusions (see 73.5). *Incomplete tables in this sense are currently not supported and will be described in a chapter of their own when they become available.* Note that the term "incomplete table" shall express that GAP cannot compute certain values such as irreducible characters or power maps. A table with access to its group is therefore always complete, also if its irreducible characters are not yet stored.

Example

```
gap> g:= SymmetricGroup( 4 );;
gap> tbl:= CharacterTable( g );  modtbl:= tbl mod 2;
CharacterTable( Sym( [ 1 .. 4 ] ) )
BrauerTable( Sym( [ 1 .. 4 ] ), 2 )
gap> IsCharacterTable( tbl );  IsCharacterTable( modtbl );
true
true
gap> IsBrauerTable( modtbl );  IsBrauerTable( tbl );
true
false
gap> IsOrdinaryTable( tbl );  IsOrdinaryTable( modtbl );
true
false
gap> IsCharacterTable( g );  IsCharacterTable( Irr( g ) );
false
false
```

71.4.2 InfoCharacterTable

▷ `InfoCharacterTable` (info class)

is the info class (see 7.4) for computations with character tables.

71.4.3 NearlyCharacterTablesFamily

▷ `NearlyCharacterTablesFamily` (global variable)

Every character table like object lies in this family (see 13.1).

71.5 Conventions for Character Tables

The following few conventions should be noted.

- The class of the *identity element* is expected to be the first one; thus the degree of a character is the character value at position 1.
- The *trivial character* of a character table need not be the first in the list of irreducibles.
- Most functions that take a character table as an argument and work with characters expect these characters as an argument, too. For some functions, the list of irreducible characters serves as the default, i.e, the value of the attribute `Irr` (71.8.2); in these cases, the `Irr` (71.8.2) value is automatically computed if it was not yet known.

- For a stored class fusion, the image table is denoted by its `Identifier` (71.9.8) value; each library table has a unique identifier by which it can be accessed (see **(CTblLib: Accessing a Character Table from the Library)** in the manual for the GAP Character Table Library), tables constructed from groups get an identifier that is unique in the current GAP session.

71.6 The Interface between Character Tables and Groups

For a character table with underlying group (see `UnderlyingGroup` (71.6.1)), the interface between table and group consists of three attribute values, namely the *group*, the *conjugacy classes* stored in the table (see `ConjugacyClasses` (71.6.2) below) and the *identification* of the conjugacy classes of table and group (see `IdentificationOfConjugacyClasses` (71.6.3) below).

Character tables constructed from groups know these values upon construction, and for character tables constructed without groups, these values are usually not known and cannot be computed from the table.

However, given a group G and a character table of a group isomorphic to G (for example a character table from the GAP table library), one can tell GAP to compute a new instance of the given table and to use it as the character table of G (see `CharacterTableWithStoredGroup` (71.6.4)).

Tasks may be delegated from a group to its character table or vice versa only if these three attribute values are stored in the character table.

71.6.1 UnderlyingGroup (for character tables)

▷ `UnderlyingGroup(`*`ordtbl`*`)` (attribute)

For an ordinary character table *`ordtbl`* of a finite group, the group can be stored as value of `UnderlyingGroup`.

Brauer tables do not store the underlying group, they access it via the ordinary table (see `OrdinaryCharacterTable` (71.8.4)).

71.6.2 ConjugacyClasses (for character tables)

▷ `ConjugacyClasses(`*`tbl`*`)` (attribute)

For a character table *`tbl`* with known underlying group G, the `ConjugacyClasses` value of *`tbl`* is a list of conjugacy classes of G. All those lists stored in the table that are related to the ordering of conjugacy classes (such as sizes of centralizers and conjugacy classes, orders of representatives, power maps, and all class functions) refer to the ordering of this list.

This ordering need *not* coincide with the ordering of conjugacy classes as stored in the underlying group of the table (see 71.21). One reason for this is that otherwise we would not be allowed to use a library table as the character table of a group for which the conjugacy classes are stored already. (Another, less important reason is that we can use the same group as underlying group of character tables that differ only w.r.t. the ordering of classes.)

The class of the identity element must be the first class (see 71.5).

If *`tbl`* was constructed from G then the conjugacy classes have been stored at the same time when G was stored. If G and *`tbl`* have been connected later than in the construction of *`tbl`*, the recommended way to do this is via `CharacterTableWithStoredGroup` (71.6.4). So there is no method for `ConjugacyClasses` that computes the value for *`tbl`* if it is not yet stored.

Brauer tables do not store the (p-regular) conjugacy classes, they access them via the ordinary table (see OrdinaryCharacterTable (71.8.4)) if necessary.

71.6.3 IdentificationOfConjugacyClasses

▷ IdentificationOfConjugacyClasses(*tbl*) (attribute)

For an ordinary character table *tbl* with known underlying group G, IdentificationOfConjugacyClasses returns a list of positive integers that contains at position i the position of the i-th conjugacy class of *tbl* in the ConjugacyClasses (71.6.2) value of G.

Example

```
gap> g:= SymmetricGroup( 4 );;
gap> repres:= [ (1,2), (1,2,3), (1,2,3,4), (1,2)(3,4), () ];;
gap> ccl:= List( repres, x -> ConjugacyClass( g, x ) );;
gap> SetConjugacyClasses( g, ccl );
gap> tbl:= CharacterTable( g );;   # the table stores already the values
gap> HasConjugacyClasses( tbl );  HasUnderlyingGroup( tbl );
true
true
gap> UnderlyingGroup( tbl ) = g;
true
gap> HasIdentificationOfConjugacyClasses( tbl );
true
gap> IdentificationOfConjugacyClasses( tbl );
[ 5, 1, 2, 3, 4 ]
```

71.6.4 CharacterTableWithStoredGroup

▷ CharacterTableWithStoredGroup(*G, tbl[, info]*) (function)

Let *G* be a group and *tbl* a character table of (a group isomorphic to) *G*, such that *G* does not store its OrdinaryCharacterTable (71.8.4) value. CharacterTableWithStoredGroup calls CompatibleConjugacyClasses (71.6.5), trying to identify the classes of *G* with the columns of *tbl*.

If this identification is unique up to automorphisms of *tbl* (see AutomorphismsOfTable (71.9.4)) then *tbl* is stored as CharacterTable (71.3.1) value of *G*, and a new character table is returned that is equivalent to *tbl*, is sorted in the same way as *tbl*, and has the values of UnderlyingGroup (71.6.1), ConjugacyClasses (71.6.2), and IdentificationOfConjugacyClasses (71.6.3) set.

Otherwise, i.e., if GAP cannot identify the classes of *G* up to automorphisms of *tbl*, fail is returned.

If a record is present as the third argument *info*, its meaning is the same as the optional argument *arec* for CompatibleConjugacyClasses (71.6.5).

If a list is entered as third argument *info* it is used as value of IdentificationOfConjugacyClasses (71.6.3), relative to the ConjugacyClasses (71.6.2) value of *G*, without further checking, and the corresponding character table is returned.

71.6.5 CompatibleConjugacyClasses

▷ CompatibleConjugacyClasses(*[G, ccl,]tbl[, arec]*) (operation)

If the arguments *G* and *ccl* are present then *ccl* must be a list of the conjugacy classes of the group *G*, and *tbl* the ordinary character table of *G*. Then CompatibleConjugacyClasses returns a list l of positive integers that describes an identification of the columns of *tbl* with the conjugacy classes *ccl* in the sense that $l[i]$ is the position in *ccl* of the class corresponding to the i-th column of *tbl*, if this identification is unique up to automorphisms of *tbl* (see AutomorphismsOfTable (71.9.4)); if GAP cannot identify the classes, fail is returned.

If *tbl* is the first argument then it must be an ordinary character table, and CompatibleConjugacyClasses checks whether the columns of *tbl* can be identified with the conjugacy classes of a group isomorphic to that for which *tbl* is the character table; the return value is a list of all those sets of class positions for which the columns of *tbl* cannot be distinguished with the invariants used, up to automorphisms of *tbl*. So the identification is unique if and only if the returned list is empty.

The usual approach is that one first calls CompatibleConjugacyClasses in the second form for checking quickly whether the first form will be successful, and only if this is the case the more time consuming calculations with both group and character table are done.

The following invariants are used.

1. element orders (see OrdersClassRepresentatives (71.9.1)),
2. class lengths (see SizesConjugacyClasses (71.9.3)),
3. power maps (see PowerMap (73.1.1), ComputedPowerMaps (73.1.1)),
4. symmetries of the table (see AutomorphismsOfTable (71.9.4)).

If the optional argument *arec* is present then it must be a record whose components describe additional information for the class identification. The following components are supported.

natchar
: if G is a permutation group or matrix group then the value of this component is regarded as the list of values of the natural character (see NaturalCharacter (72.7.2)) of *G*, w.r.t. the ordering of classes in *tbl*,

bijection
: a list describing a partial bijection; the i-th entry, if bound, is the position of the i-th conjugacy class of *tbl* in the list *ccl*.

Example

```
gap> g:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
gap> tbl:= CharacterTable( "A5" );
CharacterTable( "A5" )
gap> HasUnderlyingGroup( tbl );  HasOrdinaryCharacterTable( g );
false
false
gap> CompatibleConjugacyClasses( tbl );   # unique identification
[ ]
gap> new:= CharacterTableWithStoredGroup( g, tbl );
```

```
CharacterTable( Alt( [ 1 .. 5 ] ) )
gap> Irr( new ) = Irr( tbl );
true
gap> HasConjugacyClasses( new );  HasUnderlyingGroup( new );
true
true
gap> IdentificationOfConjugacyClasses( new );
[ 1, 2, 3, 4, 5 ]
gap> # Here is an example where the identification is not unique.
gap> CompatibleConjugacyClasses( CharacterTable( "J2" ) );
[ [ 17, 18 ], [ 9, 10 ] ]
```

71.7 Operators for Character Tables

The following infix operators are defined for character tables.

tbl1 `*` *tbl2*
: the direct product of two character tables (see `CharacterTableDirectProduct` (71.20.1)),

tbl `/` *list*
: the table of the factor group modulo the normal subgroup spanned by the classes in the list *list* (see `CharacterTableFactorGroup` (71.20.3)),

tbl `mod` *p*
: the *p*-modular Brauer character table corresponding to the ordinary character table *tbl* (see `BrauerTable` (71.3.2)),

tbl`.`*name*
: the position of the class with name *name* in *tbl* (see `ClassNames` (71.9.6)).

71.8 Attributes and Properties for Groups and Character Tables

Several *attributes for groups* are valid also for character tables.

These are first those that have the same meaning for both the group and its character table, and whose values can be read off or computed, respectively, from the character table, such as `Size` (71.8.5), `IsAbelian` (71.8.5), or `IsSolvable` (71.8.5).

Second, there are attributes whose meaning for character tables is different from the meaning for groups, such as `ConjugacyClasses` (71.6.2).

71.8.1 CharacterDegrees

▷ `CharacterDegrees(`*G[, p]*`)` (attribute)
▷ `CharacterDegrees(`*tbl*`)` (attribute)

In the first form, `CharacterDegrees` returns a collected list of the degrees of the absolutely irreducible characters of the group *G*; the optional second argument *p* must be either zero or a prime integer denoting the characteristic, the default value is zero. In the second form, *tbl* must be an

(ordinary or Brauer) character table, and CharacterDegrees returns a collected list of the degrees of the absolutely irreducible characters of *tbl*.

(The default method for the call with only argument a group is to call the operation with second argument 0.)

For solvable groups, the default method is based on [Con90b].

Example

```
gap> CharacterDegrees( SymmetricGroup( 4 ) );
[ [ 1, 2 ], [ 2, 1 ], [ 3, 2 ] ]
gap> CharacterDegrees( SymmetricGroup( 4 ), 2 );
[ [ 1, 1 ], [ 2, 1 ] ]
gap> CharacterDegrees( CharacterTable( "A5" ) );
[ [ 1, 1 ], [ 3, 2 ], [ 4, 1 ], [ 5, 1 ] ]
gap> CharacterDegrees( CharacterTable( "A5" ) mod 2 );
[ [ 1, 1 ], [ 2, 2 ], [ 4, 1 ] ]
```

71.8.2 Irr

▷ Irr(*G*[, *p*]) (attribute)

▷ Irr(*tbl*) (attribute)

Called with a group *G*, Irr returns the irreducible characters of the ordinary character table of *G*. Called with a group *G* and a prime integer *p*, Irr returns the irreducible characters of the *p*-modular Brauer table of *G*. Called with an (ordinary or Brauer) character table *tbl*, Irr returns the list of all complex absolutely irreducible characters of *tbl*.

For a character table *tbl* with underlying group, Irr may delegate to the group. For a group *G*, Irr may delegate to its character table only if the irreducibles are already stored there.

(If *G* is *p*-solvable (see IsPSolvable (39.15.23)) then the *p*-modular irreducible characters can be computed by the Fong-Swan Theorem; in all other cases, there may be no method.)

Note that the ordering of columns in the Irr matrix of the group *G* refers to the ordering of conjugacy classes in the CharacterTable (71.3.1) value of *G*, which may differ from the ordering of conjugacy classes in *G* (see 71.6). As an extreme example, for a character table obtained from sorting the classes of the CharacterTable (71.3.1) value of *G*, the ordering of columns in the Irr matrix respects the sorting of classes (see 71.21), so the irreducibles of such a table will in general not coincide with the irreducibles stored as the Irr value of *G* although also the sorted table stores the group *G*.

The ordering of the entries in the attribute Irr of a group need *not* coincide with the ordering of its IrreducibleRepresentations (71.14.4) value.

Example

```
gap> Irr( SymmetricGroup( 4 ) );
[ Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, -1, 1, 1, -1
     ] ), Character( CharacterTable( Sym( [ 1 .. 4 ] ) ),
    [ 3, -1, -1, 0, 1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 2, 0, 2, -1, 0 ] )
    , Character( CharacterTable( Sym( [ 1 .. 4 ] ) ),
    [ 3, 1, -1, 0, -1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] )
 ]
gap> Irr( SymmetricGroup( 4 ), 2 );
[ Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 1, 1 ] ),
```

```
    Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 2, -1 ] ) ]
gap> Irr( CharacterTable( "A5" ) );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
  Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ]
gap> Irr( CharacterTable( "A5" ) mod 2 );
[ Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] ),
  Character( BrauerTable( "A5", 2 ),
    [ 2, -1, E(5)+E(5)^4, E(5)^2+E(5)^3 ] ),
  Character( BrauerTable( "A5", 2 ),
    [ 2, -1, E(5)^2+E(5)^3, E(5)+E(5)^4 ] ),
  Character( BrauerTable( "A5", 2 ), [ 4, 1, -1, -1 ] ) ]
```

71.8.3 LinearCharacters

▷ `LinearCharacters(`*G[, p]*`)` (attribute)
▷ `LinearCharacters(`*tbl*`)` (attribute)

`LinearCharacters` returns the linear (i.e., degree 1) characters in the `Irr` (71.8.2) list of the group *G* or the character table *tbl*, respectively. In the second form, `LinearCharacters` returns the *p*-modular linear characters of the group *G*.

For a character table *tbl* with underlying group, `LinearCharacters` may delegate to the group. For a group *G*, `LinearCharacters` may delegate to its character table only if the irreducibles are already stored there.

The ordering of linear characters in *tbl* need not coincide with the ordering of linear characters in the irreducibles of *tbl* (see `Irr` (71.8.2)).

Example

```
gap> LinearCharacters( SymmetricGroup( 4 ) );
[ Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, -1, 1, 1, -1
    ] ) ]
```

71.8.4 OrdinaryCharacterTable

▷ `OrdinaryCharacterTable(`*G*`)` (attribute)
▷ `OrdinaryCharacterTable(`*modtbl*`)` (attribute)

`OrdinaryCharacterTable` returns the ordinary character table of the group *G* or the Brauer character table *modtbl*, respectively.

Since Brauer character tables are constructed from ordinary tables, the attribute value for *modtbl* is already stored (cf. 71.4).

Example

```
gap> OrdinaryCharacterTable( SymmetricGroup( 4 ) );
CharacterTable( Sym( [ 1 .. 4 ] ) )
gap> tbl:= CharacterTable( "A5" );;  modtbl:= tbl mod 2;
```

```
BrauerTable( "A5", 2 )
gap> OrdinaryCharacterTable( modtbl ) = tbl;
true
```

71.8.5 Group Operations Applicable to Character Tables

▷ AbelianInvariants(*tbl*) (attribute)
▷ CommutatorLength(*tbl*) (attribute)
▷ Exponent(*tbl*) (attribute)
▷ IsAbelian(*tbl*) (property)
▷ IsAlmostSimple(*tbl*) (property)
▷ IsCyclic(*tbl*) (property)
▷ IsElementaryAbelian(*tbl*) (property)
▷ IsFinite(*tbl*) (property)
▷ IsMonomial(*tbl*) (property)
▷ IsNilpotent(*tbl*) (property)
▷ IsPerfect(*tbl*) (property)
▷ IsSimple(*tbl*) (property)
▷ IsSolvable(*tbl*) (property)
▷ IsSporadicSimple(*tbl*) (property)
▷ IsSupersolvable(*tbl*) (property)
▷ IsomorphismTypeInfoFiniteSimpleGroup(*tbl*) (attribute)
▷ NrConjugacyClasses(*tbl*) (attribute)
▷ Size(*tbl*) (attribute)

These operations for groups are applicable to character tables and mean the same for a character table as for its underlying group; see Chapter 39 for the definitions. The operations are mainly useful for selecting character tables with certain properties, also for character tables without access to a group.

Example

```
gap> tables:= [ CharacterTable( CyclicGroup( 3 ) ),
>               CharacterTable( SymmetricGroup( 4 ) ),
>               CharacterTable( AlternatingGroup( 5 ) ) ];;
gap> List( tables, AbelianInvariants );
[ [ 3 ], [ 2 ], [  ] ]
gap> List( tables, CommutatorLength );
[ 1, 1, 1 ]
gap> List( tables, Exponent );
[ 3, 12, 30 ]
gap> List( tables, IsAbelian );
[ true, false, false ]
gap> List( tables, IsAlmostSimple );
[ false, false, true ]
gap> List( tables, IsCyclic );
[ true, false, false ]
gap> List( tables, IsFinite );
[ true, true, true ]
gap> List( tables, IsMonomial );
[ true, true, false ]
```

```
gap> List( tables, IsNilpotent );
[ true, false, false ]
gap> List( tables, IsPerfect );
[ false, false, true ]
gap> List( tables, IsSimple );
[ true, false, true ]
gap> List( tables, IsSolvable );
[ true, true, false ]
gap> List( tables, IsSupersolvable );
[ true, false, false ]
gap> List( tables, NrConjugacyClasses );
[ 3, 5, 5 ]
gap> List( tables, Size );
[ 3, 24, 60 ]
gap> IsomorphismTypeInfoFiniteSimpleGroup( CharacterTable( "C5" ) );
rec( name := "Z(5)", parameter := 5, series := "Z" )
gap> IsomorphismTypeInfoFiniteSimpleGroup( CharacterTable( "S3" ) );
fail
gap> IsomorphismTypeInfoFiniteSimpleGroup( CharacterTable( "S6(3)" ) );
rec( name := "C(3,3) = S(6,3)", parameter := [ 3, 3 ], series := "C" )
gap> IsomorphismTypeInfoFiniteSimpleGroup( CharacterTable( "O7(3)" ) );
rec( name := "B(3,3) = O(7,3)", parameter := [ 3, 3 ], series := "B" )
gap> IsomorphismTypeInfoFiniteSimpleGroup( CharacterTable( "A8" ) );
rec( name := "A(8) ~ A(3,2) = L(4,2) ~ D(3,2) = O+(6,2)",
  parameter := 8, series := "A" )
gap> IsomorphismTypeInfoFiniteSimpleGroup( CharacterTable( "L3(4)" ) );
rec( name := "A(2,4) = L(3,4)", parameter := [ 3, 4 ], series := "L" )
```

71.9 Attributes and Properties only for Character Tables

The following three *attributes for character tables* –`OrdersClassRepresentatives` (71.9.1), `SizesCentralizers` (71.9.2), and `SizesConjugacyClasses` (71.9.3)– would make sense also for groups but are in fact *not* used for groups. This is because the values depend on the ordering of conjugacy classes stored as the value of `ConjugacyClasses` (71.6.2), and this value may differ for a group and its character table (see 71.6). Note that for character tables, the consistency of attribute values must be guaranteed, whereas for groups, there is no need to impose such a consistency rule.

The other attributes introduced in this section apply only to character tables, not to groups.

71.9.1 OrdersClassRepresentatives

▷ `OrdersClassRepresentatives(`*`tbl`*`)` (attribute)

is a list of orders of representatives of conjugacy classes of the character table *`tbl`*, in the same ordering as the conjugacy classes of *`tbl`*.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> OrdersClassRepresentatives( tbl );
[ 1, 2, 3, 5, 5 ]
```

71.9.2 SizesCentralizers

▷ SizesCentralizers(*tbl*) (attribute)
▷ SizesCentralisers(*tbl*) (attribute)

is a list that stores at position *i* the size of the centralizer of any element in the *i*-th conjugacy class of the character table *tbl*.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> SizesCentralizers( tbl );
[ 60, 4, 3, 5, 5 ]
```

71.9.3 SizesConjugacyClasses

▷ SizesConjugacyClasses(*tbl*) (attribute)

is a list that stores at position *i* the size of the *i*-th conjugacy class of the character table *tbl*.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> SizesConjugacyClasses( tbl );
[ 1, 15, 20, 12, 12 ]
```

71.9.4 AutomorphismsOfTable

▷ AutomorphismsOfTable(*tbl*) (attribute)

is the permutation group of all column permutations of the character table *tbl* that leave the set of irreducibles and each power map of *tbl* invariant (see also TableAutomorphisms (71.22.2)).

Example

```
gap> tbl:= CharacterTable( "Dihedral", 8 );;
gap> AutomorphismsOfTable( tbl );
Group([ (4,5) ])
gap> OrdersClassRepresentatives( tbl );
[ 1, 4, 2, 2, 2 ]
gap> SizesConjugacyClasses( tbl );
[ 1, 2, 1, 2, 2 ]
```

71.9.5 UnderlyingCharacteristic

▷ UnderlyingCharacteristic(*tbl*) (attribute)
▷ UnderlyingCharacteristic(*psi*) (attribute)

For an ordinary character table *tbl*, the result is 0, for a *p*-modular Brauer table *tbl*, it is *p*. The underlying characteristic of a class function *psi* is equal to that of its underlying character table.

The underlying characteristic must be stored when the table is constructed, there is no method to compute it.

We cannot use the attribute Characteristic (31.10.1) to denote this, since of course each Brauer character is an element of characteristic zero in the sense of GAP (see Chapter 72).

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> UnderlyingCharacteristic( tbl );
0
gap> UnderlyingCharacteristic( tbl mod 17 );
17
```

71.9.6 Class Names and Character Names

▷ `ClassNames(`*`tbl[, "ATLAS"]`*`)` (attribute)
▷ `CharacterNames(`*`tbl`*`)` (attribute)

`ClassNames` and `CharacterNames` return lists of strings, one for each conjugacy class or irreducible character, respectively, of the character table *tbl*. These names are used when *tbl* is displayed.

The default method for `ClassNames` computes class names consisting of the order of an element in the class and at least one distinguishing letter.

The default method for `CharacterNames` returns the list `[ "X.1", "X.2", ... ]`, whose length is the number of irreducible characters of *tbl*.

The position of the class with name *name* in *tbl* can be accessed as *tbl*`.`*name*.

When `ClassNames` is called with two arguments, the second being the string `"ATLAS"`, the class names returned obey the convention used in the Atlas of Finite Groups [CCN+85, Chapter 7, Section 5]. If one is interested in "relative" class names of almost simple Atlas groups, one can use the function `AtlasClassNames` (**AtlasRep: AtlasClassNames**).

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> ClassNames( tbl );
[ "1a", "2a", "3a", "5a", "5b" ]
gap> tbl.2a;
2
```

71.9.7 Class Parameters and Character Parameters

▷ `ClassParameters(`*`tbl`*`)` (attribute)
▷ `CharacterParameters(`*`tbl`*`)` (attribute)

The values of these attributes are lists containing a parameter for each conjugacy class or irreducible character, respectively, of the character table *tbl*.

It depends on *tbl* what these parameters are, so there is no default to compute class and character parameters.

For example, the classes of symmetric groups can be parametrized by partitions, corresponding to the cycle structures of permutations. Character tables constructed from generic character tables (see the manual of the GAP Character Table Library) usually have class and character parameters stored.

If *tbl* is a p-modular Brauer table such that class parameters are stored in the underlying ordinary table (see `OrdinaryCharacterTable` (71.8.4)) of *tbl* then `ClassParameters` returns the sublist of class parameters of the ordinary table, for p-regular classes.

71.9.8 Identifier (for character tables)

▷ Identifier(*tbl*) (attribute)

is a string that identifies the character table *tbl* in the current GAP session. It is used mainly for class fusions into *tbl* that are stored on other character tables. For character tables without group, the identifier is also used to print the table; this is the case for library tables, but also for tables that are constructed as direct products, factors etc. involving tables that may or may not store their groups.

The default method for ordinary tables constructs strings of the form "CT*n*", where *n* is a positive integer. LARGEST_IDENTIFIER_NUMBER is a list containing the largest integer *n* used in the current GAP session.

The default method for Brauer tables returns the concatenation of the identifier of the ordinary table, the string "mod", and the (string of the) underlying characteristic.

Example

```
gap> Identifier( CharacterTable( "A5" ) );
"A5"
gap> tbl:= CharacterTable( Group( () ) );;
gap> Identifier( tbl );  Identifier( tbl mod 2 );
"CT8"
"CT8mod2"
```

71.9.9 InfoText

▷ InfoText(*tbl*) (attribute)

is a mutable string with information about the character table *tbl*. There is no default method to create an info text.

This attribute is used mainly for library tables (see the manual of the GAP Character Table Library). Usual parts of the information are the origin of the table, tests it has passed (1.o.r. for the test of orthogonality, pow[*p*] for the construction of the *p*-th power map, DEC for the decomposition of ordinary into Brauer characters, TENS for the decomposition of tensor products of irreducibles), and choices made without loss of generality.

Example

```
gap> Print( InfoText( CharacterTable( "A5" ) ), "\n" );
origin: ATLAS of finite groups, tests: 1.o.r., pow[2,3,5]
```

71.9.10 InverseClasses

▷ InverseClasses(*tbl*) (attribute)

For a character table *tbl*, InverseClasses returns the list mapping each conjugacy class to its inverse class. This list can be regarded as (-1)-st power map of *tbl* (see PowerMap (73.1.1)).

Example

```
gap> InverseClasses( CharacterTable( "A5" ) );
[ 1, 2, 3, 4, 5 ]
gap> InverseClasses( CharacterTable( "Cyclic", 3 ) );
[ 1, 3, 2 ]
```

71.9.11 RealClasses

▷ `RealClasses(tbl)` (attribute)

For a character table *tbl*, `RealClasses` returns the strictly sorted list of positions of classes in *tbl* that consist of real elements.

An element x is *real* iff it is conjugate to its inverse $x^{-1} = x^{o(x)-1}$.

Example

```
gap> RealClasses( CharacterTable( "A5" ) );
[ 1, 2, 3, 4, 5 ]
gap> RealClasses( CharacterTable( "Cyclic", 3 ) );
[ 1 ]
```

71.9.12 ClassOrbit

▷ `ClassOrbit(tbl, cc)` (operation)

is the list of positions of those conjugacy classes of the character table *tbl* that are Galois conjugate to the *cc*-th class. That is, exactly the classes at positions given by the list returned by `ClassOrbit` contain generators of the cyclic group generated by an element in the *cc*-th class.

This information is computed from the power maps of *tbl*.

Example

```
gap> ClassOrbit( CharacterTable( "A5" ), 4 );
[ 4, 5 ]
```

71.9.13 ClassRoots

▷ `ClassRoots(tbl)` (attribute)

For a character table *tbl*, `ClassRoots` returns a list containing at position i the list of positions of the classes of all nontrivial p-th roots, where p runs over the prime divisors of the `Size` (71.8.5) value of *tbl*.

This information is computed from the power maps of *tbl*.

Example

```
gap> ClassRoots( CharacterTable( "A5" ) );
[ [ 2, 3, 4, 5 ], [  ], [  ], [  ], [  ] ]
gap> ClassRoots( CharacterTable( "Cyclic", 6 ) );
[ [ 3, 4, 5 ], [  ], [ 2 ], [ 2, 6 ], [ 6 ], [  ] ]
```

71.10 Normal Subgroups Represented by Lists of Class Positions

The following attributes for a character table *tbl* correspond to attributes for the group G of *tbl*. But instead of a normal subgroup (or a list of normal subgroups) of G, they return a strictly sorted list of positive integers (or a list of such lists) which are the positions –relative to the `ConjugacyClasses` (71.6.2) value of *tbl*– of those classes forming the normal subgroup in question.

71.10.1 ClassPositionsOfNormalSubgroups

▷ `ClassPositionsOfNormalSubgroups(`*`ordtbl`*`)` (attribute)
▷ `ClassPositionsOfMaximalNormalSubgroups(`*`ordtbl`*`)` (attribute)
▷ `ClassPositionsOfMinimalNormalSubgroups(`*`ordtbl`*`)` (attribute)

correspond to `NormalSubgroups` (39.19.8), `MaximalNormalSubgroups` (39.19.9), `MinimalNormalSubgroups` (39.19.10) for the group of the ordinary character table *`ordtbl`*.

The entries of the result lists are sorted according to increasing length. (So this total order respects the partial order of normal subgroups given by inclusion.)

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfNormalSubgroups( tbls4 );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1 .. 5 ] ]
```

71.10.2 ClassPositionsOfAgemo

▷ `ClassPositionsOfAgemo(`*`ordtbl, p`*`)` (operation)

corresponds to `Agemo` (39.14.2) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfAgemo( tbls4, 2 );
[ 1, 3, 4 ]
```

71.10.3 ClassPositionsOfCentre (for a character table)

▷ `ClassPositionsOfCentre(`*`ordtbl`*`)` (attribute)
▷ `ClassPositionsOfCenter(`*`ordtbl`*`)` (attribute)

corresponds to `Centre` (35.4.5) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> ClassPositionsOfCentre( tbld8 );
[ 1, 3 ]
```

71.10.4 ClassPositionsOfDirectProductDecompositions

▷ `ClassPositionsOfDirectProductDecompositions(`*`tbl[, nclasses]`*`)` (attribute)

Let *`tbl`* be the ordinary character table of the group G, say. Called with the only argument *`tbl`*, `ClassPositionsOfDirectProductDecompositions` returns the list of all those pairs $[l_1, l_2]$ where l_1 and l_2 are lists of class positions of normal subgroups N_1, N_2 of G such that G is their direct product and $|N_1| \leq |N_2|$ holds. Called with second argument a list *`nclasses`* of class positions of a normal subgroup N of G, `ClassPositionsOfDirectProductDecompositions` returns the list of pairs describing the decomposition of N as a direct product of two normal subgroups of G.

71.10.5 ClassPositionsOfDerivedSubgroup

▷ `ClassPositionsOfDerivedSubgroup(`*`ordtbl`*`)` (attribute)

corresponds to `DerivedSubgroup` (39.12.3) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> ClassPositionsOfDerivedSubgroup( tbld8 );
[ 1, 3 ]
```

71.10.6 ClassPositionsOfElementaryAbelianSeries

▷ `ClassPositionsOfElementaryAbelianSeries(`*`ordtbl`*`)` (attribute)

corresponds to `ElementaryAbelianSeries` (39.17.9) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> tbla5:= CharacterTable( "A5" );;
gap> ClassPositionsOfElementaryAbelianSeries( tbls4 );
[ [ 1 .. 5 ], [ 1, 3, 4 ], [ 1, 3 ], [ 1 ] ]
gap> ClassPositionsOfElementaryAbelianSeries( tbla5 );
fail
```

71.10.7 ClassPositionsOfFittingSubgroup

▷ `ClassPositionsOfFittingSubgroup(`*`ordtbl`*`)` (attribute)

corresponds to `FittingSubgroup` (39.12.5) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfFittingSubgroup( tbls4 );
[ 1, 3 ]
```

71.10.8 ClassPositionsOfLowerCentralSeries

▷ `ClassPositionsOfLowerCentralSeries(`*`tbl`*`)` (attribute)

corresponds to `LowerCentralSeriesOfGroup` (39.17.11) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> ClassPositionsOfLowerCentralSeries( tbls4 );
[ [ 1 .. 5 ], [ 1, 3, 4 ] ]
gap> ClassPositionsOfLowerCentralSeries( tbld8 );
[ [ 1 .. 5 ], [ 1, 3 ], [ 1 ] ]
```

71.10.9 ClassPositionsOfUpperCentralSeries

▷ `ClassPositionsOfUpperCentralSeries(`*`ordtbl`*`)` (attribute)

corresponds to `UpperCentralSeriesOfGroup` (39.17.12) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> ClassPositionsOfUpperCentralSeries( tbls4 );
[ [ 1 ] ]
gap> ClassPositionsOfUpperCentralSeries( tbld8 );
[ [ 1, 3 ], [ 1, 2, 3, 4, 5 ] ]
```

71.10.10 ClassPositionsOfSupersolvableResiduum

▷ `ClassPositionsOfSupersolvableResiduum(`*`ordtbl`*`)` (attribute)

corresponds to `SupersolvableResiduum` (39.12.11) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfSupersolvableResiduum( tbls4 );
[ 1, 3 ]
```

71.10.11 ClassPositionsOfPCore

▷ `ClassPositionsOfPCore(`*`ordtbl, p`*`)` (operation)

corresponds to `PCore` (39.11.3) for the group of the ordinary character table *`ordtbl`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfPCore( tbls4, 2 );
[ 1, 3 ]
gap> ClassPositionsOfPCore( tbls4, 3 );
[ 1 ]
```

71.10.12 ClassPositionsOfNormalClosure

▷ `ClassPositionsOfNormalClosure(`*`ordtbl, classes`*`)` (operation)

is the sorted list of the positions of all conjugacy classes of the ordinary character table *`ordtbl`* that form the normal closure (see `NormalClosure` (39.11.4)) of the conjugacy classes at positions in the list *`classes`*.

Example

```
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfNormalClosure( tbls4, [ 1, 4 ] );
[ 1, 3, 4 ]
```

71.11 Operations Concerning Blocks

71.11.1 PrimeBlocks

▷ `PrimeBlocks(`*`ordtbl, p`*`)` (operation)
▷ `PrimeBlocksOp(`*`ordtbl, p`*`)` (operation)
▷ `ComputedPrimeBlockss(`*`tbl`*`)` (attribute)

For an ordinary character table *ordtbl* and a prime integer p, `PrimeBlocks` returns a record with the following components.

`block`
: a list, the value j at position i means that the i-th irreducible character of *ordtbl* lies in the j-th p-block of *ordtbl*,

`defect`
: a list containing at position i the defect of the i-th block,

`height`
: a list containing at position i the height of the i-th irreducible character of *ordtbl* in its block,

`relevant`
: a list of class positions such that only the restriction to these classes need be checked for deciding whether two characters lie in the same block, and

`centralcharacter`
: a list containing at position i a list whose values at the positions stored in the component `relevant` are the values of a central character in the i-th block.

The components `relevant` and `centralcharacters` are used by `SameBlock` (71.11.2).

If `InfoCharacterTable` (71.4.2) has level at least 2, the defects of the blocks and the heights of the characters are printed.

The default method uses the attribute `ComputedPrimeBlockss` for storing the computed value at position p, and calls the operation `PrimeBlocksOp` for computing values that are not yet known.

Two ordinary irreducible characters χ, ψ of a group G are said to lie in the same *p-block* if the images of their central characters ω_χ, ω_ψ (see `CentralCharacter` (72.8.17)) under the natural ring epimorphism $R \to R/M$ are equal, where R denotes the ring of algebraic integers in the complex number field, and M is a maximal ideal in R with $pR \subseteq M$. (The distribution to p-blocks is in fact independent of the choice of M, see [Isa76].)

For $|G| = p^a m$ where p does not divide m, the *defect* of a block is the integer d such that p^{a-d} is the largest power of p that divides the degrees of all characters in the block.

The *height* of a character χ in the block is defined as the largest exponent h for which p^h divides $\chi(1)/p^{a-d}$.

Example

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> pbl:= PrimeBlocks( tbl, 2 );
rec( block := [ 1, 1, 1, 1, 1, 2 ],
  centralcharacter := [ [ ,, 56,, 24 ], [ ,, -7,, 3 ] ],
  defect := [ 3, 0 ], height := [ 0, 0, 0, 1, 0, 0 ],
  relevant := [ 3, 5 ] )
```

71.11.2 SameBlock

▷ SameBlock(*p, omega1, omega2, relevant*) (function)

Let *p* be a prime integer, *omega1* and *omega2* be two central characters (or their values lists) of a character table, and *relevant* be a list of positions as is stored in the component relevant of a record returned by PrimeBlocks (71.11.1).

SameBlock returns true if *omega1* and *omega2* are equal modulo any maximal ideal in the ring of complex algebraic integers containing the ideal spanned by *p*, and false otherwise.

Example

```
gap> omega:= List( Irr( tbl ), CentralCharacter );;
gap> SameBlock( 2, omega[1], omega[2], pbl.relevant );
true
gap> SameBlock( 2, omega[1], omega[6], pbl.relevant );
false
```

71.11.3 BlocksInfo

▷ BlocksInfo(*modtbl*) (attribute)

For a Brauer character table *modtbl*, the value of BlocksInfo is a list of (mutable) records, the *i*-th entry containing information about the *i*-th block. Each record has the following components.

defect
: the defect of the block,

ordchars
: the list of positions of the ordinary characters that belong to the block, relative to Irr(OrdinaryCharacterTable(*modtbl*)),

modchars
: the list of positions of the Brauer characters that belong to the block, relative to IBr(*modtbl*).

Optional components are

basicset
: a list of positions of ordinary characters in the block whose restriction to *modtbl* is maximally linearly independent, relative to Irr(OrdinaryCharacterTable(*modtbl*)),

decmat
: the decomposition matrix of the block, it is stored automatically when DecompositionMatrix (71.11.4) is called for the block,

decinv
: inverse of the decomposition matrix of the block, restricted to the ordinary characters described by basicset,

brauertree
: a list that describes the Brauer tree of the block, in the case that the block is of defect 1.

Example

```
gap> BlocksInfo( CharacterTable( "L3(2)" ) mod 2 );
[ rec( basicset := [ 1, 2, 3 ],
      decinv := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
      defect := 3, modchars := [ 1, 2, 3 ],
      ordchars := [ 1, 2, 3, 4, 5 ] ),
  rec( basicset := [ 6 ], decinv := [ [ 1 ] ], defect := 0,
      modchars := [ 4 ], ordchars := [ 6 ] ) ]
```

71.11.4 DecompositionMatrix

▷ DecompositionMatrix(*modtbl[, blocknr]*) (operation)

Let *modtbl* be a Brauer character table.

Called with one argument, DecompositionMatrix returns the decomposition matrix of *modtbl*, where the rows and columns are indexed by the irreducible characters of the ordinary character table of *modtbl* and the irreducible characters of *modtbl*, respectively,

Called with two arguments, DecompositionMatrix returns the decomposition matrix of the block of *modtbl* with number *blocknr*; the matrix is stored as value of the decmat component of the *blocknr*-th entry of the BlocksInfo (71.11.3) list of *modtbl*.

An ordinary irreducible character is in block i if and only if all characters before the first character of the same block lie in $i-1$ different blocks. An irreducible Brauer character is in block i if it has nonzero scalar product with an ordinary irreducible character in block i.

DecompositionMatrix is based on the more general function Decomposition (25.4.1).

Example

```
gap> modtbl:= CharacterTable( "L3(2)" ) mod 2;
BrauerTable( "L3(2)", 2 )
gap> DecompositionMatrix( modtbl );
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 1, 0 ],
  [ 1, 1, 1, 0 ], [ 0, 0, 0, 1 ] ]
gap> DecompositionMatrix( modtbl, 1 );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ]
gap> DecompositionMatrix( modtbl, 2 );
[ [ 1 ] ]
```

71.11.5 LaTeXStringDecompositionMatrix

▷ LaTeXStringDecompositionMatrix(*modtbl[, blocknr][, options]*) (function)

is a string that contains LaTeX code to print a decomposition matrix (see DecompositionMatrix (71.11.4)) nicely.

The optional argument *options*, if present, must be a record with components phi, chi (strings used in each label for columns and rows), collabels, rowlabels (subscripts for the labels). The defaults for phi and chi are "{\\tt Y}" and "{\\tt X}", the defaults for collabels and rowlabels are the lists of positions of the Brauer characters and ordinary characters in the respective lists of irreducibles in the character tables.

The optional components `nrows` and `ncols` denote the maximal number of rows and columns per array; if they are present then each portion of `nrows` rows and `ncols` columns forms an array of its own which is enclosed in `\[`, `\]`.

If the component `decmat` is bound in *`options`* then it must be the decomposition matrix in question, in this case the matrix is not computed from the information in *`modtbl`*.

For those character tables from the GAP table library that belong to the Atlas of Finite Groups [CCN+85], `AtlasLabelsOfIrreducibles` (**CTblLib: AtlasLabelsOfIrreducibles**) constructs character labels that are compatible with those used in the Atlas (see (**CTblLib: Atlas Tables**) in the manual of the GAP Character Table Library).

Example

```
gap> modtbl:= CharacterTable( "L3(2)" ) mod 2;;
gap> Print( LaTeXStringDecompositionMatrix( modtbl, 1 ) );
\[
\begin{array}{r|rrr} \hline
 & {\tt Y}_{1}
 & {\tt Y}_{2}
 & {\tt Y}_{3}
 \rule[-7pt]{0pt}{20pt} \\ \hline
{\tt X}_{1} & 1 & . & . \rule[0pt]{0pt}{13pt} \\
{\tt X}_{2} & . & 1 & . \\
{\tt X}_{3} & . & . & 1 \\
{\tt X}_{4} & . & 1 & 1 \\
{\tt X}_{5} & 1 & 1 & 1 \rule[-7pt]{0pt}{5pt} \\
\hline
\end{array}
\]
gap> options:= rec( phi:= "\\varphi", chi:= "\\chi" );;
gap> Print( LaTeXStringDecompositionMatrix( modtbl, 1, options ) );
\[
\begin{array}{r|rrr} \hline
 & \varphi_{1}
 & \varphi_{2}
 & \varphi_{3}
 \rule[-7pt]{0pt}{20pt} \\ \hline
\chi_{1} & 1 & . & . \rule[0pt]{0pt}{13pt} \\
\chi_{2} & . & 1 & . \\
\chi_{3} & . & . & 1 \\
\chi_{4} & . & 1 & 1 \\
\chi_{5} & 1 & 1 & 1 \rule[-7pt]{0pt}{5pt} \\
\hline
\end{array}
\]
```

71.12 Other Operations for Character Tables

In the following, we list operations for character tables that are not attributes.

71.12.1 Index (for two character tables)

▷ `Index(tbl, subtbl)` (operation)

For two character tables *tbl* and *subtbl*, `Index` returns the quotient of the `Size` (71.8.5) values of *tbl* and *subtbl*. The containment of the underlying groups of *subtbl* and *tbl* is *not* checked; so the distinction between `Index` (39.3.2) and `IndexNC` (39.3.2) is not made for character tables.

71.12.2 IsInternallyConsistent (for character tables)

▷ `IsInternallyConsistent(tbl)` (method)

For an *ordinary* character table *tbl*, `IsInternallyConsistent` (12.8.3) checks the consistency of the following attribute values (if stored).

- `Size` (30.4.6), `SizesCentralizers` (71.9.2), and `SizesConjugacyClasses` (71.9.3).
- `SizesCentralizers` (71.9.2) and `OrdersClassRepresentatives` (71.9.1).
- `ComputedPowerMaps` (73.1.1) and `OrdersClassRepresentatives` (71.9.1).
- `SizesCentralizers` (71.9.2) and `Irr` (71.8.2).
- `Irr` (71.8.2) (first orthogonality relation).

For a *Brauer* table *tbl*, `IsInternallyConsistent` checks the consistency of the following attribute values (if stored).

- `Size` (30.4.6), `SizesCentralizers` (71.9.2), and `SizesConjugacyClasses` (71.9.3).
- `SizesCentralizers` (71.9.2) and `OrdersClassRepresentatives` (71.9.1).
- `ComputedPowerMaps` (73.1.1) and `OrdersClassRepresentatives` (71.9.1).
- `Irr` (71.8.2) (closure under complex conjugation and Frobenius map).

If no inconsistency occurs, `true` is returned, otherwise each inconsistency is printed to the screen if the level of `InfoWarning` (7.4.7) is at least 1 (see 7.4), and `false` is returned at the end.

71.12.3 IsPSolvableCharacterTable

▷ `IsPSolvableCharacterTable(tbl, p)` (operation)
▷ `IsPSolubleCharacterTable(tbl, p)` (operation)
▷ `IsPSolvableCharacterTableOp(tbl, p)` (operation)
▷ `IsPSolubleCharacterTableOp(tbl, p)` (operation)
▷ `ComputedIsPSolvableCharacterTables(tbl)` (attribute)
▷ `ComputedIsPSolubleCharacterTables(tbl)` (attribute)

`IsPSolvableCharacterTable` for the ordinary character table *tbl* corresponds to `IsPSolvable` (39.15.23) for the group of *tbl*, *p* must be either a prime integer or `0`.

The default method uses the attribute `ComputedIsPSolvableCharacterTables` for storing the computed value at position *p*, and calls the operation `IsPSolvableCharacterTableOp` for computing values that are not yet known.

Example

```
gap> tbl:= CharacterTable( "Sz(8)" );;
gap> IsPSolvableCharacterTable( tbl, 2 );
false
gap> IsPSolvableCharacterTable( tbl, 3 );
true
```

71.12.4 IsClassFusionOfNormalSubgroup

▷ IsClassFusionOfNormalSubgroup(*subtbl, fus, tbl*) (function)

For two ordinary character tables *tbl* and *subtbl* of a group G and its subgroup U, say, and a list *fus* of positive integers that describes the class fusion of U into G, IsClassFusionOfNormalSubgroup returns true if U is a normal subgroup of G, and false otherwise.

Example

```
gap> tblc2:= CharacterTable( "Cyclic", 2 );;
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> fus:= PossibleClassFusions( tblc2, tbld8 );
[ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ]
gap> List(fus, map -> IsClassFusionOfNormalSubgroup(tblc2, map, tbld8));
[ true, false, false ]
```

71.12.5 Indicator

▷ Indicator(*tbl[, characters], n*) (operation)
▷ IndicatorOp(*tbl, characters, n*) (operation)
▷ ComputedIndicators(*tbl*) (attribute)

If *tbl* is an ordinary character table then Indicator returns the list of *n*-th Frobenius-Schur indicators of the characters in the list *characters*; the default of *characters* is Irr(*tbl*).

The n-th Frobenius-Schur indicator $\nu_n(\chi)$ of an ordinary character χ of the group G is given by $\nu_n(\chi) = (\sum_{g \in G} \chi(g^n))/|G|$.

If *tbl* is a Brauer table in characteristic $\neq 2$ and $n = 2$ then Indicator returns the second indicator.

The default method uses the attribute ComputedIndicators for storing the computed value at position *n*, and calls the operation IndicatorOp for computing values that are not yet known.

Example

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> Indicator( tbl, 2 );
[ 1, 0, 0, 1, 1, 1 ]
```

71.12.6 NrPolyhedralSubgroups

▷ NrPolyhedralSubgroups(*tbl, c1, c2, c3*) (function)

returns the number and isomorphism type of polyhedral subgroups of the group with ordinary character table `tbl` which are generated by an element g of class `c1` and an element h of class `c2` with the property that the product gh lies in class `c3`.

According to [NPP84, p. 233], the number of polyhedral subgroups of isomorphism type V_4, D_{2n}, A_4, S_4, and A_5 can be derived from the class multiplication coefficient (see `ClassMultiplicationCoefficient` (71.12.7)) and the number of Galois conjugates of a class (see `ClassOrbit` (71.9.12)).

The classes `c1`, `c2` and `c3` in the parameter list must be ordered according to the order of the elements in these classes. If elements in class `c1` and `c2` do not generate a polyhedral group then `fail` is returned.

Example

```
gap> NrPolyhedralSubgroups( tbl, 2, 2, 4 );
rec( number := 21, type := "D8" )
```

71.12.7 ClassMultiplicationCoefficient (for character tables)

▷ `ClassMultiplicationCoefficient(tbl, i, j, k)` (operation)

returns the class multiplication coefficient of the classes `i`, `j`, and `k` of the group G with ordinary character table `tbl`.

The class multiplication coefficient $c_{i,j,k}$ of the classes `i`, `j`, `k` equals the number of pairs (x,y) of elements $x,y \in G$ such that x lies in class `i`, y lies in class `j`, and their product xy is a fixed element of class `k`.

In the center of the group algebra of G, these numbers are found as coefficients of the decomposition of the product of two class sums K_i and K_j into class sums:

$$K_i K_j = \sum_k c_{ijk} K_k.$$

Given the character table of a finite group G, whose classes are $C_1, \ldots, C_r$ with representatives $g_i \in C_i$, the class multiplication coefficient c_{ijk} can be computed with the following formula:

$$c_{ijk} = |C_i| \cdot |C_j| / |G| \cdot \sum_{\chi \in Irr(G)} \chi(g_i)\chi(g_j)\chi(g_k^{-1})/\chi(1).$$

On the other hand the knowledge of the class multiplication coefficients admits the computation of the irreducible characters of G, see `IrrDixonSchneider` (71.14.1).

71.12.8 ClassStructureCharTable

▷ `ClassStructureCharTable(tbl, classes)` (function)

returns the so-called class structure of the classes in the list `classes`, for the character table `tbl` of the group G. The length of `classes` must be at least 2.

Let $C = (C_1, C_2, \ldots, C_n)$ denote the n-tuple of conjugacy classes of G that are indexed by `classes`. The class structure $n(C)$ equals the number of n-tuples $(g_1, g_2, \ldots, g_n)$ of elements $g_i \in C_i$ with $g_1 g_2 \cdots g_n = 1$. Note the difference to the definition of the class multiplication coefficients in `ClassMultiplicationCoefficient` (71.12.7).

$n(C_1,C_2,\ldots,C_n)$ is computed using the formula

$$n(C_1,C_2,\ldots,C_n) = |C_1||C_2|\cdots|C_n|/|G| \cdot \sum_{\chi \in Irr(G)} \chi(g_1)\chi(g_2)\cdots\chi(g_n)/\chi(1)^{n-2}.$$

71.12.9 MatClassMultCoeffsCharTable

▷ `MatClassMultCoeffsCharTable(`*`tbl, i`*`)` (function)

For an ordinary character table *tbl* and a class position *i*, `MatClassMultCoeffsCharTable` returns the matrix $[a_{ijk}]_{j,k}$ of structure constants (see `ClassMultiplicationCoefficient` (71.12.7)).

Example

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> ClassMultiplicationCoefficient( tbl, 2, 2, 4 );
4
gap> ClassStructureCharTable( tbl, [ 2, 2, 4 ] );
168
gap> ClassStructureCharTable( tbl, [ 2, 2, 2, 4 ] );
1848
gap> MatClassMultCoeffsCharTable( tbl, 2 );
[ [ 0, 1, 0, 0, 0, 0 ], [ 21, 4, 3, 4, 0, 0 ], [ 0, 8, 6, 8, 7, 7 ],
  [ 0, 8, 6, 1, 7, 7 ], [ 0, 0, 3, 4, 0, 7 ], [ 0, 0, 3, 4, 7, 0 ] ]
```

71.13 Printing Character Tables

71.13.1 ViewObj (for a character table)

▷ `ViewObj(`*`tbl`*`)` (method)

The default `ViewObj` (6.3.5) method for ordinary character tables prints the string `"CharacterTable"`, followed by the identifier (see `Identifier` (71.9.8)) or, if known, the group of the character table enclosed in brackets. `ViewObj` (6.3.5) for Brauer tables does the same, except that the first string is replaced by `"BrauerTable"`, and that the characteristic is also shown.

71.13.2 PrintObj (for a character table)

▷ `PrintObj(`*`tbl`*`)` (method)

The default `PrintObj` (6.3.5) method for character tables does the same as `ViewObj` (6.3.5), except that `PrintObj` (6.3.5) is used for the group instead of `ViewObj` (6.3.5).

71.13.3 Display (for a character table)

▷ `Display(`*`tbl`*`)` (method)

There are various ways to customize the `Display` (6.3.6) output for character tables. First we describe the default behaviour, alternatives are then described below.

The default Display (6.3.6) method prepares the data in *tbl* for a columnwise output. The number of columns printed at one time depends on the actual line length, which can be accessed and changed by the function SizeScreen (6.12.1).

An interesting variant of Display (6.3.6) is the function PageDisplay (**GAPDoc: PageDisplay**). Convenient ways to print the Display (6.3.6) format to a file are given by the function PrintTo1 (**GAPDoc: PrintTo1**) or by using PageDisplay (**GAPDoc: PageDisplay**) and the facilities of the pager used, cf. Pager (2.4.1).

An interactive variant of Display (6.3.6) is the Browse (**Browse: Browse**) method for character tables that is provided by the GAP package Browse, see Browse (**Browse: Browse (for character tables)**).

Display (6.3.6) shows certain characters (by default all irreducible characters) of *tbl*, together with the orders of the centralizers in factorized form and the available power maps (see ComputedPowerMaps (73.1.1)). The *n*-th displayed character is given the name X.*n*.

The first lines of the output describe the order of the centralizer of an element of the class factorized into its prime divisors.

The next line gives the name of each class. If no class names are stored on *tbl*, ClassNames (71.9.6) is called.

Preceded by a name P*n*, the next lines show the *n*th power maps of *tbl* in terms of the former shown class names.

Every ambiguous or unknown (see Chapter 74) value of the table is displayed as a question mark ?.

Irrational character values are not printed explicitly because the lengths of their printed representation might disturb the layout. Instead of that every irrational value is indicated by a name, which is a string of at least one capital letter.

Once a name for an irrational value is found, it is used all over the printed table. Moreover the complex conjugate (see ComplexConjugate (18.5.2), GaloisCyc (18.5.1)) and the star of an irrationality (see StarCyc (18.5.3)) are represented by that very name preceded by a / and a *, respectively.

The printed character table is then followed by a legend, a list identifying the occurring symbols with their actual values. Occasionally this identification is supplemented by a quadratic representation of the irrationality (see Quadratic (18.5.4)) together with the corresponding Atlas notation (see [CCN+85]).

This default style can be changed by prescribing a record *arec* of options, which can be given

1. as an optional argument in the call to Display (6.3.6),
2. as the value of the attribute DisplayOptions (71.13.4) if this value is stored in the table,
3. as the value of the global variable CharacterTableDisplayDefaults.User, or
4. as the value of the global variable CharacterTableDisplayDefaults.Global

(in this order of precedence).

The following components of *arec* are supported.

centralizers
: false to suppress the printing of the orders of the centralizers, or the string "ATLAS" to force the printing of non-factorized centralizer orders in a style similar to that used in the Atlas of Finite Groups [CCN+85],

`chars`
: an integer or a list of integers to select a sublist of the irreducible characters of *tbl*, or a list of characters of *tbl* (in this case the letter `"X"` is replaced by `"Y"`),

`classes`
: an integer or a list of integers to select a sublist of the classes of *tbl*,

`indicator`
: `true` enables the printing of the second Frobenius Schur indicator, a list of integers enables the printing of the corresponding indicators (see `Indicator` (71.12.5)),

`letter`
: a single capital letter (e. g. `"P"` for permutation characters) to replace the default `"X"` in character names,

`powermap`
: an integer or a list of integers to select a subset of the available power maps, `false` to suppress the printing of power maps, or the string `"ATLAS"` to force a printing of class names and power maps in a style similar to that used in the Atlas of Finite Groups [CCN+85],

`Display`
: the function that is actually called in order to display the table; the arguments are the table and the optional record, whose components can be used inside the `Display` function,

`StringEntry`
: a function that takes either a character value or a character value and the return value of `StringEntryData` (see below), and returns the string that is actually displayed; it is called for all character values to be displayed, and also for the displayed indicator values (see above),

`StringEntryData`
: a unary function that is called once with argument *tbl* before the character values are displayed; it returns an object that is used as second argument of the function `StringEntry`,

`Legend`
: a function that takes the result of the `StringEntryData` call as its only argument, after the character table has been displayed; the return value is a string that describes the symbols used in the displayed table in a formatted way, it is printed below the displayed table.

71.13.4 DisplayOptions

▷ `DisplayOptions(`*tbl*`)` (attribute)

There is no default method to compute a value, one can set a value with `SetDisplayOptions`.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> Display( tbl );
A5

     2  2  2  .  .  .
     3  1  .  1  .  .
     5  1  .  .  1  1
```

```
       1a 2a 3a 5a 5b
    2P 1a 1a 3a 5b 5a
    3P 1a 2a 1a 5b 5a
    5P 1a 2a 3a 1a 1a

X.1     1  1  1  1  1
X.2     3 -1  .  A *A
X.3     3 -1  . *A  A
X.4     4  .  1 -1 -1
X.5     5  1 -1  .  .

A = -E(5)-E(5)^4
  = (1-Sqrt(5))/2 = -b5
gap> CharacterTableDisplayDefaults.User:= rec(
>        powermap:= "ATLAS", centralizers:= "ATLAS", chars:= false );;
gap> Display( CharacterTable( "A5" ) );
A5

    60  4  3  5  5

 p      A  A  A  A
 p'     A  A  A  A
    1A 2A 3A 5A B*

gap> options:= rec( chars:= 4, classes:= [ tbl.3a .. tbl.5a ],
>                   centralizers:= false, indicator:= true,
>                   powermap:= [ 2 ] );;
gap> Display( tbl, options );
A5

          3a 5a
       2P 3a 5b
       2
X.4    +   1 -1
gap> SetDisplayOptions( tbl, options );  Display( tbl );
A5

          3a 5a
       2P 3a 5b
       2
X.4    +   1 -1
gap> Unbind( CharacterTableDisplayDefaults.User );
```

71.13.5 PrintCharacterTable

▷ `PrintCharacterTable(`*tbl, varname*`)` (function)

Let *tbl* be a nearly character table, and *varname* a string. `PrintCharacterTable` prints those values of the supported attributes (see `SupportedCharacterTableInfo` (71.3.4)) that are known for *tbl*.

The output of PrintCharacterTable is GAP readable; actually reading it into GAP will bind the variable with name *varname* to a character table that coincides with *tbl* for all printed components.

This is used mainly for saving character tables to files. A more human readable form is produced by Display (6.3.6).

Example

```
gap> PrintCharacterTable( CharacterTable( "Cyclic", 2 ), "tbl" );
tbl:= function()
local tbl, i;
tbl:=rec();
tbl.Irr:=
[ [ 1, 1 ], [ 1, -1 ] ];
tbl.NrConjugacyClasses:=
2;
tbl.Size:=
2;
tbl.OrdersClassRepresentatives:=
[ 1, 2 ];
tbl.SizesCentralizers:=
[ 2, 2 ];
tbl.UnderlyingCharacteristic:=
0;
tbl.ClassParameters:=
[ [ 1, 0 ], [ 1, 1 ] ];
tbl.CharacterParameters:=
[ [ 1, 0 ], [ 1, 1 ] ];
tbl.Identifier:=
"C2";
tbl.InfoText:=
"computed using generic character table for cyclic groups";
tbl.ComputedPowerMaps:=
[ , [ 1, 1 ] ];
ConvertToLibraryCharacterTableNC(tbl);
return tbl;
end;
tbl:= tbl();
```

71.14 Computing the Irreducible Characters of a Group

Several algorithms are available for computing the irreducible characters of a finite group *G*. The default method for arbitrary finite groups is to use the Dixon-Schneider algorithm (see IrrDixonSchneider (71.14.1)). For supersolvable groups, Conlon's algorithm can be used (see IrrConlon (71.14.2)). For abelian-by-supersolvable groups, the Baum-Clausen algorithm for computing the irreducible representations (see IrreducibleRepresentations (71.14.4)) can be used to compute the irreducible characters (see IrrBaumClausen (71.14.3)).

These functions are installed in methods for Irr (71.8.2), but explicitly calling one of them will *not* set the Irr (71.8.2) value of *G*.

71.14.1 IrrDixonSchneider

▷ `IrrDixonSchneider(G)` (attribute)

computes the irreducible characters of the finite group `G`, using the Dixon-Schneider method (see 71.16). It calls `DixonInit` (71.17.2) and `DixonSplit` (71.17.4), and finally returns the list returned by `DixontinI` (71.17.3). See also the sections 71.18 and 71.19.

71.14.2 IrrConlon

▷ `IrrConlon(G)` (attribute)

For a finite solvable group `G`, `IrrConlon` returns a list of certain irreducible characters of `G`, among those all irreducibles that have the supersolvable residuum of `G` in their kernels; so if `G` is supersolvable, all irreducible characters of `G` are returned. An error is signalled if `G` is not solvable.

The characters are computed using Conlon's algorithm (see [Con90a] and [Con90b]). For each irreducible character in the returned list, the monomiality information (see `TestMonomial` (75.4.1)) is stored.

71.14.3 IrrBaumClausen

▷ `IrrBaumClausen(G)` (attribute)

`IrrBaumClausen` returns the absolutely irreducible ordinary characters of the factor group of the finite solvable group `G` by the derived subgroup of its supersolvable residuum.

The characters are computed using the algorithm by Baum and Clausen (see [BC94]). An error is signalled if `G` is not solvable.

Example

```
gap> g:= SL(2,3);;
gap> irr1:= IrrDixonSchneider( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 1, E(3)^2, E(3), 1, E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 1, E(3), E(3)^2, 1, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, 1, 1, -2, -1, -1, 0 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 2, E(3)^2, E(3), -2, -E(3), -E(3)^2, 0 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 2, E(3), E(3)^2, -2, -E(3)^2, -E(3), 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 0, 0, 3, 0, 0, -1 ] ) ]
gap> irr2:= IrrConlon( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 1, E(3), E(3)^2, 1, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 1, E(3)^2, E(3), 1, E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 0, 0, 3, 0, 0, -1 ] ) ]
gap> irr3:= IrrBaumClausen( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
```

```
  Character( CharacterTable( SL(2,3) ),
    [ 1, E(3), E(3)^2, 1, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ),
    [ 1, E(3)^2, E(3), 1, E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 0, 0, 3, 0, 0, -1 ] ) ]
gap> chi:= irr2[4];;  HasTestMonomial( chi );
true
```

71.14.4 IrreducibleRepresentations

▷ `IrreducibleRepresentations(`*G[, F]*`)` (attribute)

Called with a finite group *G* and a field *F*, `IrreducibleRepresentations` returns a list of representatives of the irreducible matrix representations of *G* over *F*, up to equivalence.

If *G* is the only argument then `IrreducibleRepresentations` returns a list of representatives of the absolutely irreducible complex representations of *G*, up to equivalence.

At the moment, methods are available for the following cases: If *G* is abelian by supersolvable the method of [BC94] is used.

Otherwise, if *F* and *G* are both finite, the regular module of *G* is split by MeatAxe methods which can make this an expensive operation.

Finally, if *F* is not given (i.e. it defaults to the cyclotomic numbers) and *G* is a finite group, the method of [Dix93] (see `IrreducibleRepresentationsDixon` (71.14.5)) is used.

For other cases no methods are implemented yet.

The representations obtained are *not* guaranteed to be "nice" (for example preserving a unitary form) in any way.

See also `IrreducibleModules` (71.15.1), which provides efficient methods for solvable groups.

Example

```
gap> g:= AlternatingGroup( 4 );;
gap> repr:= IrreducibleRepresentations( g );
[ Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ 1 ] ], [ [ 1 ] ], [ [ 1 ] ] ],
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ E(3) ] ], [ [ 1 ] ], [ [ 1 ] ] ],
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ E(3)^2 ] ], [ [ 1 ] ], [ [ 1 ] ] ],
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ 0, 0, 1 ], [ 1, 0, 0 ], [ 0, 1, 0 ] ],
      [ [ -1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, -1 ] ],
      [ [ 1, 0, 0 ], [ 0, -1, 0 ], [ 0, 0, -1 ] ] ] ]
gap> ForAll( repr, IsGroupHomomorphism );
true
gap> Length( repr );
4
gap> gens:= GeneratorsOfGroup( g );
[ (1,2,3), (2,3,4) ]
gap> List( gens, x -> x^repr[1] );
[ [ [ 1 ] ], [ [ 1 ] ] ]
gap>  List( gens, x -> x^repr[4] );
[ [ [ 0, 0, -1 ], [ 1, 0, 0 ], [ 0, -1, 0 ] ],
  [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ] ]
```

71.14.5 IrreducibleRepresentationsDixon

▷ IrreducibleRepresentationsDixon(*G[, chi]*) (function)

Called with one argument, a group *G*, IrreducibleRepresentationsDixon computes (representatives of) all irreducible complex representations for the finite group *G*, using the method of [Dix93], which computes the character table and computes the representation as constituent of an induced monomial representation of a subgroup.

This method can be quite expensive for larger groups, for example it might involve calculation of the subgroup lattice of *G*.

A character *chi* of *G* can be given as the second argument, in this case only a representation affording *chi* is returned.

The second argument can also be a list of characters of *G*, in this case only representations for characters in this list are computed.

Note that this method might fail if for an irreducible representation there is no subgroup in which its reduction has a linear constituent with multiplicity one.

Example

```
gap> a5:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
gap> char:= First( Irr( a5 ), x -> x[1] = 4 );
Character( CharacterTable( Alt( [ 1 .. 5 ] ) ), [ 4, 0, 1, -1, -1 ] )
gap> hom:=IrreducibleRepresentationsDixon( a5, char );;
gap> Order( a5.1*a5.2 ) = Order( Image( hom, a5.1 )*Image( hom, a5.2 ) );
true
gap> reps:= List( ConjugacyClasses( a5 ), Representative );;
gap> List( reps, g -> TraceMat( Image( hom, g ) ) );
[ 4, 0, 1, -1, -1 ]
```

71.15 Representations Given by Modules

This section describes functions that return certain modules of a given group. (Extensions by modules can be formed by the command Extensions (46.8.4).)

71.15.1 IrreducibleModules

▷ IrreducibleModules(*G, F, dim*) (operation)

returns a list of length 2. The first entry is a generating system of *G*. The second entry is a list of all irreducible modules of *G* over the field *F* in dimension *dim*, given as MeatAxe modules (see GModuleByMats (69.1.1)).

71.15.2 AbsolutelyIrreducibleModules

▷ AbsolutelyIrreducibleModules(*G, F, dim*) (operation)

▷ AbsoluteIrreducibleModules(*G, F, dim*) (operation)

▷ AbsolutIrreducibleModules(*G, F, dim*) (operation)

returns a list of length 2. The first entry is a generating system of `G`. The second entry is a list of all absolute irreducible modules of `G` over the field `F` in dimension `dim`, given as MeatAxe modules (see `GModuleByMats` (69.1.1)). The other two names are just synonyms.

71.15.3 RegularModule

▷ `RegularModule(G, F)` (operation)

returns a list of length 2. The first entry is a generating system of `G`. The second entry is the regular module of `G` over `F`, given as a MeatAxe module (see `GModuleByMats` (69.1.1)).

71.16 The Dixon-Schneider Algorithm

The GAP library implementation of the Dixon-Schneider algorithm first computes the linear characters, using the commutator factor group. If irreducible characters are missing afterwards, they are computed using the techniques described in [Dix67], [Sch90] and [Hul93].

Called with a group G, the function `CharacterTable` (71.3.1) returns a character table object that stores already information such as class lengths, but not the irreducible characters. The routines that compute the irreducibles may use the information that is already contained in this table object. In particular the ordering of classes in the computed characters coincides with the ordering of classes in the character table of `G` (see 71.6). Thus it is possible to combine computations using the group with character theoretic computations (see 71.17 for details), for example one can enter known characters. Note that the user is responsible for the correctness of the characters. (There is little use in providing the trivial character to the routine.)

The computation of irreducible characters from the group needs to identify the classes of group elements very often, so it can be helpful to store a class list of all group elements. Since this is obviously limited by the group order, it is controlled by the global function `IsDxLargeGroup` (71.17.8).

The routines compute in a prime field of size p, such that the exponent of the group divides $(p-1)$ and such that $2\sqrt{|G|} < p$. Currently prime fields of size smaller than 65 536 are handled more efficiently than larger prime fields, so the runtime of the character calculation depends on how large the chosen prime is.

The routine stores a Dixon record (see `DixonRecord` (71.17.1)) in the group that helps routines that identify classes, for example `FusionConjugacyClasses` (73.3.1), to work much faster. Note that interrupting Dixon-Schneider calculations will prevent GAP from cleaning up the Dixon record; when the computation by `IrrDixonSchneider` (71.14.1) is complete, the possibly large record is shrunk to an acceptable size.

71.17 Advanced Methods for Dixon-Schneider Calculations

The computation of irreducible characters of very large groups may take quite some time. On the other hand, for the expert only a few irreducible characters may be needed, since the other ones can be computed using character theoretic methods such as tensoring, induction, and restriction. Thus GAP provides also step-by-step routines for doing the calculations. These routines allow one to compute some characters and to stop before all are calculated. Note that there is no “safety net”: The routines (being somehow internal) do no error checking, and assume the information given is correct.

When the info level of InfoCharacterTable (71.4.2) if positive, information about the progress of splitting is printed. (The default value is zero.)

71.17.1 DixonRecord

▷ DixonRecord(*G*) (attribute)

The DixonRecord of a group contains information used by the routines to compute the irreducible characters and related information via the Dixon-Schneider algorithm such as class arrangement and character spaces split obtained so far. Usually this record is passed as argument to all subfunctions to avoid a long argument list. It has a component conjugacyClasses which contains the classes of *G* *ordered as the algorithm needs them*.

71.17.2 DixonInit

▷ DixonInit(*G*) (function)

This function does all the initializations for the Dixon-Schneider algorithm. This includes calculation of conjugacy classes, power maps, linear characters and character morphisms. It returns a record (see DixonRecord (71.17.1) and Section 71.18) that can be used when calculating the irreducible characters of *G* interactively.

71.17.3 DixontinI

▷ DixontinI(*D*) (function)

This function ends a Dixon-Schneider calculation. It sorts the characters according to the degree and unbinds components in the Dixon record that are not of use any longer. It returns a list of irreducible characters.

71.17.4 DixonSplit

▷ DixonSplit(*D*) (function)

This function performs one splitting step in the Dixon-Schneider algorithm. It selects a class, computes the (partial) class sum matrix, uses it to split character spaces and stores all the irreducible characters obtained that way.

The class to use for splitting is chosen via BestSplittingMatrix (71.17.5) and the options described for this function apply here.

DixonSplit returns the number of the class that was used for splitting if a split was performed, and fail otherwise.

71.17.5 BestSplittingMatrix

▷ BestSplittingMatrix(*D*) (function)

returns the number of the class sum matrix that is assumed to yield the best (cost/earning ration) split. This matrix then will be the next one computed and used.

The global option maxclasslen (defaulting to infinity (18.2.1)) is recognized by BestSplittingMatrix: Only classes whose length is limited by the value of this option will be considered for splitting. If no usable class remains, fail is returned.

71.17.6 DxIncludeIrreducibles

▷ DxIncludeIrreducibles(*D, new[, newmod]*) (function)

This function takes a list of irreducible characters *new*, each given as a list of values (corresponding to the class arrangement in *D*), and adds these to a partial computed list of irreducibles as maintained by the Dixon record *D*. This permits one to add characters in interactive use obtained from other sources and to continue the Dixon-Schneider calculation afterwards. If the optional argument *newmod* is given, it must be a list of reduced characters, corresponding to *new*. (Otherwise the function has to reduce the characters itself.)

The function closes the new characters under the action of Galois automorphisms and tensor products with linear characters.

71.17.7 SplitCharacters

▷ SplitCharacters(*D, list*) (function)

This routine decomposes the characters given in *list* according to the character spaces found up to this point. By applying this routine to tensor products etc., it may result in characters with smaller norm, even irreducible ones. Since the recalculation of characters is only possible if the degree is small enough, the splitting process is applied only to characters of sufficiently small degree.

71.17.8 IsDxLargeGroup

▷ IsDxLargeGroup(*G*) (function)

returns true if the order of the group *G* is smaller than the current value of the global variable DXLARGEGROUPORDER, and false otherwise. In Dixon-Schneider calculations, for small groups in the above sense a class map is stored, whereas for large groups, each occurring element is identified individually.

71.18 Components of a Dixon Record

The "Dixon record" *D* returned by DixonInit (71.17.2) stores all the information that is used by the Dixon-Schneider routines while computing the irreducible characters of a group. Some entries, however, may be useful to know about when using the algorithm interactively, see 71.19.

group
: the group *G* of which the character table is to be computed,

conjugacyClasses
: classes of *G* (all characters stored in the Dixon record correspond to this arrangement of classes),

`irreducibles`
the already known irreducible characters (given as lists of their values on the conjugacy classes),

`characterTable`
the `CharacterTable` (71.3.1) value of G (whose irreducible characters are not yet known),

`ClassElement( D, el )`
a function that returns the number of the class of G that contains the element *el*.

71.19 An Example of Advanced Dixon-Schneider Calculations

First, we set the appropriate info level higher.

Example

```
gap> SetInfoLevel( InfoCharacterTable, 1 );
```

for printout of some internal results. We now define our group, which is isomorphic to $\mathrm{PSL}_4(3)$.

Example

```
gap> g:= PrimitiveGroup(40,5);
PSL(4, 3)
gap> Size(g);
6065280
gap> d:= DixonInit( g );;
#I  29 classes
#I  choosing prime 65521
gap> c:= d.characterTable;;
```

After the initialisation, one structure matrix is evaluated, yielding smaller spaces and several irreducible characters.

Example

```
gap> DixonSplit( d );
#I  Matrix 2,Representative of Order 3,Centralizer: 5832
#I  Dimensions: [ 1, 2, 1, 4, 12, 1, 1, 2, 1, 2, 1 ]
#I  Two-dim space split
#I  Two-dim space split
#I  Two-dim space split
2
```

In this case spaces of the listed dimensions are a result of the splitting process. The three two dimensional spaces are split successfully by combinatoric means.

We obtain several irreducible characters by tensor products and notify them to the Dixon record.

Example

```
gap> asp:= AntiSymmetricParts( c, d.irreducibles, 2 );;
gap> ro:= ReducedCharacters( c, d.irreducibles, asp );;
gap> Length( ro.irreducibles );
3
gap> DxIncludeIrreducibles( d, ro.irreducibles );
```

The tensor products of the nonlinear characters among each other are reduced with the irreducible characters. The result is split according to the spaces found, which yields characters of smaller norms, but no new irreducibles.

Example

```
gap> nlc:= Filtered( d.irreducibles, i -> i[1] > 1 );;
gap> t:= Tensored( nlc, nlc );;
gap> ro:= ReducedCharacters( c, d.irreducibles, t );;  ro.irreducibles;
[  ]
gap> List( ro.remainders, i -> ScalarProduct( c, i, i) );
[ 2, 2, 4, 4, 4, 4, 13, 13, 18, 18, 19, 21, 21, 36, 36, 29, 34, 34,
  42, 34, 48, 54, 62, 68, 68, 78, 84, 84, 88, 90, 159, 169, 169, 172,
  172, 266, 271, 271, 268, 274, 274, 280, 328, 373, 373, 456, 532,
  576, 679, 683, 683, 754, 768, 768, 890, 912, 962, 1453, 1453, 1601,
  1601, 1728, 1739, 1739, 1802, 2058, 2379, 2414, 2543, 2744, 2744,
  2920, 3078, 3078, 4275, 4275, 4494, 4760, 5112, 5115, 5115, 5414,
  6080, 6318, 7100, 7369, 7369, 7798, 8644, 10392, 12373, 12922,
  14122, 14122, 18948, 21886, 24641, 24641, 25056, 38942, 44950,
  78778 ]
gap> t:= SplitCharacters( d, ro.remainders );;
gap> List( t, i -> ScalarProduct( c, i, i ) );
[ 2, 2, 4, 2, 2, 4, 4, 3, 6, 5, 5, 9, 9, 4, 12, 13, 18, 18, 20, 18,
  20, 24, 26, 32, 32, 16, 42, 59, 69, 69, 72, 72, 36, 72, 78, 78, 84,
  122, 117, 127, 117, 127, 64, 132, 100, 144, 196, 256, 456, 532,
  576, 679, 683, 683, 754, 768, 768, 890, 912, 962, 1453, 1453, 1601,
  1601, 1728, 1739, 1739, 1802, 2058, 2379, 2414, 2543, 2744, 2744,
  2920, 3078, 3078, 4275, 4275, 4494, 4760, 5112, 5115, 5115, 5414,
  6080, 6318, 7100, 7369, 7369, 7798, 8644, 10392, 12373, 12922,
  14122, 14122, 18948, 21886, 24641, 24641, 25056, 38942, 44950,
  78778 ]
```

Finally we calculate the characters induced from all cyclic subgroups and obtain the missing irreducibles by applying the LLL-algorithm to them.

Example

```
gap> ic:= InducedCyclic( c, "all" );;
gap> ro:= ReducedCharacters( c, d.irreducibles, ic );;
gap> Length( ro.irreducibles );
0
gap> l:= LLL( c, ro.remainders );;
gap> Length( l.irreducibles );
13
```

The LLL returns class function objects (see Chapter 72), and the Dixon record works with character values lists. So we convert them to a list of values before feeding them in the machinery of the Dixon-algorithm.

Example

```
gap> l.irreducibles[1];
Character( CharacterTable( PSL(4, 3) ),
 [ 640, -8, -8, -8, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, E(13)^7+E(13)^8+E(13)^11, E(13)^4+E(13)^10+E(13)^12,
  E(13)^2+E(13)^5+E(13)^6, E(13)+E(13)^3+E(13)^9, 0 ] )
gap> l:=List(l.irreducibles,ValuesOfClassFunction);;
gap> DxIncludeIrreducibles( d, l );
gap> Length( d.irreducibles );
29
```

```
gap> Length( d.classes );
29
```

It turns out we have found all irreducible characters. As the last step, we obtain the irreducible characters and tell them to the group. This makes them available also to the character table.

Example

```
gap> irrs:= DixontinI( d );;
#I  Total:1 matrices,[ 2 ]
gap> SetIrr(g,irrs);
gap> Length(Irr(c));
29
gap> SetInfoLevel( InfoCharacterTable, 0 );
```

71.20 Constructing Character Tables from Others

The following operations take one or more character table arguments, and return a character table. This holds also for `BrauerTable` (71.3.2). note that the return value of `BrauerTable` (71.3.2) will in general not know the irreducible Brauer characters, and GAP might be unable to compute these characters.

Note that whenever fusions between input and output tables occur in these operations, they are stored on the concerned tables, and the `NamesOfFusionSources` (73.3.5) values are updated.

(The interactive construction of character tables using character theoretic methods and incomplete tables is not described here.) *Currently it is not supported and will be described in a chapter of its own when it becomes available*.

71.20.1 CharacterTableDirectProduct

▷ `CharacterTableDirectProduct(`*`tbl1, tbl2`*`)` (operation)

is the table of the direct product of the character tables *tbl1* and *tbl2*.

The matrix of irreducibles of this table is the Kronecker product (see `KroneckerProduct` (24.5.8)) of the irreducibles of *tbl1* and *tbl2*.

Products of ordinary and Brauer character tables are supported.

In general, the result will not know an underlying group, so missing power maps (for prime divisors of the result) and irreducibles of the input tables may be computed in order to construct the table of the direct product.

The embeddings of the input tables into the direct product are stored, they can be fetched with `GetFusionMap` (73.3.3); if *tbl1* is equal to *tbl2* then the two embeddings are distinguished by their `specification` components `"1"` and `"2"`, respectively.

Analogously, the projections from the direct product onto the input tables are stored, and can be distinguished by the `specification` components.

The attribute `FactorsOfDirectProduct` (71.20.2) is set to the lists of arguments.

The `*` operator for two character tables (see 71.7) delegates to `CharacterTableDirectProduct`.

Example

```
gap> c2:= CharacterTable( "Cyclic", 2 );;
gap> s3:= CharacterTable( "Symmetric", 3 );;
gap> Display( CharacterTableDirectProduct( c2, s3 ) );
```

```
C2xSym(3)

     2  2  2  1  2  2  1
     3  1  .  1  1  .  1

       1a 2a 3a 2b 2c 6a
    2P 1a 1a 3a 1a 1a 3a
    3P 1a 2a 1a 2b 2c 2b

X.1     1 -1  1  1 -1  1
X.2     2  . -1  2  . -1
X.3     1  1  1  1  1  1
X.4     1 -1  1 -1  1 -1
X.5     2  . -1 -2  .  1
X.6     1  1  1 -1 -1 -1
```

71.20.2 FactorsOfDirectProduct

▷ `FactorsOfDirectProduct(`*tbl*`)` (attribute)

For an ordinary character table that has been constructed via `CharacterTableDirectProduct` (71.20.1), the value of `FactorsOfDirectProduct` is the list of arguments in the `CharacterTableDirectProduct` (71.20.1) call.

Note that there is no default method for *computing* the value of `FactorsOfDirectProduct`.

71.20.3 CharacterTableFactorGroup

▷ `CharacterTableFactorGroup(`*tbl, classes*`)` (operation)

is the character table of the factor group of the ordinary character table *tbl* by the normal closure of the classes whose positions are contained in the list *classes*.

The `/` operator for a character table and a list of class positions (see 71.7) delegates to `CharacterTableFactorGroup`.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfNormalSubgroups( s4 );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1 .. 5 ] ]
gap> f:= CharacterTableFactorGroup( s4, [ 3 ] );
CharacterTable( "Sym(4)/[ 1, 3 ]" )
gap> Display( f );
Sym(4)/[ 1, 3 ]

     2  1  1  .
     3  1  .  1

       1a 2a 3a
    2P 1a 1a 3a
    3P 1a 2a 1a

X.1     1 -1  1
```

```
X.2     2  . -1
X.3     1  1  1
```

71.20.4 CharacterTableIsoclinic

▷ CharacterTableIsoclinic(*tbl[, classes][, centre]*) (operation)
▷ SourceOfIsoclinicTable(*tbl*) (attribute)

If *tbl* is the (ordinary or modular) character table of a group with the structure 2.*G*.2 with a central subgroup *Z* of order 2 or 4 and a normal subgroup *N* of index 2 that contains *Z* then CharacterTableIsoclinic returns the table of the isoclinic group in the sense of the Atlas of Finite Groups [CCN+85, Chapter 6, Section 7]. If *N* is not uniquely determined then the positions of the classes forming *N* must be entered as list *classes*. If *Z* is not unique inside *N* then the positions of the classes in *Z* must be entered as list *centre*; If *Z* has order 2 then *centre* can be also the position of the involution in *Z*.

Note that also if *tbl* is a Brauer table then *classes* and *centre* denote class numbers w.r.t. the *ordinary* character table.

For an ordinary character table that has been constructed via CharacterTableIsoclinic, the value of SourceOfIsoclinicTable is the list of three arguments in the CharacterTableIsoclinic call.

Note that there is no default method for *computing* the value of SourceOfIsoclinicTable.

Example

```
gap> d8:= CharacterTable( "Dihedral", 8 );
CharacterTable( "Dihedral(8)" )
gap> nsg:= ClassPositionsOfNormalSubgroups( d8 );
[ [ 1 ], [ 1, 3 ], [ 1 .. 3 ], [ 1, 3, 4 ], [ 1, 3 .. 5 ], [ 1 .. 5 ]
 ]
gap> iso:= CharacterTableIsoclinic( d8, nsg[3] );;
gap> Display( iso );
Isoclinic(Dihedral(8))

     2  3  2  3  2  2

       1a 4a 2a 4b 4c
    2P 1a 2a 1a 2a 2a

X.1     1  1  1  1  1
X.2     1  1  1 -1 -1
X.3     1 -1  1  1 -1
X.4     1 -1  1 -1  1
X.5     2  . -2  .  .
gap> SourceOfIsoclinicTable( iso );
[ CharacterTable( "Dihedral(8)" ), [ 1, 2, 3 ], [ 3 ], 3 ]
```

71.20.5 CharacterTableWreathSymmetric

▷ CharacterTableWreathSymmetric(*tbl, n*) (function)

returns the character table of the wreath product of a group G with the full symmetric group on n points, where *tbl* is the character table of G.

The result has values for `ClassParameters` (71.9.7) and `CharacterParameters` (71.9.7) stored, the entries in these lists are sequences of partitions. Note that this parametrization prevents the principal character from being the first one in the list of irreducibles.

Example

```
gap> c3:= CharacterTable( "Cyclic", 3 );;
gap> wr:= CharacterTableWreathSymmetric( c3, 2 );;
gap> Display( wr );
C3wrS2

     2  1   .   .   1  .   1  1   1   1
     3  2   2   2   2  2   2  1   1   1

        1a  3a  3b  3c 3d  3e 2a  6a  6b
    2P  1a  3b  3a  3e 3d  3c 1a  3c  3e
    3P  1a  1a  1a  1a 1a  1a 2a  2a  2a

X.1     1   1   1   1  1   1 -1  -1  -1
X.2     2   A  /A   B -1  /B  .   .   .
X.3     2  /A   A  /B -1   B  .   .   .
X.4     1 -/A  -A  -A  1 -/A -1  /A   A
X.5     2  -1  -1   2 -1   2  .   .   .
X.6     1  -A -/A -/A  1  -A -1   A  /A
X.7     1   1   1   1  1   1  1   1   1
X.8     1 -/A  -A  -A  1 -/A  1 -/A  -A
X.9     1  -A -/A -/A  1  -A  1  -A -/A

A = -E(3)^2
  = (1+Sqrt(-3))/2 = 1+b3
B = 2*E(3)
  = -1+Sqrt(-3) = 2b3
gap> CharacterParameters( wr )[1];
[ [ 1, 1 ], [  ], [  ] ]
```

71.21 Sorted Character Tables

71.21.1 CharacterTableWithSortedCharacters

▷ `CharacterTableWithSortedCharacters(`*tbl[, perm]*`)` (operation)

is a character table that differs from *tbl* only by the succession of its irreducible characters. This affects the values of the attributes `Irr` (71.8.2) and `CharacterParameters` (71.9.7). Namely, these lists are permuted by the permutation *perm*.

If no second argument is given then a permutation is used that yields irreducible characters of increasing degree for the result. For the succession of characters in the result, see `SortedCharacters` (71.21.2).

The result has all those attributes and properties of *tbl* that are stored in `SupportedCharacterTableInfo` (71.3.4) and do not depend on the ordering of characters.

71.21.2 SortedCharacters

▷ SortedCharacters(*tbl, chars[, flag]*) (operation)

is a list containing the characters *chars*, ordered as specified by the other arguments.

There are three possibilities to sort characters: They can be sorted according to ascending norms (*flag* is the string "norm"), to ascending degree (*flag* is the string "degree"), or both (no third argument is given), i.e., characters with same norm are sorted according to ascending degree, and characters with smaller norm precede those with bigger norm.

Rational characters in the result precede other ones with same norm and/or same degree.

The trivial character, if contained in *chars*, will always be sorted to the first position.

71.21.3 CharacterTableWithSortedClasses

▷ CharacterTableWithSortedClasses(*tbl[, flag]*) (operation)

is a character table obtained by permutation of the classes of *tbl*. If the second argument *flag* is the string "centralizers" then the classes of the result are sorted according to descending centralizer orders. If the second argument is the string "representatives" then the classes of the result are sorted according to ascending representative orders. If no second argument is given then the classes of the result are sorted according to ascending representative orders, and classes with equal representative orders are sorted according to descending centralizer orders.

If the second argument is a permutation then the classes of the result are sorted by application of this permutation.

The result has all those attributes and properties of *tbl* that are stored in SupportedCharacterTableInfo (71.3.4) and do not depend on the ordering of classes.

71.21.4 SortedCharacterTable (w.r.t. a normal subgroup)

▷ SortedCharacterTable(*tbl, kernel*) (function)

▷ SortedCharacterTable(*tbl, normalseries*) (function)

▷ SortedCharacterTable(*tbl, facttbl, kernel*) (function)

is a character table obtained on permutation of the classes and the irreducibles characters of *tbl*.

The first form sorts the classes at positions contained in the list *kernel* to the beginning, and sorts all characters in Irr(*tbl*) such that the first characters are those that contain *kernel* in their kernel.

The second form does the same successively for all kernels k_i in the list *normalseries* $= [k_1, k_2, \ldots, k_n]$ where k_i must be a sublist of k_{i+1} for $1 \leq i \leq n-1$.

The third form computes the table F of the factor group of *tbl* modulo the normal subgroup formed by the classes whose positions are contained in the list *kernel*; F must be permutation equivalent to the table *facttbl*, in the sense of TransformingPermutationsCharacterTables (71.22.4), otherwise fail is returned. The classes of *tbl* are sorted such that the preimages of a class of F are consecutive, and that the succession of preimages is that of *facttbl*. The Irr (71.8.2) value of *tbl* is sorted as with SortCharTable(*tbl*, *kernel*).

(*Note* that the transformation is only unique up to table automorphisms of F, and this need not be unique up to table automorphisms of *tbl*.)

All rearrangements of classes and characters are stable, i.e., the relative positions of classes and characters that are not distinguished by any relevant property is not changed.

The result has all those attributes and properties of *tbl* that are stored in SupportedCharacterTableInfo (71.3.4) and do not depend on the ordering of classes and characters.

The ClassPermutation (71.21.5) value of *tbl* is changed if necessary, see 71.5.

SortedCharacterTable uses CharacterTableWithSortedClasses (71.21.3) and CharacterTableWithSortedCharacters (71.21.1).

71.21.5 ClassPermutation

▷ ClassPermutation(*tbl*) (attribute)

is a permutation π of classes of the character table *tbl*. If it is stored then class fusions into *tbl* that are stored on other tables must be followed by π in order to describe the correct fusion.

This attribute value is bound only if *tbl* was obtained from another table by permuting the classes, using CharacterTableWithSortedClasses (71.21.3) or SortedCharacterTable (71.21.4).

It is necessary because the original table and the sorted table have the same identifier (and the same group if known), and hence the same fusions are valid for the two tables.

Example

```
gap> tbl:= CharacterTable( "Symmetric", 4 );
CharacterTable( "Sym(4)" )
gap> Display( tbl );
Sym(4)

     2  3  2  3  .  2
     3  1  .  .  1  .

       1a 2a 2b 3a 4a
    2P 1a 1a 1a 3a 2b
    3P 1a 2a 2b 1a 4a

X.1     1 -1  1  1 -1
X.2     3 -1 -1  .  1
X.3     2  .  2 -1  .
X.4     3  1 -1  . -1
X.5     1  1  1  1  1
gap> srt1:= CharacterTableWithSortedCharacters( tbl );
CharacterTable( "Sym(4)" )
gap> List( Irr( srt1 ), Degree );
[ 1, 1, 2, 3, 3 ]
gap> srt2:= CharacterTableWithSortedClasses( tbl );
CharacterTable( "Sym(4)" )
gap> SizesCentralizers( tbl );
[ 24, 4, 8, 3, 4 ]
gap> SizesCentralizers( srt2 );
[ 24, 8, 4, 3, 4 ]
gap> nsg:= ClassPositionsOfNormalSubgroups( tbl );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1 .. 5 ] ]
gap> srt3:= SortedCharacterTable( tbl, nsg );
CharacterTable( "Sym(4)" )
```

```
gap> nsg:= ClassPositionsOfNormalSubgroups( srt3 );
[ [ 1 ], [ 1, 2 ], [ 1 .. 3 ], [ 1 .. 5 ] ]
gap> Display( srt3 );
Sym(4)

     2  3  3  .  2  2
     3  1  .  1  .  .

       1a 2a 3a 2b 4a
    2P 1a 1a 3a 1a 2a
    3P 1a 2a 1a 2b 4a

X.1     1  1  1  1  1
X.2     1  1  1 -1 -1
X.3     2  2 -1  .  .
X.4     3 -1  . -1  1
X.5     3 -1  .  1 -1
gap> ClassPermutation( srt3 );
(2,4,3)
```

71.22 Automorphisms and Equivalence of Character Tables

71.22.1 MatrixAutomorphisms

▷ MatrixAutomorphisms(*mat[, maps, subgroup]*) (operation)

For a matrix *mat*, MatrixAutomorphisms returns the group of those permutations of the columns of *mat* that leave the set of rows of *mat* invariant.

If the arguments *maps* and *subgroup* are given, only the group of those permutations is constructed that additionally fix each list in the list *maps* under pointwise action OnTuples (41.2.5), and *subgroup* is a permutation group that is known to be a subgroup of this group of automorphisms.

Each entry in *maps* must be a list of same length as the rows of *mat*. For example, if *mat* is a list of irreducible characters of a group then the list of element orders of the conjugacy classes (see OrdersClassRepresentatives (71.9.1)) may be an entry in *maps*.

71.22.2 TableAutomorphisms

▷ TableAutomorphisms(*tbl, characters[, info]*) (operation)

TableAutomorphisms returns the permutation group of those matrix automorphisms (see MatrixAutomorphisms (71.22.1)) of the list *characters* that leave the element orders (see OrdersClassRepresentatives (71.9.1)) and all stored power maps (see ComputedPowerMaps (73.1.1)) of the character table *tbl* invariant.

If *characters* is closed under Galois conjugacy –this is always fulfilled for the list of all irreducible characters of ordinary character tables– the string "closed" may be entered as the third argument *info*. Alternatively, a known subgroup of the table automorphisms can be entered as the third argument *info*.

The attribute AutomorphismsOfTable (71.9.4) can be used to compute and store the table automorphisms for the case that *characters* equals the Irr (71.8.2) value of *tbl*.

Example

```
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> irrd8:= Irr( tbld8 );
[ Character( CharacterTable( "Dihedral(8)" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 1, 1, 1, -1, -1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 1, -1, 1, 1, -1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 1, -1, 1, -1, 1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 2, 0, -2, 0, 0 ] ) ]
gap> orders:= OrdersClassRepresentatives( tbld8 );
[ 1, 4, 2, 2, 2 ]
gap> MatrixAutomorphisms( irrd8 );
Group([ (4,5), (2,4) ])
gap> MatrixAutomorphisms( irrd8, [ orders ], Group( () ) );
Group([ (4,5) ])
gap> TableAutomorphisms( tbld8, irrd8 );
Group([ (4,5) ])
```

71.22.3 TransformingPermutations

▷ TransformingPermutations(*mat1, mat2*) (operation)

Let *mat1* and *mat2* be matrices. TransformingPermutations tries to construct a permutation π that transforms the set of rows of the matrix *mat1* to the set of rows of the matrix *mat2* by permuting the columns.

If such a permutation exists, a record with the components columns, rows, and group is returned, otherwise fail. For TransformingPermutations(*mat1*, *mat2*) = *r* $\neq$ fail, we have *mat2* = Permuted(List(*mat1*, x -> Permuted(x, *r*.columns)), *r*.rows).

r.group is the group of matrix automorphisms of *mat2* (see MatrixAutomorphisms (71.22.1)). This group stabilizes the transformation in the sense that applying any of its elements to the columns of *mat2* preserves the set of rows of *mat2*.

71.22.4 TransformingPermutationsCharacterTables

▷ TransformingPermutationsCharacterTables(*tbl1, tbl2*) (operation)

Let *tbl1* and *tbl2* be character tables. TransformingPermutationsCharacterTables tries to construct a permutation π that transforms the set of rows of the matrix Irr(*tbl1*) to the set of rows of the matrix Irr(*tbl2*) by permuting the columns (see TransformingPermutations (71.22.3)), such that π transforms also the power maps and the element orders.

If such a permutation π exists then a record with the components columns (π), rows (the permutation of Irr(*tbl1*) corresponding to π), and group (the permutation group of table automorphisms of *tbl2*, see AutomorphismsOfTable (71.9.4)) is returned. If no such permutation exists, fail is returned.

Example

```
gap> tblq8:= CharacterTable( "Quaternionic", 8 );;
gap> irrq8:= Irr( tblq8 );
[ Character( CharacterTable( "Q8" ), [ 1, 1, 1, 1, 1 ] ),
```

```
    Character( CharacterTable( "Q8" ), [ 1, 1, 1, -1, -1 ] ),
    Character( CharacterTable( "Q8" ), [ 1, -1, 1, 1, -1 ] ),
    Character( CharacterTable( "Q8" ), [ 1, -1, 1, -1, 1 ] ),
    Character( CharacterTable( "Q8" ), [ 2, 0, -2, 0, 0 ] ) ]
  gap> OrdersClassRepresentatives( tblq8 );
  [ 1, 4, 2, 4, 4 ]
  gap> TransformingPermutations( irrd8, irrq8 );
  rec( columns := (), group := Group([ (4,5), (2,4) ]), rows := () )
  gap> TransformingPermutationsCharacterTables( tbld8, tblq8 );
  fail
  gap> tbld6:= CharacterTable( "Dihedral", 6 );;
  gap> tbls3:= CharacterTable( "Symmetric", 3 );;
  gap> TransformingPermutationsCharacterTables( tbld6, tbls3 );
  rec( columns := (2,3), group := Group(()), rows := (1,3,2) )
```

71.22.5 FamiliesOfRows

▷ `FamiliesOfRows(`*mat, maps*`)` (function)

distributes the rows of the matrix *mat* into families, as follows. Two rows of *mat* belong to the same family if there is a permutation of columns that maps one row to the other row. Each entry in the list *maps* is regarded to form a family of length 1.

`FamiliesOfRows` returns a record with the components

`famreps`
: the list of representatives for each family,

`permutations`
: the list that contains at position i a list of permutations that map the members of the family with representative `famreps`$[i]$ to that representative,

`families`
: the list that contains at position i the list of positions of members of the family of representative `famreps`$[i]$; (for the element *maps*$[i]$ the only member of the family will get the number `Length(` *mat* `) +` i).

71.23 Storing Normal Subgroup Information

71.23.1 NormalSubgroupClassesInfo

▷ `NormalSubgroupClassesInfo(`*tbl*`)` (attribute)

Let *tbl* be the ordinary character table of the group G. Many computations for group characters of G involve computations in normal subgroups or factor groups of G.

In some cases the character table *tbl* is sufficient; for example questions about a normal subgroup N of G can be answered if one knows the conjugacy classes that form N, e.g., the question whether a character of G restricts irreducibly to N. But other questions require the computation of N or even more information, like the character table of N.

In order to do these computations only once, one stores in the group a record with components to store normal subgroups, the corresponding lists of conjugacy classes, and (if necessary) the factor groups, namely

`nsg` list of normal subgroups of G, may be incomplete,

`nsgclasses`
at position i, the list of positions of conjugacy classes of *tbl* forming the i-th entry of the `nsg` component,

`nsgfactors`
at position i, if bound, the factor group modulo the i-th entry of the `nsg` component.

`NormalSubgroupClasses` (71.23.3), `FactorGroupNormalSubgroupClasses` (71.23.4), and `ClassPositionsOfNormalSubgroup` (71.23.2) each use these components, and they are the only functions to do so.

So if you need information about a normal subgroup for that you know the conjugacy classes, you should get it using `NormalSubgroupClasses` (71.23.3). If the normal subgroup was already used it is just returned, with all the knowledge it contains. Otherwise the normal subgroup is added to the lists, and will be available for the next call.

For example, if you are dealing with kernels of characters using the `KernelOfCharacter` (72.8.9) function you make use of this feature because `KernelOfCharacter` (72.8.9) calls `NormalSubgroupClasses` (71.23.3).

71.23.2 ClassPositionsOfNormalSubgroup

▷ `ClassPositionsOfNormalSubgroup(`*tbl, N*`)` (function)

is the list of positions of conjugacy classes of the character table *tbl* that are contained in the normal subgroup *N* of the underlying group of *tbl*.

71.23.3 NormalSubgroupClasses

▷ `NormalSubgroupClasses(`*tbl, classes*`)` (function)

returns the normal subgroup of the underlying group G of the ordinary character table *tbl* that consists of those conjugacy classes of *tbl* whose positions are in the list *classes*.

If `NormalSubgroupClassesInfo( `*tbl*` ).nsg` does not yet contain the required normal subgroup, and if `NormalSubgroupClassesInfo( `*tbl*` ).normalSubgroups` is bound then the result will be identical to the group in `NormalSubgroupClassesInfo( `*tbl*` ).normalSubgroups`.

71.23.4 FactorGroupNormalSubgroupClasses

▷ `FactorGroupNormalSubgroupClasses(`*tbl, classes*`)` (function)

is the factor group of the underlying group G of the ordinary character table *tbl* modulo the normal subgroup of G that consists of those conjugacy classes of *tbl* whose positions are in the list *classes*.

Example

```
gap> g:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> SetName( g, "S4" );
gap> tbl:= CharacterTable( g );
CharacterTable( S4 )
gap> irr:= Irr( g );
[ Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] ),
  Character( CharacterTable( S4 ), [ 3, -1, -1, 0, 1 ] ),
  Character( CharacterTable( S4 ), [ 2, 0, 2, -1, 0 ] ),
  Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] ),
  Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] ) ]
gap> kernel:= KernelOfCharacter( irr[3] );
Group([ (1,2)(3,4), (1,4)(2,3) ])
gap> HasNormalSubgroupClassesInfo( tbl );
true
gap> NormalSubgroupClassesInfo( tbl );
rec( nsg := [ Group([ (1,2)(3,4), (1,4)(2,3) ]) ],
  nsgclasses := [ [ 1, 3 ] ], nsgfactors := [  ] )
gap> ClassPositionsOfNormalSubgroup( tbl, kernel );
[ 1, 3 ]
gap> FactorGroupNormalSubgroupClasses( tbl, [ 1, 3 ] );
Group([ f1, f2 ])
gap> NormalSubgroupClassesInfo( tbl );
rec( nsg := [ Group([ (1,2)(3,4), (1,4)(2,3) ]) ],
  nsgclasses := [ [ 1, 3 ] ], nsgfactors := [ Group([ f1, f2 ]) ] )
```

Chapter 72

Class Functions

This chapter describes operations for *class functions of finite groups*. For operations concerning *character tables*, see Chapter 71.

Several examples in this chapter require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

Example

```
gap> LoadPackage( "ctbllib" );
true
```

72.1 Why Class Functions?

In principle it is possible to represent group characters or more general class functions by the plain lists of their values, and in fact many operations for class functions work with plain lists of class function values. But this has two disadvantages.

First, it is then necessary to regard a values list explicitly as a class function of a particular character table, by supplying this character table as an argument. In practice this means that with this setup, the user has the task to put the objects into the right context. For example, forming the scalar product or the tensor product of two class functions or forming an induced class function or a conjugate class function then needs three arguments in this case; this is particularly inconvenient in cases where infix operations cannot be used because of the additional argument, as for tensor products and induced class functions.

Second, when one says that "χ is a character of a group G" then this object χ carries a lot of information. χ has certain properties such as being irreducible or not. Several subgroups of G are related to χ, such as the kernel and the centre of χ. Other attributes of characters are the determinant and the central character. This knowledge cannot be stored in a plain list.

For dealing with a group together with its characters, and maybe also subgroups and their characters, it is desirable that GAP keeps track of the interpretation of characters. On the other hand, for using characters without accessing their groups, such as characters of tables from the GAP table library, dealing just with values lists is often sufficient. In particular, if one deals with incomplete character tables then it is often necessary to specify the arguments explicitly, for example one has to choose a fusion map or power map from a set of possibilities.

The main idea behind class function objects is that a class function object is equal to its values list in the sense of \= (31.11.1), so class function objects can be used wherever their values lists can be used, but there are operations for class function objects that do not work just with values lists.

GAP library functions prefer to return class function objects rather than returning just values lists, for example `Irr` (71.8.2) lists consist of class function objects, and `TrivialCharacter` (72.7.1) returns a class function object.

Here is an *example* that shows both approaches. First we define some groups.

Example

```
gap> S4:= SymmetricGroup( 4 );;  SetName( S4, "S4" );
gap> D8:= SylowSubgroup( S4, 2 );; SetName( D8, "D8" );
```

We do some computations using the functions described later in this Chapter, first with class function objects.

Example

```
gap> irrS4:= Irr( S4 );;
gap> irrD8:= Irr( D8 );;
gap> chi:= irrD8[4];
Character( CharacterTable( D8 ), [ 1, -1, 1, -1, 1 ] )
gap> chi * chi;
Character( CharacterTable( D8 ), [ 1, 1, 1, 1, 1 ] )
gap> ind:= chi ^ S4;
Character( CharacterTable( S4 ), [ 3, -1, -1, 0, 1 ] )
gap> List( irrS4, x -> ScalarProduct( x, ind ) );
[ 0, 1, 0, 0, 0 ]
gap> det:= Determinant( ind );
Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] )
gap> cent:= CentralCharacter( ind );
ClassFunction( CharacterTable( S4 ), [ 1, -2, -1, 0, 2 ] )
gap> rest:= Restricted( cent, D8 );
ClassFunction( CharacterTable( D8 ), [ 1, -2, -1, -1, 2 ] )
```

Now we repeat these calculations with plain lists of character values. Here we need the character tables in some places.

Example

```
gap> tS4:= CharacterTable( S4 );;
gap> tD8:= CharacterTable( D8 );;
gap> chi:= ValuesOfClassFunction( irrD8[4] );
[ 1, -1, 1, -1, 1 ]
gap> Tensored( [ chi ], [ chi ] )[1];
[ 1, 1, 1, 1, 1 ]
gap> ind:= InducedClassFunction( tD8, chi, tS4 );
ClassFunction( CharacterTable( S4 ), [ 3, -1, -1, 0, 1 ] )
gap> List( Irr( tS4 ), x -> ScalarProduct( tS4, x, ind ) );
[ 0, 1, 0, 0, 0 ]
gap> det:= DeterminantOfCharacter( tS4, ind );
ClassFunction( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] )
gap> cent:= CentralCharacter( tS4, ind );
ClassFunction( CharacterTable( S4 ), [ 1, -2, -1, 0, 2 ] )
gap> rest:= Restricted( tS4, cent, tD8 );
ClassFunction( CharacterTable( D8 ), [ 1, -2, -1, -1, 2 ] )
```

If one deals with character tables from the GAP table library then one has no access to their groups, but often the tables provide enough information for computing induced or restricted class

functions, symmetrizations etc., because the relevant class fusions and power maps are often stored on library tables. In these cases it is possible to use the tables instead of the groups as arguments. (If necessary information is not uniquely determined by the tables then an error is signalled.)

Example

```
gap> s5 := CharacterTable( "A5.2" );; irrs5 := Irr( s5  );;
gap> m11:= CharacterTable( "M11"  );; irrm11:= Irr( m11 );;
gap> chi:= TrivialCharacter( s5 );
Character( CharacterTable( "A5.2" ), [ 1, 1, 1, 1, 1, 1, 1 ] )
gap> chi ^ m11;
Character( CharacterTable( "M11" ), [ 66, 10, 3, 2, 1, 1, 0, 0, 0, 0
 ] )
gap> Determinant( irrs5[4] );
Character( CharacterTable( "A5.2" ), [ 1, 1, 1, 1, -1, -1, -1 ] )
```

Functions that compute *normal* subgroups related to characters have counterparts that return the list of class positions corresponding to these groups.

Example

```
gap> ClassPositionsOfKernel( irrs5[2] );
[ 1, 2, 3, 4 ]
gap> ClassPositionsOfCentre( irrs5[2] );
[ 1, 2, 3, 4, 5, 6, 7 ]
```

Non-normal subgroups cannot be described this way, so for example inertia subgroups (see `InertiaSubgroup` (72.8.13)) can in general not be computed from character tables without access to their groups.

72.1.1 IsClassFunction

▷ `IsClassFunction(`*obj*`)` (Category)

A *class function* (in characteristic p) of a finite group G is a map from the set of (p-regular) elements in G to the field of cyclotomics that is constant on conjugacy classes of G.

Each class function in GAP is represented by an *immutable list*, where at the i-th position the value on the i-th conjugacy class of the character table of G is stored. The ordering of the conjugacy classes is the one used in the underlying character table. Note that if the character table has access to its underlying group then the ordering of conjugacy classes in the group and in the character table may differ (see 71.6); class functions always refer to the ordering of classes in the character table.

Class function objects in GAP are not just plain lists, they store the character table of the group G as value of the attribute `UnderlyingCharacterTable` (72.2.1). The group G itself is accessible only via the character table and thus only if the character table stores its group, as value of the attribute `UnderlyingGroup` (71.6.1). The reason for this is that many computations with class functions are possible without using their groups, for example class functions of character tables in the GAP character table library do in general not have access to their underlying groups.

There are (at least) two reasons why class functions in GAP are *not* implemented as mappings. First, we want to distinguish class functions in different characteristics, for example to be able to define the Frobenius character of a given Brauer character; viewed as mappings, the trivial characters in all characteristics coprime to the order of G are equal. Second, the product of two class functions shall be again a class function, whereas the product of general mappings is defined as composition.

A further argument is that the typical operations for mappings such as `Image` (32.4.6) and `PreImage` (32.5.6) play no important role for class functions.

72.2 Basic Operations for Class Functions

Basic operations for class functions are `UnderlyingCharacterTable` (72.2.1), `ValuesOfClassFunction` (72.2.2), and the basic operations for lists (see 21.2).

72.2.1 UnderlyingCharacterTable

▷ `UnderlyingCharacterTable(`*psi*`)` (attribute)

For a class function *psi* of the group G, say, the character table of G is stored as value of `UnderlyingCharacterTable`. The ordering of entries in the list *psi* (see `ValuesOfClassFunction` (72.2.2)) refers to the ordering of conjugacy classes in this character table.

If *psi* is an ordinary class function then the underlying character table is the ordinary character table of G (see `OrdinaryCharacterTable` (71.8.4)), if *psi* is a class function in characteristic $p \neq 0$ then the underlying character table is the p-modular Brauer table of G (see `BrauerTable` (71.3.2)). So the underlying characteristic of *psi* can be read off from the underlying character table.

72.2.2 ValuesOfClassFunction

▷ `ValuesOfClassFunction(`*psi*`)` (attribute)

is the list of values of the class function *psi*, the i-th entry being the value on the i-th conjugacy class of the underlying character table (see `UnderlyingCharacterTable` (72.2.1)).

Example

```
gap> g:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> psi:= TrivialCharacter( g );
Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] )
gap> UnderlyingCharacterTable( psi );
CharacterTable( Sym( [ 1 .. 4 ] ) )
gap> ValuesOfClassFunction( psi );
[ 1, 1, 1, 1, 1 ]
gap> IsList( psi );
true
gap> psi[1];
1
gap> Length( psi );
5
gap> IsBound( psi[6] );
false
gap> Concatenation( psi, [ 2, 3 ] );
[ 1, 1, 1, 1, 1, 2, 3 ]
```

72.3 Comparison of Class Functions

With respect to \= (31.11.1) and \< (31.11.1), class functions behave equally to their lists of values (see ValuesOfClassFunction (72.2.2)). So two class functions are equal if and only if their lists of values are equal, no matter whether they are class functions of the same character table, of the same group but w.r.t. different class ordering, or of different groups.

Example

```
gap> grps:= Filtered( AllSmallGroups( 8 ), g -> not IsAbelian( g ) );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
gap> t1:= CharacterTable( grps[1] );  SetName( t1, "t1" );
CharacterTable( <pc group of size 8 with 3 generators> )
gap> t2:= CharacterTable( grps[2] );  SetName( t2, "t2" );
CharacterTable( <pc group of size 8 with 3 generators> )
gap> irr1:= Irr( grps[1] );
[ Character( t1, [ 1, 1, 1, 1, 1 ] ),
  Character( t1, [ 1, -1, -1, 1, 1 ] ),
  Character( t1, [ 1, -1, 1, 1, -1 ] ),
  Character( t1, [ 1, 1, -1, 1, -1 ] ),
  Character( t1, [ 2, 0, 0, -2, 0 ] ) ]
gap> irr2:= Irr( grps[2] );
[ Character( t2, [ 1, 1, 1, 1, 1 ] ),
  Character( t2, [ 1, -1, -1, 1, 1 ] ),
  Character( t2, [ 1, -1, 1, 1, -1 ] ),
  Character( t2, [ 1, 1, -1, 1, -1 ] ),
  Character( t2, [ 2, 0, 0, -2, 0 ] ) ]
gap> irr1 = irr2;
true
gap> IsSSortedList( irr1 );
false
gap> irr1[1] < irr1[2];
false
gap> irr1[2] < irr1[3];
true
```

72.4 Arithmetic Operations for Class Functions

Class functions are *row vectors* of cyclotomics. The *additive* behaviour of class functions is defined such that they are equal to the plain lists of class function values except that the results are represented again as class functions whenever this makes sense. The *multiplicative* behaviour, however, is different. This is motivated by the fact that the tensor product of class functions is a more interesting operation than the vector product of plain lists. (Another candidate for a multiplication of compatible class functions would have been the inner product, which is implemented via the function ScalarProduct (72.8.5). In terms of filters, the arithmetic of class functions is based on the decision that they lie in IsGeneralizedRowVector (21.12.1), with additive nesting depth 1, but they do *not* lie in IsMultiplicativeGeneralizedRowVector (21.12.2).

More specifically, the scalar multiple of a class function with a cyclotomic is a class function, and the sum and the difference of two class functions of the same underlying character table (see UnderlyingCharacterTable (72.2.1)) are again class functions of this table. The sum and the

difference of a class function and a list that is *not* a class function are plain lists, as well as the sum and the difference of two class functions of different character tables.

Example

```
gap> g:= SymmetricGroup( 4 );;  tbl:= CharacterTable( g );;
gap> SetName( tbl, "S4" );  irr:= Irr( g );
[ Character( S4, [ 1, -1, 1, 1, -1 ] ),
  Character( S4, [ 3, -1, -1, 0, 1 ] ),
  Character( S4, [ 2, 0, 2, -1, 0 ] ),
  Character( S4, [ 3, 1, -1, 0, -1 ] ),
  Character( S4, [ 1, 1, 1, 1, 1 ] ) ]
gap> 2 * irr[5];
Character( S4, [ 2, 2, 2, 2, 2 ] )
gap> irr[1] / 7;
ClassFunction( S4, [ 1/7, -1/7, 1/7, 1/7, -1/7 ] )
gap> lincomb:= irr[3] + irr[1] - irr[5];
VirtualCharacter( S4, [ 2, -2, 2, -1, -2 ] )
gap> lincomb:= lincomb + 2 * irr[5];
VirtualCharacter( S4, [ 4, 0, 4, 1, 0 ] )
gap> IsCharacter( lincomb );
true
gap> lincomb;
Character( S4, [ 4, 0, 4, 1, 0 ] )
gap> irr[5] + 2;
[ 3, 3, 3, 3, 3 ]
gap> irr[5] + [ 1, 2, 3, 4, 5 ];
[ 2, 3, 4, 5, 6 ]
gap> zero:= 0 * irr[1];
VirtualCharacter( S4, [ 0, 0, 0, 0, 0 ] )
gap> zero + Z(3);
[ Z(3), Z(3), Z(3), Z(3), Z(3) ]
gap> irr[5] + TrivialCharacter( DihedralGroup( 8 ) );
[ 2, 2, 2, 2, 2 ]
```

The product of two class functions of the same character table is the tensor product (pointwise product) of these class functions. Thus the set of all class functions of a fixed group forms a ring, and for any field F of cyclotomics, the F-span of a given set of class functions forms an algebra.

The product of two class functions of *different* tables and the product of a class function and a list that is *not* a class function are not defined, an error is signalled in these cases. Note that in this respect, class functions behave differently from their values lists, for which the product is defined as the standard scalar product.

Example

```
gap> tens:= irr[3] * irr[4];
Character( S4, [ 6, 0, -2, 0, 0 ] )
gap> ValuesOfClassFunction( irr[3] ) * ValuesOfClassFunction( irr[4] );
4
```

Class functions without zero values are invertible, the *inverse* is defined pointwise. As a consequence, for example groups of linear characters can be formed.

Example

```
gap> tens / irr[1];
Character( S4, [ 6, 0, -2, 0, 0 ] )
```

Other (somewhat strange) implications of the definition of arithmetic operations for class functions, together with the general rules of list arithmetic (see 21.11), apply to the case of products involving *lists* of class functions. No inverse of the list of irreducible characters as a matrix is defined; if one is interested in the inverse matrix then one can compute it from the matrix of class function values.

Example

```
gap> Inverse( List( irr, ValuesOfClassFunction ) );
[ [ 1/24, 1/8, 1/12, 1/8, 1/24 ], [ -1/4, -1/4, 0, 1/4, 1/4 ],
  [ 1/8, -1/8, 1/4, -1/8, 1/8 ], [ 1/3, 0, -1/3, 0, 1/3 ],
  [ -1/4, 1/4, 0, -1/4, 1/4 ] ]
```

Also the product of a class function with a list of class functions is *not* a vector-matrix product but the list of pointwise products.

Example

```
gap> irr[1] * irr{ [ 1 .. 3 ] };
[ Character( S4, [ 1, 1, 1, 1, 1 ] ),
  Character( S4, [ 3, 1, -1, 0, -1 ] ),
  Character( S4, [ 2, 0, 2, -1, 0 ] ) ]
```

And the product of two lists of class functions is *not* the matrix product but the sum of the pointwise products.

Example

```
gap> irr * irr;
Character( S4, [ 24, 4, 8, 3, 4 ] )
```

The *powering* operator `\^` (31.12.1) has several meanings for class functions. The power of a class function by a nonnegative integer is clearly the tensor power. The power of a class function by an element that normalizes the underlying group or by a Galois automorphism is the conjugate class function. (As a consequence, the application of the permutation induced by such an action cannot be denoted by `\^` (31.12.1); instead one can use `Permuted` (21.20.18).) The power of a class function by a group or a character table is the induced class function (see `InducedClassFunction` (72.9.3)). The power of a group element by a class function is the class function value at (the conjugacy class containing) this element.

Example

```
gap> irr[3] ^ 3;
Character( S4, [ 8, 0, 8, -1, 0 ] )
gap> lin:= LinearCharacters( DerivedSubgroup( g ) );
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
    [ 1, 1, E(3)^2, E(3) ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
    [ 1, 1, E(3), E(3)^2 ] ) ]
gap> List( lin, chi -> chi ^ (1,2) );
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
    [ 1, 1, E(3), E(3)^2 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
    [ 1, 1, E(3)^2, E(3) ] ) ]
gap> Orbit( GaloisGroup( CF(3) ), lin[2] );
```

```
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
    [ 1, 1, E(3)^2, E(3) ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
    [ 1, 1, E(3), E(3)^2 ] ) ]
gap> lin[1]^g;
Character( S4, [ 2, 0, 2, 2, 0 ] )
gap> (1,2,3)^lin[2];
E(3)^2
```

72.4.1 Characteristic (for a class function)

▷ `Characteristic(`*chi*`)` (function)

The *characteristic* of class functions is zero, as for all list of cyclotomics. For class functions of a p-modular character table, such as Brauer characters, the prime p is given by the `UnderlyingCharacteristic` (71.9.5) value of the character table.

Example

```
gap> Characteristic( irr[1] );
0
gap> irrmod2:= Irr( g, 2 );
[ Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 1, 1 ] ),
  Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 2, -1 ] ) ]
gap> Characteristic( irrmod2[1] );
0
gap> UnderlyingCharacteristic( UnderlyingCharacterTable( irrmod2[1] ) );
2
```

72.4.2 ComplexConjugate (for a class function)

▷ `ComplexConjugate(`*chi*`)` (function)
▷ `GaloisCyc(`*chi*`, `*k*`)` (function)
▷ `Permuted(`*chi*`, `*pi*`)` (function)

The operations `ComplexConjugate`, `GaloisCyc`, and `Permuted` return a class function when they are called with a class function; The complex conjugate of a class function that is known to be a (virtual) character is again known to be a (virtual) character, and applying an arbitrary Galois automorphism to an ordinary (virtual) character yields a (virtual) character.

Example

```
gap> ComplexConjugate( lin[2] );
Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2
 ] )
gap> GaloisCyc( lin[2], 5 );
Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2
 ] )
gap> Permuted( lin[2], (2,3,4) );
ClassFunction( CharacterTable( Alt( [ 1 .. 4 ] ) ),
[ 1, E(3), 1, E(3)^2 ] )
```

72.4.3 Order (for a class function)

▷ `Order(`*chi*`)` (function)

By definition of `Order` (31.10.10) for arbitrary monoid elements, the return value of `Order` (31.10.10) for a character must be its multiplicative order. The *determinantal order* (see `DeterminantOfCharacter` (72.8.18)) of a character *chi* can be computed as `Order( Determinant(` *chi* `) )`.

Example

```
gap> det:= Determinant( irr[3] );
Character( S4, [ 1, -1, 1, 1, -1 ] )
gap> Order( det );
2
```

72.5 Printing Class Functions

72.5.1 ViewObj (for class functions)

▷ `ViewObj(`*chi*`)` (method)

The default `ViewObj` (6.3.5) methods for class functions print one of the strings `"ClassFunction"`, `"VirtualCharacter"`, `"Character"` (depending on whether the class function is known to be a character or virtual character, see `IsCharacter` (72.8.1), `IsVirtualCharacter` (72.8.2)), followed by the `ViewObj` (6.3.5) output for the underlying character table (see 71.13), and the list of values. The table is chosen (and not the group) in order to distinguish class functions of different underlying characteristic (see `UnderlyingCharacteristic` (71.9.5)).

72.5.2 PrintObj (for class functions)

▷ `PrintObj(`*chi*`)` (method)

The default `PrintObj` (6.3.5) method for class functions does the same as `ViewObj` (6.3.5), except that the character table is is `Print` (6.3.4)-ed instead of `View` (6.3.3)-ed.

Note that if a class function is shown only with one of the strings `"ClassFunction"`, `"VirtualCharacter"`, it may still be that it is in fact a character; just this was not known at the time when the class function was printed.

In order to reduce the space that is needed to print a class function, it may be useful to give a name (see `Name` (12.8.2)) to the underlying character table.

72.5.3 Display (for class functions)

▷ `Display(`*chi*`)` (method)

The default `Display` (6.3.6) method for a class function *chi* calls `Display` (6.3.6) for its underlying character table (see 71.13), with *chi* as the only entry in the `chars` list of the options record.

Example

```
gap> chi:= TrivialCharacter( CharacterTable( "A5" ) );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
```

```
gap> Display( chi );
A5

     2  2  2  .  .  .
     3  1  .  1  .  .
     5  1  .  .  1  1

       1a 2a 3a 5a 5b
    2P 1a 1a 3a 5b 5a
    3P 1a 2a 1a 5b 5a
    5P 1a 2a 3a 1a 1a

Y.1     1  1  1  1  1
```

72.6 Creating Class Functions from Values Lists

72.6.1 ClassFunction (for a character table and a list)

▷ `ClassFunction(`*tbl*`, `*values*`)` (operation)
▷ `ClassFunction(`*G*`, `*values*`)` (operation)

In the first form, `ClassFunction` returns the class function of the character table *tbl* with values given by the list *values* of cyclotomics. In the second form, *G* must be a group, and the class function of its ordinary character table is returned.

Note that *tbl* determines the underlying characteristic of the returned class function (see `UnderlyingCharacteristic` (71.9.5)).

72.6.2 VirtualCharacter (for a character table and a list)

▷ `VirtualCharacter(`*tbl*`, `*values*`)` (operation)
▷ `VirtualCharacter(`*G*`, `*values*`)` (operation)

`VirtualCharacter` returns the virtual character (see `IsVirtualCharacter` (72.8.2)) of the character table *tbl* or the group *G*, respectively, with values given by the list *values*.

It is *not* checked whether the given values really describe a virtual character.

72.6.3 Character (for a character table and a list)

▷ `Character(`*tbl*`, `*values*`)` (operation)
▷ `Character(`*G*`, `*values*`)` (operation)

`Character` returns the character (see `IsCharacter` (72.8.1)) of the character table *tbl* or the group *G*, respectively, with values given by the list *values*.

It is *not* checked whether the given values really describe a character.

Example

```
gap> g:= DihedralGroup( 8 );  tbl:= CharacterTable( g );
<pc group of size 8 with 3 generators>
CharacterTable( <pc group of size 8 with 3 generators> )
gap> SetName( tbl, "D8" );
```

```
gap> phi:= ClassFunction( g, [ 1, -1, 0, 2, -2 ] );
ClassFunction( D8, [ 1, -1, 0, 2, -2 ] )
gap> psi:= ClassFunction( tbl,
>              List( Irr( g ), chi -> ScalarProduct( chi, phi ) ) );
ClassFunction( D8, [ -3/8, 9/8, 5/8, 1/8, -1/4 ] )
gap> chi:= VirtualCharacter( g, [ 0, 0, 8, 0, 0 ] );
VirtualCharacter( D8, [ 0, 0, 8, 0, 0 ] )
gap> reg:= Character( tbl, [ 8, 0, 0, 0, 0 ] );
Character( D8, [ 8, 0, 0, 0, 0 ] )
```

72.6.4 ClassFunctionSameType

▷ `ClassFunctionSameType(`*tbl, chi, values*`)` (function)

Let *tbl* be a character table, *chi* a class function object (*not* necessarily a class function of *tbl*), and *values* a list of cyclotomics. `ClassFunctionSameType` returns the class function ψ of *tbl* with values list *values*, constructed with `ClassFunction` (72.6.1).

If *chi* is known to be a (virtual) character then ψ is also known to be a (virtual) character.

Example

```
gap> h:= Centre( g );;
gap> centbl:= CharacterTable( h );;  SetName( centbl, "C2" );
gap> ClassFunctionSameType( centbl, phi, [ 1, 1 ] );
ClassFunction( C2, [ 1, 1 ] )
gap> ClassFunctionSameType( centbl, chi, [ 1, 1 ] );
VirtualCharacter( C2, [ 1, 1 ] )
gap> ClassFunctionSameType( centbl, reg, [ 1, 1 ] );
Character( C2, [ 1, 1 ] )
```

72.7 Creating Class Functions using Groups

72.7.1 TrivialCharacter

▷ `TrivialCharacter(`*tbl*`)` (attribute)
▷ `TrivialCharacter(`*G*`)` (attribute)

is the *trivial character* of the group *G* or its character table *tbl*, respectively. This is the class function with value equal to 1 for each class.

Example

```
gap> TrivialCharacter( CharacterTable( "A5" ) );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
gap> TrivialCharacter( SymmetricGroup( 3 ) );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 1, 1, 1 ] )
```

72.7.2 NaturalCharacter (for a group)

▷ `NaturalCharacter(`*G*`)` (attribute)
▷ `NaturalCharacter(`*hom*`)` (attribute)

If the argument is a permutation group *G* then NaturalCharacter returns the (ordinary) character of the natural permutation representation of *G* on the set of moved points (see MovedPoints (42.3.3)), that is, the value on each class is the number of points among the moved points of *G* that are fixed by any permutation in that class.

If the argument is a matrix group *G* in characteristic zero then NaturalCharacter returns the (ordinary) character of the natural matrix representation of *G*, that is, the value on each class is the trace of any matrix in that class.

If the argument is a group homomorphism *hom* whose image is a permutation group or a matrix group then NaturalCharacter returns the restriction of the natural character of the image of *hom* to the preimage of *hom*.

Example

```
gap> NaturalCharacter( SymmetricGroup( 3 ) );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 3, 1, 0 ] )
gap> NaturalCharacter( Group( [ [ 0, -1 ], [ 1, -1 ] ] ) );
Character( CharacterTable( Group([ [ [ 0, -1 ], [ 1, -1 ] ] ]) ),
[ 2, -1, -1 ] )
gap> d8:= DihedralGroup( 8 );;  hom:= IsomorphismPermGroup( d8 );;
gap> NaturalCharacter( hom );
Character( CharacterTable( <pc group of size 8 with 3 generators> ),
[ 8, 0, 0, 0, 0 ] )
```

72.7.3 PermutationCharacter

▷ PermutationCharacter(*G, D, opr*) (operation)

▷ PermutationCharacter(*G, U*) (operation)

Called with a group *G*, an action domain or proper set *D*, and an action function *opr* (see Chapter 41), PermutationCharacter returns the *permutation character* of the action of *G* on *D* via *opr*, that is, the value on each class is the number of points in *D* that are fixed by an element in this class under the action *opr*.

If the arguments are a group *G* and a subgroup *U* of *G* then PermutationCharacter returns the permutation character of the action of *G* on the right cosets of *U* via right multiplication.

To compute the permutation character of a *transitive permutation group G* on the cosets of a point stabilizer *U*, the attribute NaturalCharacter (72.7.2) of *G* can be used instead of PermutationCharacter(*G*, *U*).

More facilities concerning permutation characters are the transitivity test (see Section 72.8) and several tools for computing possible permutation characters (see 72.13, 72.14).

Example

```
gap> PermutationCharacter( GL(2,2), AsSSortedList( GF(2)^2 ), OnRight );
Character( CharacterTable( SL(2,2) ), [ 4, 2, 1 ] )
gap> s3:= SymmetricGroup( 3 );;  a3:= DerivedSubgroup( s3 );;
gap> PermutationCharacter( s3, a3 );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 2, 0, 2 ] )
```

72.8 Operations for Class Functions

In the description of the following operations, the optional first argument *tbl* is needed only if the argument *chi* is a plain list and not a class function object. In this case, *tbl* must always be the

character table of which *chi* shall be regarded as a class function.

72.8.1 IsCharacter

▷ IsCharacter(*[tbl,]chi*) (property)

An *ordinary character* of a group G is a class function of G whose values are the traces of a complex matrix representation of G.

A *Brauer character* of G in characteristic p is a class function of G whose values are the complex lifts of a matrix representation of G with image a finite field of characteristic p.

72.8.2 IsVirtualCharacter

▷ IsVirtualCharacter(*[tbl,]chi*) (property)

A *virtual character* is a class function that can be written as the difference of two proper characters (see IsCharacter (72.8.1)).

72.8.3 IsIrreducibleCharacter

▷ IsIrreducibleCharacter(*[tbl,]chi*) (property)

A character is *irreducible* if it cannot be written as the sum of two characters. For ordinary characters this can be checked using the scalar product of class functions (see ScalarProduct (72.8.5)). For Brauer characters there is no generic method for checking irreducibility.

Example

```
gap> S4:= SymmetricGroup( 4 );;  SetName( S4, "S4" );
gap> psi:= ClassFunction( S4, [ 1, 1, 1, -2, 1 ] );
ClassFunction( CharacterTable( S4 ), [ 1, 1, 1, -2, 1 ] )
gap> IsVirtualCharacter( psi );
true
gap> IsCharacter( psi );
false
gap> chi:= ClassFunction( S4, SizesCentralizers( CharacterTable( S4 ) ) );
ClassFunction( CharacterTable( S4 ), [ 24, 4, 8, 3, 4 ] )
gap> IsCharacter( chi );
true
gap> IsIrreducibleCharacter( chi );
false
gap> IsIrreducibleCharacter( TrivialCharacter( S4 ) );
true
```

72.8.4 DegreeOfCharacter

▷ DegreeOfCharacter(*chi*) (attribute)

is the value of the character *chi* on the identity element. This can also be obtained as *chi*[1].

Example

```
gap> List( Irr( S4 ), DegreeOfCharacter );
[ 1, 3, 2, 3, 1 ]
gap> nat:= NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> nat[1];
4
```

72.8.5 ScalarProduct (for characters)

▷ ScalarProduct(*[tbl,]chi, psi*) (operation)

Returns: the scalar product of the class functions *chi* and *psi*, which belong to the same character table *tbl*.

If *chi* and *psi* are class function objects, the argument *tbl* is not needed, but *tbl* is necessary if at least one of *chi*, *psi* is just a plain list.

The scalar product of two *ordinary* class functions χ, ψ of a group G is defined as $(\sum_{g\in G}\chi(g)\psi(g^{-1}))/|G|$.

For two *p-modular* class functions, the scalar product is defined as $(\sum_{g\in S}\chi(g)\psi(g^{-1}))/|G|$, where S is the set of p-regular elements in G.

72.8.6 MatScalarProducts

▷ MatScalarProducts(*[tbl,]list[, list2]*) (operation)

Called with two lists *list*, *list2* of class functions of the same character table (which may be given as the argument *tbl*), MatScalarProducts returns the matrix of scalar products (see ScalarProduct (72.8.5)) More precisely, this matrix contains in the i-th row the list of scalar products of *list2*$[i]$ with the entries of *list*.

If only one list *list* of class functions is given then a lower triangular matrix of scalar products is returned, containing (for $j \leq i$) in the i-th row in column j the value ScalarProduct(*tbl*, *list*$[j]$, *list*$[i]$).

72.8.7 Norm (for a class function)

▷ Norm(*[tbl,]chi*) (attribute)

For an ordinary class function *chi* of the group G, say, we have *chi* $= \sum_{\chi\in Irr(G)} a_\chi\chi$, with complex coefficients a_χ. The *norm* of *chi* is defined as $\sum_{\chi\in Irr(G)} a_\chi\overline{a_\chi}$.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> ScalarProduct( TrivialCharacter( tbl ), Sum( Irr( tbl ) ) );
1
gap> ScalarProduct( tbl, [ 1, 1, 1, 1, 1 ], Sum( Irr( tbl ) ) );
1
gap> tbl2:= tbl mod 2;
BrauerTable( "A5", 2 )
gap> chi:= Irr( tbl2 )[1];
Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] )
gap> ScalarProduct( chi, chi );
```

```
3/4
gap> ScalarProduct( tbl2, [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ] );
3/4
gap> chars:= Irr( tbl ){ [ 2 .. 4 ] };;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> MatScalarProducts( Irr( tbl ), chars );
[ [ 0, 0, 0, 1, 1 ], [ 1, 1, 0, 0, 1 ], [ 1, 0, 1, 0, 1 ],
  [ 0, 1, 0, 1, 1 ], [ 0, 0, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ] ]
gap> MatScalarProducts( tbl, chars );
[ [ 2 ], [ 1, 3 ], [ 1, 2, 3 ], [ 2, 2, 1, 3 ], [ 2, 1, 2, 2, 3 ],
  [ 2, 3, 3, 3, 3, 5 ] ]
gap> List( chars, Norm );
[ 2, 3, 3, 3, 3, 5 ]
```

72.8.8 ConstituentsOfCharacter

▷ ConstituentsOfCharacter(*[tbl,]chi*) (attribute)

is the set of irreducible characters that occur in the decomposition of the (virtual) character *chi* with nonzero coefficient.

Example

```
gap> nat:= NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> ConstituentsOfCharacter( nat );
[ Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] ) ]
```

72.8.9 KernelOfCharacter

▷ KernelOfCharacter(*[tbl,]chi*) (attribute)

For a class function *chi* of the group G, say, KernelOfCharacter returns the normal subgroup of G that is formed by those conjugacy classes for which the value of *chi* equals the degree of *chi*. If the underlying character table of *chi* does not store the group G then an error is signalled. (See ClassPositionsOfKernel (72.8.10) for a way to handle the kernel implicitly, by listing the positions of conjugacy classes in the kernel.)

The returned group is the kernel of any representation of G that affords *chi*.

Example

```
gap> List( Irr( S4 ), KernelOfCharacter );
[ Alt( [ 1 .. 4 ] ), Group(()), Group([ (1,2)(3,4), (1,3)(2,4) ]),
  Group(()), Group([ (), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ]) ]
```

72.8.10 ClassPositionsOfKernel

▷ ClassPositionsOfKernel(*chi*) (attribute)

is the list of positions of those conjugacy classes that form the kernel of the character *chi*, that is, those positions with character value equal to the character degree.

Example

```
gap> List( Irr( S4 ), ClassPositionsOfKernel );
[ [ 1, 3, 4 ], [ 1 ], [ 1, 3 ], [ 1 ], [ 1, 2, 3, 4, 5 ] ]
```

72.8.11 CentreOfCharacter

▷ CentreOfCharacter(*[tbl,]chi*) (attribute)

For a character *chi* of the group G, say, CentreOfCharacter returns the *centre* of *chi*, that is, the normal subgroup of all those elements of G for which the quotient of the value of *chi* by the degree of *chi* is a root of unity.

If the underlying character table of *psi* does not store the group G then an error is signalled. (See ClassPositionsOfCentre (72.8.12) for a way to handle the centre implicitly, by listing the positions of conjugacy classes in the centre.)

Example

```
gap> List( Irr( S4 ), CentreOfCharacter );
[ Group([ (), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ]), Group(()),
  Group([ (1,2)(3,4), (1,3)(2,4) ]), Group(()),
  Group([ (), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ]) ]
```

72.8.12 ClassPositionsOfCentre (for a character)

▷ ClassPositionsOfCentre(*chi*) (attribute)

is the list of positions of classes forming the centre of the character *chi* (see CentreOfCharacter (72.8.11)).

Example

```
gap> List( Irr( S4 ), ClassPositionsOfCentre );
[ [ 1, 2, 3, 4, 5 ], [ 1 ], [ 1, 3 ], [ 1 ], [ 1, 2, 3, 4, 5 ] ]
```

72.8.13 InertiaSubgroup

▷ InertiaSubgroup(*[tbl,]G, chi*) (operation)

Let *chi* be a character of the group H, say, and *tbl* the character table of H; if the argument *tbl* is not given then the underlying character table of *chi* (see UnderlyingCharacterTable (72.2.1)) is used instead. Furthermore, let *G* be a group that contains H as a normal subgroup.

InertiaSubgroup returns the stabilizer in *G* of *chi*, w.r.t. the action of *G* on the classes of H via conjugation. In other words, InertiaSubgroup returns the group of all those elements $g \in G$ that satisfy $chi^g = chi$.

Example

```
gap> der:= DerivedSubgroup( S4 );
Alt( [ 1 .. 4 ] )
gap> List( Irr( der ), chi -> InertiaSubgroup( S4, chi ) );
[ S4, Alt( [ 1 .. 4 ] ), Alt( [ 1 .. 4 ] ), S4 ]
```

72.8.14 CycleStructureClass

▷ CycleStructureClass(*[tbl,]chi, class*) (operation)

Let *permchar* be a permutation character, and *class* be the position of a conjugacy class of the character table of *permchar*. CycleStructureClass returns a list describing the cycle structure of each element in class *class* in the underlying permutation representation, in the same format as the result of CycleStructurePerm (42.4.2).

Example

```
gap> nat:= NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> List( [ 1 .. 5 ], i -> CycleStructureClass( nat, i ) );
[ [  ], [ 1 ], [ 2 ], [ , 1 ], [ ,, 1 ] ]
```

72.8.15 IsTransitive (for a character)

▷ IsTransitive(*[tbl,]chi*) (property)

For a permutation character *chi* of the group G that corresponds to an action on the G-set Ω (see PermutationCharacter (72.7.3)), IsTransitive (41.10.1) returns true if the action of G on Ω is transitive, and false otherwise.

72.8.16 Transitivity (for a character)

▷ Transitivity(*[tbl,]chi*) (attribute)

For a permutation character *chi* of the group G that corresponds to an action on the G-set Ω (see PermutationCharacter (72.7.3)), Transitivity returns the maximal nonnegative integer k such that the action of G on Ω is k-transitive.

Example

```
gap> IsTransitive( nat );  Transitivity( nat );
true
4
gap> Transitivity( 2 * TrivialCharacter( S4 ) );
0
```

72.8.17 CentralCharacter

▷ CentralCharacter(*[tbl,]chi*) (attribute)

For a character *chi* of the group G, say, CentralCharacter returns the *central character* of *chi*.

The central character of χ is the class function ω_χ defined by $\omega_\chi(g) = |g^G| \cdot \chi(g)/\chi(1)$ for each $g \in G$.

72.8.18 DeterminantOfCharacter

▷ DeterminantOfCharacter(*[tbl,]chi*) (attribute)

`DeterminantOfCharacter` returns the *determinant character* of the character *chi*. This is defined to be the character obtained by taking the determinant of representing matrices of any representation affording *chi*; the determinant can be computed using `EigenvaluesChar` (72.8.19).

It is also possible to call `Determinant` (24.4.4) instead of `DeterminantOfCharacter`.

Note that the determinant character is well-defined for virtual characters.

Example

```
gap> CentralCharacter( TrivialCharacter( S4 ) );
ClassFunction( CharacterTable( S4 ), [ 1, 6, 3, 8, 6 ] )
gap> DeterminantOfCharacter( Irr( S4 )[3] );
Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] )
```

72.8.19 EigenvaluesChar

▷ `EigenvaluesChar(`*tbl*`, `*chi*`, `*class*`)` (operation)

Let *chi* be a character of the group G, say. For an element $g \in G$ in the *class*-th conjugacy class, of order n, let M be a matrix of a representation affording *chi*.

`EigenvaluesChar` returns the list of length n where at position k the multiplicity of $\mathrm{E}(n)^k = \exp(2\pi ik/n)$ as an eigenvalue of M is stored.

We have *chi*`[ `*class*` ] = List( [ 1 .. n ], k -> E(n)^k ) * EigenvaluesChar(` *tbl*`, `*chi*`, `*class*` )`.

It is also possible to call `Eigenvalues` (24.8.3) instead of `EigenvaluesChar`.

Example

```
gap> chi:= Irr( CharacterTable( "A5" ) )[2];
Character( CharacterTable( "A5" ),
[ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] )
gap> List( [ 1 .. 5 ], i -> Eigenvalues( chi, i ) );
[ [ 3 ], [ 2, 1 ], [ 1, 1, 1 ], [ 0, 1, 1, 0, 1 ], [ 1, 0, 0, 1, 1 ] ]
```

72.8.20 Tensored

▷ `Tensored(`*chars1*`, `*chars2*`)` (operation)

Let *chars1* and *chars2* be lists of (values lists of) class functions of the same character table. `Tensored` returns the list of tensor products of all entries in *chars1* with all entries in *chars2*.

Example

```
gap> irra5:= Irr( CharacterTable( "A5" ) );;
gap> chars1:= irra5{ [ 1 .. 3 ] };;  chars2:= irra5{ [ 2, 3 ] };;
gap> Tensored( chars1, chars2 );
[ Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
  Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
  Character( CharacterTable( "A5" ),
    [ 9, 1, 0, -2*E(5)-E(5)^2-E(5)^3-2*E(5)^4,
      -E(5)-2*E(5)^2-2*E(5)^3-E(5)^4 ] ),
  Character( CharacterTable( "A5" ), [ 9, 1, 0, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 9, 1, 0, -1, -1 ] ),
  Character( CharacterTable( "A5" ),
```

```
    [ 9, 1, 0, -E(5)-2*E(5)^2-2*E(5)^3-E(5)^4,
      -2*E(5)-E(5)^2-E(5)^3-2*E(5)^4 ] ) ]
```

72.9 Restricted and Induced Class Functions

For restricting a class function of a group G to a subgroup H and for inducing a class function of H to G, the *class fusion* from H to G must be known (see 73.3).

If F is the factor group of G by the normal subgroup N then each class function of F can be naturally regarded as a class function of G, with N in its kernel. For a class function of F, the corresponding class function of G is called the *inflated* class function. Restriction and inflation are in principle the same, namely indirection of a class function by the appropriate fusion map, and thus no extra operation is needed for this process. But note that contrary to the case of a subgroup fusion, the factor fusion can in general not be computed from the groups G and F; either one needs the natural homomorphism, or the factor fusion to the character table of F must be stored on the table of G. This explains the different syntax for computing restricted and inflated class functions.

In the following, the meaning of the optional first argument *tbl* is the same as in Section 72.8.

72.9.1 RestrictedClassFunction

▷ RestrictedClassFunction(*[tbl,]chi, target*) (operation)

Let *chi* be a class function of the group G, say, and let *target* be either a subgroup H of G or an injective homomorphism from H to G or the character table of H. Then RestrictedClassFunction returns the class function of H obtained by restricting *chi* to H.

If *chi* is a class function of a *factor group* $G$of H, where *target* is either the group H or a homomorphism from H to G or the character table of H then the restriction can be computed in the case of the homomorphism; in the other cases, this is possible only if the factor fusion from H to G is stored on the character table of H.

72.9.2 RestrictedClassFunctions

▷ RestrictedClassFunctions(*[tbl,]chars, target*) (operation)

RestrictedClassFunctions is similar to RestrictedClassFunction (72.9.1), the only difference is that it takes a list *chars* of class functions instead of one class function, and returns the list of restricted class functions.

Example

```
gap> a5:= CharacterTable( "A5" );;  s5:= CharacterTable( "S5" );;
gap> RestrictedClassFunction( Irr( s5 )[2], a5 );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
gap> RestrictedClassFunctions( Irr( s5 ), a5 );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 6, -2, 0, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ]
```

```
gap> hom:= NaturalHomomorphismByNormalSubgroup( S4, der );;
gap> RestrictedClassFunctions( Irr( Image( hom ) ), hom );
[ Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] ) ]
```

72.9.3 InducedClassFunction

▷ InducedClassFunction(*[tbl,]chi, H*) (operation)
▷ InducedClassFunction(*[tbl,]chi, hom*) (operation)
▷ InducedClassFunction(*[tbl,]chi, suptbl*) (operation)

Let *chi* be a class function of the group G, say, and let *target* be either a supergroup H of G or an injective homomorphism from H to G or the character table of H. Then InducedClassFunction returns the class function of H obtained by inducing *chi* to H.

72.9.4 InducedClassFunctions

▷ InducedClassFunctions(*[tbl,]chars, target*) (operation)

InducedClassFunctions is similar to InducedClassFunction (72.9.3), the only difference is that it takes a list *chars* of class functions instead of one class function, and returns the list of induced class functions.

Example

```
gap> InducedClassFunctions( Irr( a5 ), s5 );
[ Character( CharacterTable( "A5.2" ), [ 2, 2, 2, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, -2, 0, 1, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, -2, 0, 1, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 8, 0, 2, -2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 10, 2, -2, 0, 0, 0, 0 ] ) ]
```

72.9.5 InducedClassFunctionsByFusionMap

▷ InducedClassFunctionsByFusionMap(*subtbl, tbl, chars, fusionmap*) (function)

Let *subtbl* and *tbl* be two character tables of groups H and G, such that H is a subgroup of G, let *chars* be a list of class functions of *subtbl*, and let *fusionmap* be a fusion map from *subtbl* to *tbl*. The function returns the list of induced class functions of *tbl* that correspond to *chars*, w.r.t. the given fusion map.

InducedClassFunctionsByFusionMap is the function that does the work for InducedClassFunction (72.9.3) and InducedClassFunctions (72.9.4).

Example

```
gap> fus:= PossibleClassFusions( a5, s5 );
[ [ 1, 2, 3, 4, 4 ] ]
gap> InducedClassFunctionsByFusionMap( a5, s5, Irr( a5 ), fus[1] );
[ Character( CharacterTable( "A5.2" ), [ 2, 2, 2, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, -2, 0, 1, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, -2, 0, 1, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 8, 0, 2, -2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 10, 2, -2, 0, 0, 0, 0 ] ) ]
```

72.9.6 InducedCyclic

▷ `InducedCyclic(`*`tbl[, classes][, "all"]`*`)` (operation)

`InducedCyclic` calculates characters induced up from cyclic subgroups of the ordinary character table *`tbl`* to *`tbl`*, and returns the strictly sorted list of the induced characters.

If the string `"all"` is specified then all irreducible characters of these subgroups are induced, otherwise only the permutation characters are calculated.

If a list *`classes`* is specified then only those cyclic subgroups generated by these classes are considered, otherwise all classes of *`tbl`* are considered.

Example

```
gap> InducedCyclic( a5, "all" );
[ Character( CharacterTable( "A5" ), [ 12, 0, 0, 2, 2 ] ),
  Character( CharacterTable( "A5" ),
    [ 12, 0, 0, E(5)^2+E(5)^3, E(5)+E(5)^4 ] ),
  Character( CharacterTable( "A5" ),
    [ 12, 0, 0, E(5)+E(5)^4, E(5)^2+E(5)^3 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, -1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 30, -2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 30, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 60, 0, 0, 0, 0 ] ) ]
```

72.10 Reducing Virtual Characters

The following operations are intended for the situation that one is given a list of virtual characters of a character table and is interested in the irreducible characters of this table. The idea is to compute virtual characters of small norm from the given ones, hoping to get eventually virtual characters of norm 1.

72.10.1 ReducedClassFunctions

▷ `ReducedClassFunctions(`*`[tbl, ][constituents, ]reducibles`*`)` (operation)

Let *`reducibles`* be a list of ordinary virtual characters of the group G, say. If *`constituents`* is given then it must also be a list of ordinary virtual characters of G, otherwise we have *`constituents`* equal to *`reducibles`* in the following.

`ReducedClassFunctions` returns a record with the components `remainders` and `irreducibles`, both lists of virtual characters of G. These virtual characters are computed as follows.

Let `rems` be the set of nonzero class functions obtained by subtraction of

$$\sum_{\chi} ([\mathit{reducibles}[i], \chi]/[\chi, \chi]) \cdot \chi$$

from *`reducibles`*$[i]$, where the summation runs over *`constituents`* and $[\chi, \psi]$ denotes the scalar product of G-class functions. Let `irrs` be the list of irreducible characters in `rems`.

We project `rems` into the orthogonal space of `irrs` and all those irreducibles found this way until no new irreducibles arise. Then the `irreducibles` list is the set of all found irreducible characters, and the `remainders` list is the set of all nonzero remainders.

72.10.2 ReducedCharacters

▷ ReducedCharacters(*[tbl,]constituents, reducibles*) (operation)

ReducedCharacters is similar to ReducedClassFunctions (72.10.1), the only difference is that *constituents* and *reducibles* are assumed to be lists of characters. This means that only those scalar products must be formed where the degree of the character in *constituents* does not exceed the degree of the character in *reducibles*.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> chars:= Irr( tbl ){ [ 2 .. 4 ] };;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> red:= ReducedClassFunctions( chars );
rec(
  irreducibles :=
    [ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "A5" ),
        [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
      Character( CharacterTable( "A5" ),
        [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
      Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
      Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ],
  remainders := [  ] )
```

72.10.3 IrreducibleDifferences

▷ IrreducibleDifferences(*tbl, reducibles, reducibles2[, scprmat]*) (function)

IrreducibleDifferences returns the list of irreducible characters which occur as difference of an element of *reducibles* and an element of *reducibles2*, where these two arguments are lists of class functions of the character table *tbl*.

If *reducibles2* is the string "triangle" then the differences of elements in *reducibles* are considered.

If *scprmat* is not specified then it will be calculated, otherwise we must have *scprmat* = MatScalarProducts(*tbl*, *reducibles*, *reducibles2*) or *scprmat* = MatScalarProducts(*tbl*, *reducibles*), respectively.

Example

```
gap> IrreducibleDifferences( a5, chars, "triangle" );
[ Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
  Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ) ]
```

72.10.4 LLL

▷ LLL(*tbl, characters[, y][, "sort"][, "linearcomb"]*) (function)

LLL calls the LLL algorithm (see LLLReducedBasis (25.5.1)) in the case of lattices spanned by the virtual characters *characters* of the ordinary character table *tbl* (see ScalarProduct (72.8.5)).

By finding shorter vectors in the lattice spanned by *characters*, i.e., virtual characters of smaller norm, in some cases LLL is able to find irreducible characters.

LLL returns a record with at least components `irreducibles` (the list of found irreducible characters), `remainders` (a list of reducible virtual characters), and `norms` (the list of norms of the vectors in `remainders`). `irreducibles` together with `remainders` form a basis of the $\mathbb{Z}$-lattice spanned by *characters*.

Note that the vectors in the `remainders` list are in general *not* orthogonal (see `ReducedClassFunctions` (72.10.1)) to the irreducible characters in `irreducibles`.

Optional arguments of LLL are

y controls the sensitivity of the algorithm, see `LLLReducedBasis` (25.5.1),

"sort"
: LLL sorts *characters* and the `remainders` component of the result according to the degrees,

"linearcomb"
: the returned record contains components `irreddecomp` and `reddecomp`, which are decomposition matrices of `irreducibles` and `remainders`, with respect to *characters*.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> chars:= [ [ 8, 0, 0, -1, 0 ], [ 6, 0, 2, 0, 2 ],
>     [ 12, 0, -4, 0, 0 ], [ 6, 0, -2, 0, 0 ], [ 24, 0, 0, 0, 0 ],
>     [ 12, 0, 4, 0, 0 ], [ 6, 0, 2, 0, -2 ], [ 12, -2, 0, 0, 0 ],
>     [ 8, 0, 0, 2, 0 ], [ 12, 2, 0, 0, 0 ], [ 1, 1, 1, 1, 1 ] ];;
gap> LLL( s4, chars );
rec(
  irreducibles :=
    [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ) ],
  norms := [  ], remainders := [  ] )
```

72.10.5 Extract

▷ `Extract(`*tbl, reducibles, grammat[, missing]*`)` (function)

Let *tbl* be an ordinary character table, *reducibles* a list of characters of *tbl*, and *grammat* the matrix of scalar products of *reducibles* (see `MatScalarProducts` (72.8.6)). `Extract` tries to find irreducible characters by drawing conclusions out of the scalar products, using combinatorial and backtrack means.

The optional argument *missing* is the maximal number of irreducible characters that occur as constituents of *reducibles*. Specification of *missing* may accelerate `Extract`.

`Extract` returns a record *ext* with the components `solution` and `choice`, where the value of `solution` is a list $[M_1, \ldots, M_n]$ of decomposition matrices M_i (up to permutations of rows) with the property that $M_i^{tr} \cdot X$ is equal to the sublist at the positions *ext*`.choice[i]` of *reducibles*, for a matrix X of irreducible characters; the value of `choice` is a list of length n whose entries are lists of indices.

So the j-th column in each matrix M_i corresponds to *reducibles*$[j]$, and each row in M_i corresponds to an irreducible character. Decreased (72.10.7) can be used to examine the solution for computable irreducibles.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> red:= [ [ 5, 1, 5, 2, 1 ], [ 2, 0, 2, 2, 0 ], [ 3, -1, 3, 0, -1 ],
>           [ 6, 0, -2, 0, 0 ], [ 4, 0, 0, 1, 2 ] ];;
gap> gram:= MatScalarProducts( s4, red, red );
[ [ 6, 3, 2, 0, 2 ], [ 3, 2, 1, 0, 1 ], [ 2, 1, 2, 0, 0 ],
  [ 0, 0, 0, 2, 1 ], [ 2, 1, 0, 1, 2 ] ]
gap> ext:= Extract( s4, red, gram, 5 );
rec( choice := [ [ 2, 5, 3, 4, 1 ] ],
  solution :=
    [
      [ [ 1, 1, 0, 0, 2 ], [ 1, 0, 1, 0, 1 ], [ 0, 1, 0, 1, 0 ],
          [ 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0 ] ] ] )
gap> dec:= Decreased( s4, red, ext.solution[1], ext.choice[1] );
rec(
  irreducibles :=
    [ Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ) ],
  matrix := [  ], remainders := [  ] )
```

72.10.6 OrthogonalEmbeddingsSpecialDimension

▷ OrthogonalEmbeddingsSpecialDimension(*tbl, reducibles, grammat[, "positive"], dim*) (function)

OrthogonalEmbeddingsSpecialDimension is a variant of OrthogonalEmbeddings (25.6.1) for the situation that *tbl* is an ordinary character table, *reducibles* is a list of virtual characters of *tbl*, *grammat* is the matrix of scalar products (see MatScalarProducts (72.8.6)), and *dim* is an upper bound for the number of irreducible characters of *tbl* that occur as constituents of *reducibles*; if the vectors in *reducibles* are known to be proper characters then the string "positive" may be entered as fourth argument. (See OrthogonalEmbeddings (25.6.1) for information why this may help.)

OrthogonalEmbeddingsSpecialDimension first uses OrthogonalEmbeddings (25.6.1) to compute all orthogonal embeddings of *grammat* into a standard lattice of dimension up to *dim*, and then calls Decreased (72.10.7) in order to find irreducible characters of *tbl*.

OrthogonalEmbeddingsSpecialDimension returns a record with the following components.

`irreducibles`
: a list of found irreducibles, the intersection of all lists of irreducibles found by Decreased (72.10.7), for all possible embeddings, and

`remainders`
: a list of remaining reducible virtual characters.

Example

```
gap> s6:= CharacterTable( "S6" );;
gap> red:= InducedCyclic( s6, "all" );;
gap> Add( red, TrivialCharacter( s6 ) );
gap> lll:= LLL( s6, red );;
gap> irred:= lll.irreducibles;
[ Character( CharacterTable( "A6.2_1" ),
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A6.2_1" ),
    [ 9, 1, 0, 0, 1, -1, -3, -3, 1, 0, 0 ] ),
  Character( CharacterTable( "A6.2_1" ),
    [ 16, 0, -2, -2, 0, 1, 0, 0, 0, 0, 0 ] ) ]
gap> Set( Flat( MatScalarProducts( s6, irred, lll.remainders ) ) );
[ 0 ]
gap> dim:= NrConjugacyClasses( s6 ) - Length( lll.irreducibles );
8
gap> rem:= lll.remainders;;  Length( rem );
8
gap> gram:= MatScalarProducts( s6, rem, rem );;  RankMat( gram );
8
gap> emb1:= OrthogonalEmbeddings( gram, 8 );
rec( norms := [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  solutions := [ [ 1, 2, 3, 7, 11, 12, 13, 15 ],
      [ 1, 2, 4, 8, 10, 12, 13, 14 ], [ 1, 2, 5, 6, 9, 12, 13, 16 ] ],
  vectors :=
    [ [ -1, 0, 1, 0, 1, 0, 1, 0 ], [ 1, 0, 0, 1, 0, 1, 0, 0 ],
      [ 0, 1, 1, 0, 0, 0, 1, 1 ], [ 0, 1, 1, 0, 0, 0, 1, 0 ],
      [ 0, 1, 1, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 1, 0 ],
      [ 0, -1, 0, 0, 0, 0, 0, 1 ], [ 0, 1, 0, 0, 0, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0, 0, 1, 1 ], [ 0, 0, 1, 0, 0, 0, 0, 1 ],
      [ 0, 0, 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, -1, 1, 0, 0, 0 ],
      [ 0, 0, 0, 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 1, 1 ],
      [ 0, 0, 0, 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 1 ] ] )
```

In the following example we temporarily decrease the line length limit from its default value 80 to 62 in order to get a nicer output format.

Example

```
gap> emb2:= OrthogonalEmbeddingsSpecialDimension( s6, rem, gram, 8 );
rec(
  irreducibles :=
    [ Character( CharacterTable( "A6.2_1" ),
        [ 5, 1, -1, 2, -1, 0, 1, -3, -1, 1, 0 ] ),
      Character( CharacterTable( "A6.2_1" ),
        [ 5, 1, 2, -1, -1, 0, -3, 1, -1, 0, 1 ] ),
      Character( CharacterTable( "A6.2_1" ),
        [ 10, -2, 1, 1, 0, 0, -2, 2, 0, 1, -1 ] ),
      Character( CharacterTable( "A6.2_1" ),
        [ 10, -2, 1, 1, 0, 0, 2, -2, 0, -1, 1 ] ) ],
  remainders :=
    [ VirtualCharacter( CharacterTable( "A6.2_1" ),
        [ 0, 0, 3, -3, 0, 0, 4, -4, 0, 1, -1 ] ),
      VirtualCharacter( CharacterTable( "A6.2_1" ),
```

```
          [ 6, 2, 3, 0, 0, 1, 2, -2, 0, -1, -2 ] ),
        VirtualCharacter( CharacterTable( "A6.2_1" ),
          [ 10, 2, 1, 1, 2, 0, 2, 2, -2, -1, -1 ] ),
        VirtualCharacter( CharacterTable( "A6.2_1" ),
          [ 14, 2, 2, -1, 0, -1, 6, 2, 0, 0, -1 ] ) ] )
```

72.10.7 Decreased

▷ `Decreased(tbl, chars, decompmat[, choice])` (function)

Let *tbl* be an ordinary character table, *chars* a list of virtual characters of *tbl*, and *decompmat* a decomposition matrix, that is, a matrix M with the property that $M^{tr} \cdot X =$ *chars* holds, where X is a list of irreducible characters of *tbl*. `Decreased` tries to compute the irreducibles in X or at least some of them.

Usually `Decreased` is applied to the output of `Extract` (72.10.5) or `OrthogonalEmbeddings` (25.6.1) or `OrthogonalEmbeddingsSpecialDimension` (72.10.6). In the case of `Extract` (72.10.5), the choice component corresponding to the decomposition matrix must be entered as argument *choice* of `Decreased`.

`Decreased` returns `fail` if it can prove that no list X of irreducible characters corresponding to the arguments exists; otherwise `Decreased` returns a record with the following components.

`irreducibles`
: the list of found irreducible characters,

`remainders`
: the remaining reducible characters, and

`matrix`
: the decomposition matrix of the characters in the `remainders` component.

In the following example we temporarily decrease the line length limit from its default value 80 to 62 in order to get a nicer output format.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> x:= Irr( s4 );;
gap> red:= [ x[1]+x[2], -x[1]-x[3], -x[1]+x[3], -x[2]-x[4] ];;
gap> mat:= MatScalarProducts( s4, red, red );
[ [ 2, -1, -1, -1 ], [ -1, 2, 0, 0 ], [ -1, 0, 2, 0 ],
  [ -1, 0, 0, 2 ] ]
gap> emb:= OrthogonalEmbeddings( mat );
rec( norms := [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  solutions := [ [ 1, 6, 7, 12 ], [ 2, 5, 8, 11 ], [ 3, 4, 9, 10 ] ],
  vectors := [ [ -1, 1, 1, 0 ], [ -1, 1, 0, 1 ], [ 1, -1, 0, 0 ],
      [ -1, 0, 1, 1 ], [ -1, 0, 1, 0 ], [ -1, 0, 0, 1 ],
      [ 0, -1, 1, 0 ], [ 0, -1, 0, 1 ], [ 0, 1, 0, 0 ],
      [ 0, 0, -1, 1 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ] )
gap> dec:= Decreased( s4, red, emb.vectors{ emb.solutions[1] } );
rec(
  irreducibles :=
    [ Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
```

```
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ] ) ],
  matrix := [  ], remainders := [  ] )
gap> Decreased( s4, red, emb.vectors{ emb.solutions[2] } );
fail
gap> Decreased( s4, red, emb.vectors{ emb.solutions[3] } );
fail
```

72.10.8 DnLattice

▷ DnLattice(*tbl*, *grammat*, *reducibles*) (function)

Let *tbl* be an ordinary character table, and *reducibles* a list of virtual characters of *tbl*.

DnLattice searches for sublattices isomorphic to root lattices of type D_n, for $n \geq 4$, in the lattice that is generated by *reducibles*; each vector in *reducibles* must have norm 2, and the matrix of scalar products (see MatScalarProducts (72.8.6)) of *reducibles* must be entered as argument *grammat*.

DnLattice is able to find irreducible characters if there is a lattice of type D_n with $n > 4$. In the case $n = 4$, DnLattice may fail to determine irreducibles.

DnLattice returns a record with components

irreducibles
: the list of found irreducible characters,

remainders
: the list of remaining reducible virtual characters, and

gram
: the Gram matrix of the vectors in remainders.

The remainders list is transformed in such a way that the gram matrix is a block diagonal matrix that exhibits the structure of the lattice generated by the vectors in remainders. So DnLattice might be useful even if it fails to find irreducible characters.

In the following example we temporarily decrease the line length limit from its default value 80 to 62 in order to get a nicer output format.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> red:= [ [ 2, 0, 2, 2, 0 ], [ 4, 0, 0, 1, 2 ],
>           [ 5, -1, 1, -1, 1 ], [ -1, 1, 3, -1, -1 ] ];;
gap> gram:= MatScalarProducts( s4, red, red );
[ [ 2, 1, 0, 0 ], [ 1, 2, 1, -1 ], [ 0, 1, 2, 0 ], [ 0, -1, 0, 2 ] ]
gap> dn:= DnLattice( s4, gram, red );
rec( gram := [  ],
  irreducibles :=
    [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ) ],
  remainders := [  ] )
```

72.10.9 DnLatticeIterative

▷ DnLatticeIterative(*tbl, reducibles*) (function)

Let *tbl* be an ordinary character table, and *reducibles* either a list of virtual characters of *tbl* or a record with components remainders and norms, for example a record returned by LLL (72.10.4).

DnLatticeIterative was designed for iterative use of DnLattice (72.10.8). DnLatticeIterative selects the vectors of norm 2 among the given virtual character, calls DnLattice (72.10.8) for them, reduces the virtual characters with found irreducibles, calls DnLattice (72.10.8) again for the remaining virtual characters, and so on, until no new irreducibles are found.

DnLatticeIterative returns a record with the same components and meaning of components as LLL (72.10.4).

In the following example we temporarily decrease the line length limit from its default value 80 to 62 in order to get a nicer output format.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> red:= [ [ 2, 0, 2, 2, 0 ], [ 4, 0, 0, 1, 2 ],
>           [ 5, -1, 1, -1, 1 ], [ -1, 1, 3, -1, -1 ] ];;
gap> dn:= DnLatticeIterative( s4, red );
rec(
  irreducibles :=
    [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ) ],
  norms := [  ], remainders := [  ] )
```

72.11 Symmetrizations of Class Functions

72.11.1 Symmetrizations

▷ Symmetrizations(*[tbl,]characters, n*) (operation)

Symmetrizations returns the list of symmetrizations of the characters *characters* of the ordinary character table *tbl* with the ordinary irreducible characters of the symmetric group of degree *n*; instead of the integer *n*, the character table of the symmetric group can be entered.

The symmetrization $\chi^{[\lambda]}$ of the character χ of *tbl* with the character λ of the symmetric group S_n of degree n is defined by

$$\chi^{[\lambda]}(g) = \left(\sum_{\rho \in S_n} \lambda(\rho) \prod_{k=1}^{n} \chi(g^k)^{a_k(\rho)} \right) / n!,$$

where $a_k(\rho)$ is the number of cycles of length k in ρ.

For special kinds of symmetrizations, see SymmetricParts (72.11.2), AntiSymmetricParts (72.11.3), MinusCharacter (73.6.5) and OrthogonalComponents (72.11.4), SymplecticComponents (72.11.5).

Note that the returned list may contain zero class functions, and duplicates are not deleted.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> Symmetrizations( Irr( tbl ){ [ 1 .. 3 ] }, 3 );
[ VirtualCharacter( CharacterTable( "A5" ), [ 0, 0, 0, 0, 0 ] ),
  VirtualCharacter( CharacterTable( "A5" ), [ 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ),
    [ 8, 0, -1, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ),
    [ 8, 0, -1, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ) ]
```

72.11.2 SymmetricParts

▷ `SymmetricParts(`*tbl, characters, n*`)` (function)

is the list of symmetrizations of the characters *characters* of the character table *tbl* with the trivial character of the symmetric group of degree *n* (see `Symmetrizations` (72.11.1)).

Example

```
gap> SymmetricParts( tbl, Irr( tbl ), 3 );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 35, 3, 2, 0, 0 ] ) ]
```

72.11.3 AntiSymmetricParts

▷ `AntiSymmetricParts(`*tbl, characters, n*`)` (function)

is the list of symmetrizations of the characters *characters* of the character table *tbl* with the alternating character of the symmetric group of degree *n* (see `Symmetrizations` (72.11.1)).

Example

```
gap> AntiSymmetricParts( tbl, Irr( tbl ), 3 );
[ VirtualCharacter( CharacterTable( "A5" ), [ 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ) ]
```

72.11.4 OrthogonalComponents

▷ `OrthogonalComponents(`*tbl, chars, m*`)` (function)

If χ is a nonlinear character with indicator $+1$, a splitting of the tensor power χ^m is given by the so-called Murnaghan functions (see [Mur58]). These components in general have fewer irreducible

constituents than the symmetrizations with the symmetric group of degree *m* (see Symmetrizations (72.11.1)).

OrthogonalComponents returns the Murnaghan components of the nonlinear characters of the character table *tbl* in the list *chars* up to the power *m*, where *m* is an integer between 2 and 6.

The Murnaghan functions are implemented as in [Fra82].

Note: If *chars* is a list of character objects (see IsCharacter (72.8.1)) then also the result consists of class function objects. It is not checked whether all characters in *chars* do really have indicator $+1$; if there are characters with indicator 0 or -1, the result might contain virtual characters (see also SymplecticComponents (72.11.5)), therefore the entries of the result do in general not know that they are characters.

Example

```
gap> tbl:= CharacterTable( "A8" );;  chi:= Irr( tbl )[2];
Character( CharacterTable( "A8" ), [ 7, -1, 3, 4, 1, -1, 1, 2, 0, -1,
  0, 0, -1, -1 ] )
gap> OrthogonalComponents( tbl, [ chi ], 3 );
[ ClassFunction( CharacterTable( "A8" ),
    [ 21, -3, 1, 6, 0, 1, -1, 1, -2, 0, 0, 0, 1, 1 ] ),
  ClassFunction( CharacterTable( "A8" ),
    [ 27, 3, 7, 9, 0, -1, 1, 2, 1, 0, -1, -1, -1, -1 ] ),
  ClassFunction( CharacterTable( "A8" ),
    [ 105, 1, 5, 15, -3, 1, -1, 0, -1, 1, 0, 0, 0, 0 ] ),
  ClassFunction( CharacterTable( "A8" ),
    [ 35, 3, -5, 5, 2, -1, -1, 0, 1, 0, 0, 0, 0, 0 ] ),
  ClassFunction( CharacterTable( "A8" ),
    [ 77, -3, 13, 17, 2, 1, 1, 2, 1, 0, 0, 0, 2, 2 ] ) ]
```

72.11.5 SymplecticComponents

▷ SymplecticComponents(*tbl, chars, m*) (function)

If χ is a (nonlinear) character with indicator -1, a splitting of the tensor power χ^m is given in terms of the so-called Murnaghan functions (see [Mur58]). These components in general have fewer irreducible constituents than the symmetrizations with the symmetric group of degree *m* (see Symmetrizations (72.11.1)).

SymplecticComponents returns the symplectic symmetrizations of the nonlinear characters of the character table *tbl* in the list *chars* up to the power *m*, where *m* is an integer between 2 and 5.

Note: If *chars* is a list of character objects (see IsCharacter (72.8.1)) then also the result consists of class function objects. It is not checked whether all characters in *chars* do really have indicator -1; if there are characters with indicator 0 or $+1$, the result might contain virtual characters (see also OrthogonalComponents (72.11.4)), therefore the entries of the result do in general not know that they are characters.

Example

```
gap> tbl:= CharacterTable( "U3(3)" );;  chi:= Irr( tbl )[2];
Character( CharacterTable( "U3(3)" ),
[ 6, -2, -3, 0, -2, -2, 2, 1, -1, -1, 0, 0, 1, 1 ] )
gap> SymplecticComponents( tbl, [ chi ], 3 );
[ ClassFunction( CharacterTable( "U3(3)" ),
    [ 14, -2, 5, -1, 2, 2, 2, 1, 0, 0, 0, 0, -1, -1 ] ),
  ClassFunction( CharacterTable( "U3(3)" ),
```

```
    [ 21, 5, 3, 0, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1 ] ),
  ClassFunction( CharacterTable( "U3(3)" ),
    [ 64, 0, -8, -2, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 ] ),
  ClassFunction( CharacterTable( "U3(3)" ),
    [ 14, 6, -4, 2, -2, -2, 2, 0, 0, 0, 0, 0, -2, -2 ] ),
  ClassFunction( CharacterTable( "U3(3)" ),
    [ 56, -8, 2, 2, 0, 0, 0, -2, 0, 0, 0, 0, 0, 0 ] ) ]
```

72.12 Molien Series

72.12.1 MolienSeries

▷ MolienSeries(*[tbl,]psi[, chi]*) (function)

The *Molien series* of the character ψ, relative to the character χ, is the rational function given by the series $M_{\psi,\chi}(z) = \sum_{d=0}^{\infty} [\chi, \psi^{[d]}] z^d$, where $\psi^{[d]}$ denotes the symmetrization of ψ with the trivial character of the symmetric group S_d (see SymmetricParts (72.11.2)).

MolienSeries returns the Molien series of *psi*, relative to *chi*, where *psi* and *chi* must be characters of the same character table; this table must be entered as *tbl* if *chi* and *psi* are only lists of character values. The default for *chi* is the trivial character of *tbl*.

The return value of MolienSeries stores a value for the attribute MolienSeriesInfo (72.12.2). This admits the computation of coefficients of the series with ValueMolienSeries (72.12.3). Furthermore, this attribute gives access to numerator and denominator of the Molien series viewed as rational function, where the denominator is a product of polynomials of the form $(1-z^r)^k$; the Molien series is also displayed in this form. Note that such a representation is not unique, one can use MolienSeriesWithGivenDenominator (72.12.4) to obtain the series with a prescribed denominator.

For more information about Molien series, see [NPP84].

Example

```
gap> t:= CharacterTable( AlternatingGroup( 5 ) );;
gap> psi:= First( Irr( t ), x -> Degree( x ) = 3 );;
gap> mol:= MolienSeries( psi );
( 1-z^2-z^3+z^6+z^7-z^9 ) / ( (1-z^5)*(1-z^3)*(1-z^2)^2 )
```

72.12.2 MolienSeriesInfo

▷ MolienSeriesInfo(*ratfun*) (attribute)

If the rational function *ratfun* was constructed by MolienSeries (72.12.1), a representation as quotient of polynomials is known such that the denominator is a product of terms of the form $(1-z^r)^k$. This information is encoded as value of MolienSeriesInfo. Additionally, there is a special PrintObj (6.3.5) method for Molien series based on this.

MolienSeriesInfo returns a record that describes the rational function *ratfun* as a Molien series. The components of this record are

numer
: numerator of *ratfun* (in general a multiple of the numerator one gets by NumeratorOfRationalFunction (66.4.2)),

`denom`
: denominator of *ratfun* (in general a multiple of the denominator one gets by `NumeratorOfRationalFunction` (66.4.2)),

`ratfun`
: the rational function *ratfun* itself,

`numerstring`
: string corresponding to the polynomial `numer`, expressed in terms of `z`,

`denomstring`
: string corresponding to the polynomial `denom`, expressed in terms of `z`,

`denominfo`
: a list of the form $[[r_1,k_1],\ldots,[r_n,k_n]]$ such that `denom` is $\prod_{i=1}^{n}(1-z^{r_i})^{k_i}$.

`summands`
: a list of records, each with the components `numer`, `r`, and `k`, describing the summand `numer`$/(1-z^r)^k$,

`size`
: the order of the underlying matrix group,

`degree`
: the degree of the underlying matrix representation.

Example

```
gap> HasMolienSeriesInfo( mol );
true
gap> MolienSeriesInfo( mol );
rec( degree := 3,
  denom := x_1^12-2*x_1^10-x_1^9+x_1^8+x_1^7+x_1^5+x_1^4-x_1^3-2*x_1^2\
+1, denominfo := [ 5, 1, 3, 1, 2, 2 ],
  denomstring := "(1-z^5)*(1-z^3)*(1-z^2)^2",
  numer := -x_1^9+x_1^7+x_1^6-x_1^3-x_1^2+1,
  numerstring := "1-z^2-z^3+z^6+z^7-z^9",
  ratfun := ( 1-z^2-z^3+z^6+z^7-z^9 ) / ( (1-z^5)*(1-z^3)*(1-z^2)^2 ),
  size := 60,
  summands := [ rec( k := 1, numer := [ -24, -12, -24 ], r := 5 ),
      rec( k := 1, numer := [ -20 ], r := 3 ),
      rec( k := 2, numer := [ -45/4, 75/4, -15/4, -15/4 ], r := 2 ),
      rec( k := 3, numer := [ -1 ], r := 1 ),
      rec( k := 1, numer := [ -15/4 ], r := 1 ) ] )
```

72.12.3 ValueMolienSeries

▷ `ValueMolienSeries(`*molser, i*`)` (function)

is the *i*-th coefficient of the Molien series *series* computed by `MolienSeries` (72.12.1).

Example

```
gap> List( [ 0 .. 20 ], i -> ValueMolienSeries( mol, i ) );
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1, 6, 1, 7 ]
```

72.12.4 MolienSeriesWithGivenDenominator

▷ `MolienSeriesWithGivenDenominator(`*`molser, list`*`)` (function)

is a Molien series equal to *molser* as rational function, but viewed as quotient with denominator $\prod_{i=1}^{n}(1-z^{r_i})$, where *list* $= [r_1, r_2, \ldots, r_n]$. If *molser* cannot be represented this way, `fail` is returned.

Example

```
gap> MolienSeriesWithGivenDenominator( mol, [ 2, 6, 10 ] );
( 1+z^15 ) / ( (1-z^10)*(1-z^6)*(1-z^2) )
```

72.13 Possible Permutation Characters

For groups H and G with $H \leq G$, the induced character $(1_G)^H$ is called the *permutation character* of the operation of G on the right cosets of H. If only the character table of G is available and not the group G itself, one can try to get information about possible subgroups of G by inspection of those G-class functions that might be permutation characters, using that such a class function π must have at least the following properties. (For details, see [Isa76, Theorem 5.18.]),

(a) π is a character of G,

(b) $\pi(g)$ is a nonnegative integer for all $g \in G$,

(c) $\pi(1)$ divides $|G|$,

(d) $\pi(g^n) \geq \pi(g)$ for $g \in G$ and integers n,

(e) $[\pi, 1_G] = 1$,

(f) the multiplicity of any rational irreducible G-character ψ as a constituent of π is at most $\psi(1)/[\psi,\psi]$,

(g) $\pi(g) = 0$ if the order of g does not divide $|G|/\pi(1)$,

(h) $\pi(1)|N_G(g)|$ divides $\pi(g)|G|$ for all $g \in G$,

(i) $\pi(g) \leq (|G| - \pi(1))/(|g^G||Gal_G(g)|)$ for all nonidentity $g \in G$, where $|Gal_G(g)|$ denotes the number of conjugacy classes of G that contain generators of the group $\langle g \rangle$,

(j) if p is a prime that divides $|G|/\pi(1)$ only once then $s/(p-1)$ divides $|G|/\pi(1)$ and is congruent to 1 modulo p, where s is the number of elements of order p in the (hypothetical) subgroup H for which $\pi = (1_H)^G$ holds. (Note that $s/(p-1)$ equals the number of Sylow p subgroups in H.)

Any G-class function with these properties is called a *possible permutation character* in GAP.

(Condition (d) is checked only for those power maps that are stored in the character table of G; clearly (d) holds for all integers if it holds for all prime divisors of the group order $|G|$.)

GAP provides some algorithms to compute possible permutation characters (see `PermChars` (72.14.1)), and also provides functions to check a few more criteria whether a given character can be a transitive permutation character (see `TestPerm1` (72.14.2)).

Some information about the subgroup U can be computed from the permutation character $(1_U)^G$ using `PermCharInfo` (72.13.1).

72.13.1 PermCharInfo

▷ `PermCharInfo(tbl, permchars[, format])` (function)

Let *tbl* be the ordinary character table of the group G, and *permchars* either the permutation character $(1_U)^G$, for a subgroup U of G, or a list of such permutation characters. `PermCharInfo` returns a record with the following components.

`contained`**:**
a list containing, for each character $\psi = (1_U)^G$ in *permchars*, a list containing at position i the number $\psi[i]|U|/$ `SizesCentralizers( tbl )`$[i]$, which equals the number of those elements of U that are contained in class i of *tbl*,

`bound`**:**
a list containing, for each character $\psi = (1_U)^G$ in *permchars*, a list containing at position i the number $|U|/\gcd(|U|,$ `SizesCentralizers( tbl )`$[i])$, which divides the class length in U of an element in class i of *tbl*,

`display`**:**
a record that can be used as second argument of `Display` (6.3.6) to display each permutation character in *permchars* and the corresponding components `contained` and `bound`, for those classes where at least one character of *permchars* is nonzero,

`ATLAS`**:**
a list of strings describing the decomposition of the permutation characters in *permchars* into the irreducible characters of *tbl*, given in an Atlas-like notation. This means that the irreducible constituents are indicated by their degrees followed by lower case letters `a`, `b`, `c`, ..., which indicate the successive irreducible characters of *tbl* of that degree, in the order in which they appear in `Irr( tbl )`. A sequence of small letters (not necessarily distinct) after a single number indicates a sum of irreducible constituents all of the same degree, an exponent *n* for the letter *lett* means that *lett* is repeated *n* times. The default notation for exponentiation is *lett*`^{`*n*`}`, this is also chosen if the optional third argument *format* is the string `"LaTeX"`; if the third argument is the string `"HTML"` then exponentiation is denoted by *lett*`<sup>`*n*`</sup>`.

Example

```
gap> t:= CharacterTable( "A6" );;
gap> psi:= Sum( Irr( t ){ [ 1, 3, 6 ] } );
Character( CharacterTable( "A6" ), [ 15, 3, 0, 3, 1, 0, 0 ] )
gap> info:= PermCharInfo( t, psi );
rec( ATLAS := [ "1a+5b+9a" ], bound := [ [ 1, 3, 8, 8, 6, 24, 24 ] ],
  contained := [ [ 1, 9, 0, 8, 6, 0, 0 ] ],
  display :=
    rec(
      chars := [ [ 15, 3, 0, 3, 1, 0, 0 ], [ 1, 9, 0, 8, 6, 0, 0 ],
          [ 1, 3, 8, 8, 6, 24, 24 ] ], classes := [ 1, 2, 4, 5 ],
      letter := "I" ) )
gap> Display( t, info.display );
A6

     2  3  3  .  2
     3  2  .  2  .
     5  1  .  .  .
```

```
       1a 2a 3b 4a
    2P 1a 1a 3b 2a
    3P 1a 2a 1a 4a
    5P 1a 2a 3b 4a

I.1    15  3  3  1
I.2     1  9  8  6
I.3     1  3  8  6
gap> j1:= CharacterTable( "J1" );;
gap> psi:= TrivialCharacter( CharacterTable( "7:6" ) )^j1;
Character( CharacterTable( "J1" ), [ 4180, 20, 10, 0, 0, 2, 1, 0, 0,
  0, 0, 0, 0, 0, 0 ] )
gap> PermCharInfo( j1, psi ).ATLAS;
[ "1a+56aabb+76aaab+77aabbcc+120aaabbbccc+133a^{4}bbcc+209a^{5}" ]
```

72.13.2 PermCharInfoRelative

▷ PermCharInfoRelative(*tbl, tbl2, permchars*) (function)

Let *tbl* and *tbl2* be the ordinary character tables of two groups H and G, respectively, where H is of index two in G, and *permchars* either the permutation character $(1_U)^G$, for a subgroup U of G, or a list of such permutation characters. PermCharInfoRelative returns a record with the same components as PermCharInfo (72.13.1), the only exception is that the entries of the ATLAS component are names relative to *tbl*.

More precisely, the i-th entry of the ATLAS component is a string describing the decomposition of the i-th entry in *permchars*. The degrees and distinguishing letters of the constituents refer to the irreducibles of *tbl*, as follows. The two irreducible characters of *tbl2* of degree N, say, that extend the irreducible character N a of *tbl* are denoted by N a$^+$ and Na$^-$. The irreducible character of *tbl2* of degree $2N$, say, whose restriction to *tbl* is the sum of the irreducible characters N a and N b is denoted as N ab. Multiplicities larger than 1 of constituents are denoted by exponents.

(This format is useful mainly for multiplicity free permutation characters.)

Example

```
gap> t:= CharacterTable( "A5" );;
gap> t2:= CharacterTable( "A5.2" );;
gap> List( Irr( t2 ), x -> x[1] );
[ 1, 1, 6, 4, 4, 5, 5 ]
gap> List( Irr( t ), x -> x[1] );
[ 1, 3, 3, 4, 5 ]
gap> permchars:= List( [ [1], [1,2], [1,7], [1,3,4,4,6,6,7] ],
>                      l -> Sum( Irr( t2 ){ l } ) );
[ Character( CharacterTable( "A5.2" ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5.2" ), [ 2, 2, 2, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, 2, 0, 1, 0, 2, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 30, 2, 0, 0, 6, 0, 0 ] ) ]
gap> info:= PermCharInfoRelative( t, t2, permchars );;
gap> info.ATLAS;
[ "1a^+", "1a^{\\pm}", "1a^++5a^-",
  "1a^++3ab+4(a^+)^{2}+5a^+a^{\\pm}" ]
```

72.14 Computing Possible Permutation Characters

72.14.1 PermChars

▷ PermChars(*tbl[, cond]*) (function)

GAP provides several algorithms to determine possible permutation characters from a given character table. They are described in detail in [BP98]. The algorithm is selected from the choice of the optional argument *cond*. The user is encouraged to try different approaches, especially if one choice fails to come to an end.

Regardless of the algorithm used in a specific case, PermChars returns a list of *all* possible permutation characters with the properties described by *cond*. There is no guarantee that a character of this list is in fact a permutation character. But an empty list always means there is no permutation character with these properties (e.g., of a certain degree).

Called with only one argument, a character table *tbl*, PermChars returns the list of all possible permutation characters of the group with this character table. This list might be rather long for big groups, and its computation might take much time. The algorithm is described in [BP98, Section 3.2]; it depends on a preprocessing step, where the inequalities arising from the condition $\pi(g) \geq 0$ are transformed into a system of inequalities that guides the search (see Inequalities (72.14.5)). So the following commands compute the list of 39 possible permutation characters of the Mathieu group M_{11}.

Example

```
gap> m11:= CharacterTable( "M11" );;
gap> SetName( m11, "m11" );
gap> perms:= PermChars( m11 );;
gap> Length( perms );
39
```

There are two different search strategies for this algorithm. The default strategy simply constructs all characters with nonnegative values and then tests for each such character whether its degree is a divisor of the order of the group. The other strategy uses the inequalities to predict whether a character of a certain degree can lie in the currently searched part of the search tree. To choose this strategy, enter a record as the second argument of PermChars, and set its component degree to the range of degrees (which might also be a range containing all divisors of the group order) you want to look for; additionally, the record component ineq can take the inequalities computed by Inequalities (72.14.5) if they are needed more than once.

If a positive integer is given as the second argument *cond*, PermChars returns the list of all possible permutation characters of *tbl* that have degree *cond*. For that purpose, a preprocessing step is performed where essentially the rational character table is inverted in order to determine boundary points for the simplex in which the possible permutation characters of the given degree must lie (see PermBounds (72.14.3)). The algorithm is described at the end of [BP98, Section 3.2]. Note that inverting big integer matrices needs a lot of time and space. So this preprocessing is restricted to groups with less than 100 classes, say.

Example

```
gap> deg220:= PermChars( m11, 220 );
[ Character( m11, [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ) ]
```

If a record is given as the second argument *cond*, PermChars returns the list of all possible permutation characters that have the properties described by the components of this record. One such situation has been mentioned above. If *cond* contains a degree as value of the record component degree then PermChars will behave exactly as if this degree was entered as *cond*.

Example

```
gap> deg220 = PermChars( m11, rec( degree:= 220 ) );
true
```

For the meaning of additional components of *cond* besides degree, see PermComb (72.14.4).

Instead of degree, *cond* may have the component torso bound to a list that contains some known values of the required characters at the right positions; at least the degree *cond*.torso[1] must be an integer. In this case, the algorithm described in [BP98, Section 3.3] is chosen. The component chars, if present, holds a list of all those *rational* irreducible characters of *tbl* that might be constituents of the required characters.

(*Note*: If *cond*.chars is bound and does not contain *all* rational irreducible characters of *tbl*, GAP checks whether the scalar products of all class functions in the result list with the omitted rational irreducible characters of *tbl* are nonnegative; so there should be nontrivial reasons for excluding a character that is known to be not a constituent of the desired possible permutation characters.)

Example

```
gap> PermChars( m11, rec( torso:= [ 220 ] ) );
[ Character( m11, [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ] ) ]
gap> PermChars( m11, rec( torso:= [ 220,,,,, 2 ] ) );
[ Character( m11, [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ) ]
```

An additional restriction on the possible permutation characters computed can be forced if *con* contains, in addition to torso, the components normalsubgroup and nonfaithful, with values a list of class positions of a normal subgroup N of the group G of *tbl* and a possible permutation character π of G, respectively, such that N is contained in the kernel of π. In this case, PermChars returns the list of those possible permutation characters ψ of *tbl* coinciding with torso wherever its values are bound and having the property that no irreducible constituent of $\psi - \pi$ has N in its kernel. If the component chars is bound in *cond* then the above statements apply. An interpretation of the computed characters is the following. Suppose there exists a subgroup V of G such that $\pi = (1_V)^G$; Then $N \leq V$, and if a computed character is of the form $(1_U)^G$, for a subgroup U of G, then $V = UN$.

Example

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> nsg:= ClassPositionsOfDerivedSubgroup( s4 );;
gap> pi:= TrivialCharacter( s4 );;
gap> PermChars( s4, rec( torso:= [ 12 ], normalsubgroup:= nsg,
>                        nonfaithful:= pi ) );
[ Character( CharacterTable( "Sym(4)" ), [ 12, 2, 0, 0, 0 ] ) ]
gap> pi:= Sum( Filtered( Irr( s4 ),
>              chi -> IsSubset( ClassPositionsOfKernel( chi ), nsg ) ) );
Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, 2, 0 ] )
gap> PermChars( s4, rec( torso:= [ 12 ], normalsubgroup:= nsg,
>                        nonfaithful:= pi ) );
[ Character( CharacterTable( "Sym(4)" ), [ 12, 0, 4, 0, 0 ] ) ]
```

The class functions returned by PermChars have the properties tested by TestPerm1 (72.14.2), TestPerm2 (72.14.2), and TestPerm3 (72.14.2). So they are possible permutation characters. See TestPerm1 (72.14.2) for criteria whether a possible permutation character can in fact be a permutation character.

72.14.2 TestPerm1, ..., TestPerm5

▷ TestPerm1(*tbl*, *char*) (function)
▷ TestPerm2(*tbl*, *char*) (function)
▷ TestPerm3(*tbl*, *chars*) (function)
▷ TestPerm4(*tbl*, *chars*) (function)
▷ TestPerm5(*tbl*, *chars*, *modtbl*) (function)

The first three of these functions implement tests of the properties of possible permutation characters listed in Section 72.13, The other two implement test of additional properties. Let *tbl* be the ordinary character table of a group G, say, *char* a rational character of *tbl*, and *chars* a list of rational characters of *tbl*. For applying TestPerm5, the knowledge of a p-modular Brauer table *modtbl* of G is required. TestPerm4 and TestPerm5 expect the characters in *chars* to satisfy the conditions checked by TestPerm1 and TestPerm2 (see below).

The return values of the functions were chosen parallel to the tests listed in [NPP84].

TestPerm1 return 1 or 2 if *char* fails because of (T1) or (T2), respectively; this corresponds to the criteria (b) and (d). Note that only those power maps are considered that are stored on *tbl*. If *char* satisfies the conditions, 0 is returned.

TestPerm2 returns 1 if *char* fails because of the criterion (c), it returns 3, 4, or 5 if *char* fails because of (T3), (T4), or (T5), respectively; these tests correspond to (g), a weaker form of (h), and (j). If *char* satisfies the conditions, 0 is returned.

TestPerm3 returns the list of all those class functions in the list *chars* that satisfy criterion (h); this is a stronger version of (T6).

TestPerm4 returns the list of all those class functions in the list *chars* that satisfy (T8) and (T9) for each prime divisor p of the order of G; these tests use modular representation theory but do not require the knowledge of decomposition matrices (cf. TestPerm5 below).

(T8) implements the test of the fact that in the case that p divides $|G|$ and the degree of a transitive permutation character π exactly once, the projective cover of the trivial character is a summand of π. (This test is omitted if the projective cover cannot be identified.)

Given a permutation character π of a group G and a prime integer p, the restriction π_B to a p-block B of G has the following property, which is checked by (T9). For each $g \in G$ such that g^n is a p-element of G, $\pi_B(g^n)$ is a nonnegative integer that satisfies $|\pi_B(g)| \leq \pi_B(g^n) \leq \pi(g^n)$. (This is [Sco73, Corollary A on p. 113].)

TestPerm5 requires the p-modular Brauer table *modtbl* of G, for some prime p dividing the order of G, and checks whether those characters in the list *chars* whose degree is divisible by the p-part of the order of G can be decomposed into projective indecomposable characters; TestPerm5 returns the sublist of all those characters in *chars* that either satisfy this condition or to which the test does not apply.

Example

```
gap> tbl:= CharacterTable( "A5" );;
gap> rat:= RationalizedMat( Irr( tbl ) );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
```

```
  Character( CharacterTable( "A5" ), [ 6, -2, 0, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ]
gap> tup:= Filtered( Tuples( [ 0, 1 ], 4 ), x -> not IsZero( x ) );
[ [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ], [ 0, 0, 1, 1 ], [ 0, 1, 0, 0 ],
  [ 0, 1, 0, 1 ], [ 0, 1, 1, 0 ], [ 0, 1, 1, 1 ], [ 1, 0, 0, 0 ],
  [ 1, 0, 0, 1 ], [ 1, 0, 1, 0 ], [ 1, 0, 1, 1 ], [ 1, 1, 0, 0 ],
  [ 1, 1, 0, 1 ], [ 1, 1, 1, 0 ], [ 1, 1, 1, 1 ] ]
gap> lincomb:= List( tup, coeff -> coeff * rat );;
gap> List( lincomb, psi -> TestPerm1( tbl, psi ) );
[ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0 ]
gap> List( lincomb, psi -> TestPerm2( tbl, psi ) );
[ 0, 5, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1 ]
gap> Set( List( TestPerm3(tbl, lincomb), x -> Position(lincomb, x) ) );
[ 1, 4, 6, 7, 8, 9, 10, 11, 13 ]
gap> tbl:= CharacterTable( "A7" );
CharacterTable( "A7" )
gap> perms:= PermChars( tbl, rec( degree:= 315 ) );
[ Character( CharacterTable( "A7" ), [ 315, 3, 0, 0, 3, 0, 0, 0, 0 ] )
    , Character( CharacterTable( "A7" ),
    [ 315, 15, 0, 0, 1, 0, 0, 0, 0 ] ) ]
gap> TestPerm4( tbl, perms );
[ Character( CharacterTable( "A7" ), [ 315, 15, 0, 0, 1, 0, 0, 0, 0
     ] ) ]
gap> perms:= PermChars( tbl, rec( degree:= 15 ) );
[ Character( CharacterTable( "A7" ), [ 15, 3, 0, 3, 1, 0, 0, 1, 1 ] ),
  Character( CharacterTable( "A7" ), [ 15, 3, 3, 0, 1, 0, 3, 1, 1 ] )
 ]
gap> TestPerm5( tbl, perms, tbl mod 5 );
[ Character( CharacterTable( "A7" ), [ 15, 3, 0, 3, 1, 0, 0, 1, 1 ] )
 ]
```

72.14.3 PermBounds

▷ `PermBounds(`*tbl, d*`)` (function)

Let *tbl* be the ordinary character table of the group G. All G-characters π satisfying $\pi(g) > 0$ and $\pi(1) = d$, for a given degree *d*, lie in a simplex described by these conditions. `PermBounds` computes the boundary points of this simplex for $d = 0$, from which the boundary points for any other *d* are easily derived. (Some conditions from the power maps of *tbl* are also involved.) For this purpose, a matrix similar to the rational character table of G has to be inverted. These boundary points are used by `PermChars` (72.14.1) to construct all possible permutation characters (see 72.13) of a given degree. `PermChars` (72.14.1) either calls `PermBounds` or takes this information from the `bounds` component of its argument record.

72.14.4 PermComb

▷ `PermComb(`*tbl, arec*`)` (function)

PermComb computes possible permutation characters of the character table *tbl* by the improved combinatorial approach described at the end of [BP98, Section 3.2].

For computing the possible linear combinations *without* prescribing better bounds (i.e., when the computation of bounds shall be suppressed), enter

arec:= rec(degree := *degree*, bounds := false),

where *degree* is the character degree; this is useful if the multiplicities are expected to be small, and if this is forced by high irreducible degrees.

A list of upper bounds on the multiplicities of the rational irreducibles characters can be explicitly prescribed as a maxmult component in *arec*.

72.14.5 Inequalities

▷ Inequalities(*tbl, chars[, option]*) (operation)

Let *tbl* be the ordinary character table of a group G. The condition $\pi(g) \geq 0$ for every possible permutation character π of G places restrictions on the multiplicities a_i of the irreducible constituents χ_i of $\pi = \sum_{i=1}^{r} a_i\chi_i$. For every element $g \in G$, we have $\sum_{i=1}^{r} a_i\chi_i(g) \geq 0$. The power maps provide even stronger conditions.

This system of inequalities is kind of diagonalized, resulting in a system of inequalities restricting a_i in terms of a_j, $j < i$. These inequalities are used to construct characters with nonnegative values (see PermChars (72.14.1)). PermChars (72.14.1) either calls Inequalities or takes this information from the ineq component of its argument record.

The number of inequalities arising in the process of diagonalization may grow very strongly.

There are two ways to organize the projection. The first, which is chosen if no *option* argument is present, is the straight approach which takes the rational irreducible characters in their original order and by this guarantees the character with the smallest degree to be considered first. The other way, which is chosen if the string "small" is entered as third argument *option*, tries to keep the number of intermediate inequalities small by eventually changing the order of characters.

Example

```
gap> tbl:= CharacterTable( "M11" );;
gap> PermComb( tbl, rec( degree:= 110 ) );
[ Character( CharacterTable( "M11" ),
    [ 110, 6, 2, 2, 0, 0, 2, 2, 0, 0 ] ),
  Character( CharacterTable( "M11" ),
    [ 110, 6, 2, 6, 0, 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "M11" ), [ 110, 14, 2, 2, 0, 2, 0, 0, 0,
      0 ] ) ]
gap> # Now compute only multiplicity free permutation characters.
gap> bounds:= List( RationalizedMat( Irr( tbl ) ), x -> 1 );;
gap> PermComb( tbl, rec( degree:= 110, maxmult:= bounds ) );
[ Character( CharacterTable( "M11" ),
    [ 110, 6, 2, 2, 0, 0, 2, 2, 0, 0 ] ) ]
```

72.15 Operations for Brauer Characters

72.15.1 FrobeniusCharacterValue

▷ `FrobeniusCharacterValue(value, p)` (function)

Let *value* be a cyclotomic whose coefficients over the rationals are in the ring $\mathbb{Z}_p$ of p-local numbers, where p is a prime integer. Assume that *value* lies in $\mathbb{Z}_p[\zeta]$ for $\zeta = \exp(p^n - 1)$, for some positive integer n.

`FrobeniusCharacterValue` returns the image of *value* under the ring homomorphism from $\mathbb{Z}_p[\zeta]$ to the field with p^n elements that is defined with the help of Conway polynomials (see `ConwayPolynomial` (59.5.1)), more information can be found in [JLPW95, Sections 2-5].

If *value* is a Brauer character value in characteristic p then the result can be described as the corresponding value of the Frobenius character, that is, as the trace of a representing matrix with the given Brauer character value.

If the result of `FrobeniusCharacterValue` cannot be expressed as an element of a finite field in GAP (see Chapter 59) then `FrobeniusCharacterValue` returns `fail`.

If the Conway polynomial of degree n is required for the computation then it is computed only if `IsCheapConwayPolynomial` (59.5.2) returns `true` when it is called with p and n, otherwise `fail` is returned.

72.15.2 BrauerCharacterValue

▷ `BrauerCharacterValue(mat)` (attribute)

For an invertible matrix *mat* over a finite field F, `BrauerCharacterValue` returns the Brauer character value of *mat* if the order of *mat* is coprime to the characteristic of F, and `fail` otherwise.

The *Brauer character value* of a matrix is the sum of complex lifts of its eigenvalues.

Example

```
gap> g:= SL(2,4);;           # 2-dim. irreducible representation of A5
gap> ccl:= ConjugacyClasses( g );;
gap> rep:= List( ccl, Representative );;
gap> List( rep, Order );
[ 1, 2, 5, 5, 3 ]
gap> phi:= List( rep, BrauerCharacterValue );
[ 2, fail, E(5)^2+E(5)^3, E(5)+E(5)^4, -1 ]
gap> List( phi{ [ 1, 3, 4, 5 ] }, x -> FrobeniusCharacterValue( x, 2 ) );
[ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2)^0 ]
gap> List( rep{ [ 1, 3, 4, 5 ] }, TraceMat );
[ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2)^0 ]
```

72.15.3 SizeOfFieldOfDefinition

▷ `SizeOfFieldOfDefinition(val, p)` (function)

For a cyclotomic or a list of cyclotomics *val*, and a prime integer p, `SizeOfFieldOfDefinition` returns the size of the smallest finite field in characteristic p that contains the p-modular reduction of *val*.

The reduction map is defined as in [JLPW95], that is, the complex $(p^d - 1)$-th root of unity $\exp(p^d - 1)$ is mapped to the residue class of the indeterminate, modulo the ideal spanned by the Conway polynomial (see `ConwayPolynomial` (59.5.1)) of degree d over the field with p elements.

If *val* is a Brauer character then the value returned is the size of the smallest finite field in characteristic p over which the corresponding representation lives.

72.15.4 RealizableBrauerCharacters

▷ `RealizableBrauerCharacters(`*matrix*`, `*q*`)` (function)

For a list *matrix* of absolutely irreducible Brauer characters in characteristic p, and a power q of p, `RealizableBrauerCharacters` returns a duplicate-free list of sums of Frobenius conjugates of the rows of *matrix*, each irreducible over the field with q elements.

Example

```
gap> irr:= Irr( CharacterTable( "A5" ) mod 2 );
[ Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] ),
  Character( BrauerTable( "A5", 2 ),
    [ 2, -1, E(5)+E(5)^4, E(5)^2+E(5)^3 ] ),
  Character( BrauerTable( "A5", 2 ),
    [ 2, -1, E(5)^2+E(5)^3, E(5)+E(5)^4 ] ),
  Character( BrauerTable( "A5", 2 ), [ 4, 1, -1, -1 ] ) ]
gap> List( irr, phi -> SizeOfFieldOfDefinition( phi, 2 ) );
[ 2, 4, 4, 2 ]
gap> RealizableBrauerCharacters( irr, 2 );
[ Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] ),
  ClassFunction( BrauerTable( "A5", 2 ), [ 4, -2, -1, -1 ] ),
  Character( BrauerTable( "A5", 2 ), [ 4, 1, -1, -1 ] ) ]
```

72.16 Domains Generated by Class Functions

GAP supports groups, vector spaces, and algebras generated by class functions.

Chapter 73

Maps Concerning Character Tables

Besides the characters, *power maps* are an important part of a character table, see Section 73.1. Often their computation is not easy, and if the table has no access to the underlying group then in general they cannot be obtained from the matrix of irreducible characters; so it is useful to store them on the table.

If not only a single table is considered but different tables of a group and a subgroup or of a group and a factor group are used, also *class fusion maps* (see Section 73.3) must be known to get information about the embedding or simply to induce or restrict characters, see Section 72.9).

These are examples of functions from conjugacy classes which will be called *maps* in the following. (This should not be confused with the term mapping, cf. Chapter 32.) In GAP, maps are represented by lists. Also each character, each list of element orders, of centralizer orders, or of class lengths are maps, and the list returned by `ListPerm` (42.5.1), when this function is called with a permutation of classes, is a map.

When maps are constructed without access to a group, often one only knows that the image of a given class is contained in a set of possible images, e. g., that the image of a class under a subgroup fusion is in the set of all classes with the same element order. Using further information, such as centralizer orders, power maps and the restriction of characters, the sets of possible images can be restricted further. In many cases, at the end the images are uniquely determined.

Because of this approach, many functions in this chapter work not only with maps but with *parametrized maps* (or *paramaps* for short). More about parametrized maps can be found in Section 73.5.

The implementation follows [Bre91], a description of the main ideas together with several examples can be found in [Bre99].

Several examples in this chapter require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

Example

```
gap> LoadPackage( "ctbllib" );
true
```

73.1 Power Maps

The n-th power map of a character table is represented by a list that stores at position i the position of the class containing the n-th powers of the elements in the i-th class. The n-th power map can be

composed from the power maps of the prime divisors of n, so usually only power maps for primes are actually stored in the character table.

For an ordinary character table *tbl* with access to its underlying group G, the p-th power map of *tbl* can be computed using the identification of the conjugacy classes of G with the classes of *tbl*. For an ordinary character table without access to a group, in general the p-th power maps (and hence also the element orders) for prime divisors p of the group order are not uniquely determined by the matrix of irreducible characters. So only necessary conditions can be checked in this case, which in general yields only a list of several possibilities for the desired power map. Character tables of the GAP character table library store all p-th power maps for prime divisors p of the group order.

Power maps of Brauer tables can be derived from the power maps of the underlying ordinary tables.

For (computing and) accessing the n-th power map of a character table, PowerMap (73.1.1) can be used; if the n-th power map cannot be uniquely determined then PowerMap (73.1.1) returns fail.

The list of all possible p-th power maps of a table in the sense that certain necessary conditions are satisfied can be computed with PossiblePowerMaps (73.1.2). This provides a default strategy, the subroutines are listed in Section 73.6.

73.1.1 PowerMap

▷ PowerMap(*tbl, n[, class]*) (operation)
▷ PowerMapOp(*tbl, n[, class]*) (operation)
▷ ComputedPowerMaps(*tbl*) (attribute)

Called with first argument a character table *tbl* and second argument an integer *n*, PowerMap returns the *n*-th power map of *tbl*. This is a list containing at position i the position of the class of *n*-th powers of the elements in the i-th class of *tbl*.

If the additional third argument *class* is present then the position of *n*-th powers of the *class*-th class is returned.

If the *n*-th power map is not uniquely determined by *tbl* then fail is returned. This can happen only if *tbl* has no access to its underlying group.

The power maps of *tbl* that were computed already by PowerMap are stored in *tbl* as value of the attribute ComputedPowerMaps, the n-th power map at position n. PowerMap checks whether the desired power map is already stored, computes it using the operation PowerMapOp if it is not yet known, and stores it. So methods for the computation of power maps can be installed for the operation PowerMapOp.

Example

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> ComputedPowerMaps( tbl );
[ , [ 1, 1, 3, 2, 5, 6 ], [ 1, 2, 1, 4, 6, 5 ],,,,
  [ 1, 2, 3, 4, 1, 1 ] ]
gap> PowerMap( tbl, 5 );
[ 1, 2, 3, 4, 6, 5 ]
gap> ComputedPowerMaps( tbl );
[ , [ 1, 1, 3, 2, 5, 6 ], [ 1, 2, 1, 4, 6, 5 ],, [ 1, 2, 3, 4, 6, 5 ],
  , [ 1, 2, 3, 4, 1, 1 ] ]
gap> PowerMap( tbl, 137, 2 );
2
```

73.1.2 PossiblePowerMaps

▷ `PossiblePowerMaps(`*tbl, p[, options]*`)` (operation)

For the ordinary character table *tbl* of the group G, say, and a prime integer *p*, `PossiblePowerMaps` returns the list of all maps that have the following properties of the p-th power map of *tbl*. (Representative orders are used only if the `OrdersClassRepresentatives` (71.9.1) value of *tbl* is known.

1. For class i, the centralizer order of the image is a multiple of the i-th centralizer order; if the elements in the i-th class have order coprime to p then the centralizer orders of class i and its image are equal.

2. Let n be the order of elements in class i. If *prime* divides n then the images have order n/p; otherwise the images have order n. These criteria are checked in `InitPowerMap` (73.6.1).

3. For each character χ of G and each element g in G, the values $\chi(g^p)$ and $\mathtt{GaloisCyc}(\chi(g),p)$ are algebraic integers that are congruent modulo p; if p does not divide the element order of g then the two values are equal. This congruence is checked for the characters specified below in the discussion of the *options* argument; For linear characters λ among these characters, the condition $\chi(g)^p = \chi(g^p)$ is checked. The corresponding function is `Congruences` (73.6.2).

4. For each character χ of G, the kernel is a normal subgroup N, and $g^p \in N$ for all $g \in N$; moreover, if N has index p in G then $g^p \in N$ for all $g \in G$, and if the index of N in G is coprime to p then $g^p \notin N$ for each $g \notin N$. These conditions are checked for the kernels of all characters χ specified below, the corresponding function is `ConsiderKernels` (73.6.3).

5. If p is larger than the order m of an element $g \in G$ then the class of g^p is determined by the power maps for primes dividing the residue of p modulo m. If these power maps are stored in the `ComputedPowerMaps` (73.1.1) value of *tbl* then this information is used. This criterion is checked in `ConsiderSmallerPowerMaps` (73.6.4).

6. For each character χ of G, the symmetrization ψ defined by $\psi(g) = (\chi(g)^p - \chi(g^p))/p$ is a character. This condition is checked for the kernels of all characters χ specified below, the corresponding function is `PowerMapsAllowedBySymmetrizations` (73.6.6).

If *tbl* is a Brauer table, the possibilities are computed from those for the underlying ordinary table.

The optional argument *options*, if given, must be a record that may have the following components:

`chars`**:**
: a list of characters which are used for the check of the criteria 3., 4., and 6.; the default is `Irr(` *tbl* `)`,

`powermap`**:**
: a parametrized map which is an approximation of the desired map

`decompose`**:**
: a Boolean; a `true` value indicates that all constituents of the symmetrizations of `chars` computed for criterion 6. lie in `chars`, so the symmetrizations can be decomposed into elements

of `chars`; the default value of `decompose` is `true` if `chars` is not bound and `Irr( tbl )` is known, otherwise `false`,

`quick`**:**
: a Boolean; if `true` then the subroutines are called with value `true` for the argument *quick*; especially, as soon as only one candidate remains this candidate is returned immediately; the default value is `false`,

`parameters`**:**
: a record with components `maxamb`, `minamb` and `maxlen` which control the subroutine `PowerMapsAllowedBySymmetrizations` (73.6.6); it only uses characters with current indeterminateness up to `maxamb`, tests decomposability only for characters with current indeterminateness at least `minamb`, and admits a branch according to a character only if there is one with at most `maxlen` possible symmetrizations.

Example

```
gap> tbl:= CharacterTable( "U4(3).4" );;
gap> PossiblePowerMaps( tbl, 2 );
[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4,
      5, 6, 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18,
      18, 20, 20, 20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29 ] ]
```

73.1.3 ElementOrdersPowerMap

▷ `ElementOrdersPowerMap(`*powermap*`)` (function)

Let *powermap* be a nonempty list containing at position p, if bound, the p-th power map of a character table or group. `ElementOrdersPowerMap` returns a list of the same length as each entry in *powermap*, with entry at position i equal to the order of elements in class i if this order is uniquely determined by *powermap*, and equal to an unknown (see Chapter 74) otherwise.

Example

```
gap> tbl:= CharacterTable( "U4(3).4" );;
gap> known:= ComputedPowerMaps( tbl );;
gap> Length( known );
7
gap> sub:= ShallowCopy( known );;  Unbind( sub[7] );
gap> ElementOrdersPowerMap( sub );
[ 1, 2, 3, 3, 3, 4, 4, 5, 6, 6, Unknown(1), Unknown(2), 8, 9, 12, 2,
  2, 4, 4, 6, 6, 6, 8, 10, 12, 12, 12, Unknown(3), Unknown(4), 4, 4,
  4, 4, 4, 4, 8, 8, 8, 8, 12, 12, 12, 12, 12, 12, 20, 20, 24, 24,
  Unknown(5), Unknown(6), Unknown(7), Unknown(8) ]
gap> ord:= ElementOrdersPowerMap( known );
[ 1, 2, 3, 3, 3, 4, 4, 5, 6, 6, 7, 7, 8, 9, 12, 2, 2, 4, 4, 6, 6, 6,
  8, 10, 12, 12, 12, 14, 14, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 12, 12,
  12, 12, 12, 12, 20, 20, 24, 24, 28, 28, 28, 28 ]
gap> ord = OrdersClassRepresentatives( tbl );
true
```

73.1.4 PowerMapByComposition

▷ PowerMapByComposition(*tbl*, *n*) (function)

tbl must be a nearly character table, and *n* a positive integer. If the power maps for all prime divisors of *n* are stored in the ComputedPowerMaps (73.1.1) list of *tbl* then PowerMapByComposition returns the *n*-th power map of *tbl*. Otherwise fail is returned.

Example

```
gap> tbl:= CharacterTable( "U4(3).4" );;  exp:= Exponent( tbl );
2520
gap> PowerMapByComposition( tbl, exp );
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1 ]
gap> Length( ComputedPowerMaps( tbl ) );
7
gap> PowerMapByComposition( tbl, 11 );
fail
gap> PowerMap( tbl, 11 );;
gap> PowerMapByComposition( tbl, 11 );
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 26, 25, 27, 28, 29, 31, 30, 33, 32, 35, 34, 37,
  36, 39, 38, 41, 40, 43, 42, 45, 44, 47, 46, 49, 48, 51, 50, 53, 52 ]
```

73.2 Orbits on Sets of Possible Power Maps

The permutation group of matrix automorphisms (see MatrixAutomorphisms (71.22.1)) acts on the possible power maps returned by PossiblePowerMaps (73.1.2) by permuting a list via Permuted (21.20.18) and then mapping the images via OnPoints (41.2.1). Note that by definition, the group of *table* automorphisms acts trivially.

73.2.1 OrbitPowerMaps

▷ OrbitPowerMaps(*map*, *permgrp*) (function)

returns the orbit of the power map *map* under the action of the permutation group *permgrp* via a combination of Permuted (21.20.18) and OnPoints (41.2.1).

73.2.2 RepresentativesPowerMaps

▷ RepresentativesPowerMaps(*listofmaps*, *permgrp*) (function)

returns a list of orbit representatives of the power maps in the list *listofmaps* under the action of the permutation group *permgrp* via a combination of Permuted (21.20.18) and OnPoints (41.2.1).

Example

```
gap> tbl:= CharacterTable( "3.McL" );;
gap> grp:= MatrixAutomorphisms( Irr( tbl ) );  Size( grp );
<permutation group with 5 generators>
32
```

```
gap> poss:= PossiblePowerMaps( CharacterTable( "3.McL" ), 3 );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ],
  [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> reps:= RepresentativesPowerMaps( poss, grp );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> orb:= OrbitPowerMaps( reps[1], grp );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ],
  [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> Parametrized( orb );
[ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
  4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, [ 8, 9 ],
  [ 8, 9 ], 37, 37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52,
  52, 49, 49, 49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ]
```

73.3 Class Fusions between Character Tables

For a group G and a subgroup H of G, the fusion map between the character table of H and the character table of G is represented by a list that stores at position i the position of the i-th class of the table of H in the classes list of the table of G.

For ordinary character tables *tbl1* and *tbl2* of H and G, with access to the groups H and G, the class fusion between *tbl1* and *tbl2* can be computed using the identifications of the conjugacy classes of H with the classes of *tbl1* and the conjugacy classes of G with the classes of *tbl2*. For two ordinary character tables without access to an underlying group, or in the situation that the group stored in *tbl1* is not physically a subgroup of the group stored in *tbl2* but an isomorphic copy, in general the class fusion is not uniquely determined by the information stored on the tables such as irreducible characters and power maps. So only necessary conditions can be checked in this case, which in general yields only a list of several possibilities for the desired class fusion. Character tables of the GAP character table library store various class fusions that are regarded as important, for example fusions from maximal subgroups (see `ComputedClassFusions` (73.3.2) and `Maxes` (**CTblLib: Maxes**) in the manual for the GAP Character Table Library).

Class fusions between Brauer tables can be derived from the class fusions between the underlying ordinary tables. The class fusion from a Brauer table to the underlying ordinary table is stored when the Brauer table is constructed from the ordinary table, so no method is needed to compute such a fusion.

For (computing and) accessing the class fusion between two character tables, FusionConjugacyClasses (73.3.1) can be used; if the class fusion cannot be uniquely determined then FusionConjugacyClasses (73.3.1) returns fail.

The list of all possible class fusion between two tables in the sense that certain necessary conditions are satisfied can be computed with PossibleClassFusions (73.3.6). This provides a default strategy, the subroutines are listed in Section 73.7.

It should be noted that all the following functions except FusionConjugacyClasses (73.3.1) deal only with the situation of class fusions from subgroups. The computation of *factor fusions* from a character table to the table of a factor group is not dealt with here. Since the ordinary character table of a group G determines the character tables of all factor groups of G, the factor fusion to a given character table of a factor group of G is determined up to table automorphisms (see AutomorphismsOfTable (71.9.4)) once the class positions of the kernel of the natural epimorphism have been fixed.

73.3.1 FusionConjugacyClasses

▷ FusionConjugacyClasses(*tbl1, tbl2*) (operation)
▷ FusionConjugacyClasses(*H, G*) (operation)
▷ FusionConjugacyClasses(*hom[, tbl1, tbl2]*) (operation)
▷ FusionConjugacyClassesOp(*tbl1, tbl2*) (operation)
▷ FusionConjugacyClassesOp(*hom*) (attribute)

Called with two character tables *tbl1* and *tbl2*, FusionConjugacyClasses returns the fusion of conjugacy classes between *tbl1* and *tbl2*. (If one of the tables is a Brauer table, it will delegate this task to the underlying ordinary table.)

Called with two groups *H* and *G* where *H* is a subgroup of *G*, FusionConjugacyClasses returns the fusion of conjugacy classes between *H* and *G*. This is done by delegating to the ordinary character tables of *H* and *G*, since class fusions are stored only for character tables and not for groups.

Note that the returned class fusion refers to the ordering of conjugacy classes in the character tables if the arguments are character tables and to the ordering of conjugacy classes in the groups if the arguments are groups (see ConjugacyClasses (71.6.2)).

Called with a group homomorphism *hom*, FusionConjugacyClasses returns the fusion of conjugacy classes between the preimage and the image of *hom*; contrary to the two cases above, also factor fusions can be handled by this variant. If *hom* is the only argument then the class fusion refers to the ordering of conjugacy classes in the groups. If the character tables of preimage and image are given as *tbl1* and *tbl2*, respectively (each table with its group stored), then the fusion refers to the ordering of classes in these tables.

If no class fusion exists or if the class fusion is not uniquely determined, fail is returned; this may happen when FusionConjugacyClasses is called with two character tables that do not know compatible underlying groups.

Methods for the computation of class fusions can be installed for the operation FusionConjugacyClassesOp.

Example

```
gap> s4:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> tbls4:= CharacterTable( s4 );;
gap> d8:= SylowSubgroup( s4, 2 );
Group([ (1,2), (3,4), (1,3)(2,4) ])
gap> FusionConjugacyClasses( d8, s4 );
```

```
[ 1, 2, 3, 3, 5 ]
gap> tbls5:= CharacterTable( "S5" );;
gap> FusionConjugacyClasses( CharacterTable( "A5" ), tbls5 );
[ 1, 2, 3, 4, 4 ]
gap> FusionConjugacyClasses(CharacterTable("A5"), CharacterTable("J1"));
fail
gap> PossibleClassFusions(CharacterTable("A5"), CharacterTable("J1"));
[ [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 5, 4 ] ]
```

73.3.2 ComputedClassFusions

▷ ComputedClassFusions(*tbl*) (attribute)

The class fusions from the character table *tbl* that have been computed already by FusionConjugacyClasses (73.3.1) or explicitly stored by StoreFusion (73.3.4) are stored in the ComputedClassFusions list of *tbl1*. Each entry of this list is a record with the following components.

name
: the Identifier (71.9.8) value of the character table to which the fusion maps,

map
: the list of positions of image classes,

text **(optional)**
: a string giving additional information about the fusion map, for example whether the map is uniquely determined by the character tables,

specification **(optional, rarely used)**
: a value that distinguishes different fusions between the same tables.

Note that stored fusion maps may differ from the maps returned by GetFusionMap (73.3.3) and the maps entered by StoreFusion (73.3.4) if the table *destination* has a nonidentity ClassPermutation (71.21.5) value. So if one fetches a fusion map from a table *tbl1* to a table *tbl2* via access to the data in the ComputedClassFusions list of *tbl1* then the stored value must be composed with the ClassPermutation (71.21.5) value of *tbl2* in order to obtain the correct class fusion. (If one handles fusions only via GetFusionMap (73.3.3) and StoreFusion (73.3.4) then this adjustment is made automatically.)

Fusions are identified via the Identifier (71.9.8) value of the destination table and not by this table itself because many fusions between character tables in the GAP character table library are stored on library tables, and it is not desirable to load together with a library table also all those character tables that occur as destinations of fusions from this table.

For storing fusions and accessing stored fusions, see also GetFusionMap (73.3.3), StoreFusion (73.3.4). For accessing the identifiers of tables that store a fusion into a given character table, see NamesOfFusionSources (73.3.5).

73.3.3 GetFusionMap

▷ GetFusionMap(*source, destination[, specification]*) (function)

For two ordinary character tables *source* and *destination*, GetFusionMap checks whether the ComputedClassFusions (73.3.2) list of *source* contains a record with name component Identifier(*destination*), and returns returns the map component of the first such record. GetFusionMap(*source*, *destination*, *specification*) fetches that fusion map for which the record additionally has the specification component *specification*.

If both *source* and *destination* are Brauer tables, first the same is done, and if no fusion map was found then GetFusionMap looks whether a fusion map between the ordinary tables is stored; if so then the fusion map between *source* and *destination* is stored on *source*, and then returned.

If no appropriate fusion is found, GetFusionMap returns fail. For the computation of class fusions, see FusionConjugacyClasses (73.3.1).

73.3.4 StoreFusion

▷ StoreFusion(*source, fusion, destination*) (function)

For two character tables *source* and *destination*, StoreFusion stores the fusion *fusion* from *source* to *destination* in the ComputedClassFusions (73.3.2) list of *source*, and adds the Identifier (71.9.8) string of *destination* to the NamesOfFusionSources (73.3.5) list of *destination*.

fusion can either be a fusion map (that is, the list of positions of the image classes) or a record as described in ComputedClassFusions (73.3.2).

If fusions to *destination* are already stored on *source* then another fusion can be stored only if it has a record component specification that distinguishes it from the stored fusions. In the case of such an ambiguity, StoreFusion raises an error.

Example

```
gap> tbld8:= CharacterTable( d8 );;
gap> ComputedClassFusions( tbld8 );
[ rec( map := [ 1, 2, 3, 3, 5 ], name := "CT1" ) ]
gap> Identifier( tbls4 );
"CT1"
gap> GetFusionMap( tbld8, tbls4 );
[ 1, 2, 3, 3, 5 ]
gap> GetFusionMap( tbls4, tbls5 );
fail
gap> poss:= PossibleClassFusions( tbls4, tbls5 );
[ [ 1, 5, 2, 3, 6 ] ]
gap> StoreFusion( tbls4, poss[1], tbls5 );
gap> GetFusionMap( tbls4, tbls5 );
[ 1, 5, 2, 3, 6 ]
```

73.3.5 NamesOfFusionSources

▷ NamesOfFusionSources(*tbl*) (attribute)

For a character table *tbl*, NamesOfFusionSources returns the list of identifiers of all those character tables that are known to have fusions to *tbl* stored. The NamesOfFusionSources value is updated whenever a fusion to *tbl* is stored using StoreFusion (73.3.4).

Example

```
gap> NamesOfFusionSources( tbls4 );
[ "CT2" ]
gap> Identifier( CharacterTable( d8 ) );
"CT2"
```

73.3.6 PossibleClassFusions

▷ PossibleClassFusions(*subtbl, tbl[, options]*) (operation)

For two ordinary character tables *subtbl* and *tbl* of the groups H and G, say, PossibleClassFusions returns the list of all maps that have the following properties of class fusions from *subtbl* to *tbl*.

1. For class i, the centralizer order of the image in G is a multiple of the i-th centralizer order in H, and the element orders in the i-th class and its image are equal. These criteria are checked in InitFusion (73.7.1).

2. The class fusion commutes with power maps. This is checked using TestConsistencyMaps (73.5.12).

3. If the permutation character of G corresponding to the action of G on the cosets of H is specified (see the discussion of the *options* argument below) then it prescribes for each class C of G the number of elements of H fusing into C. The corresponding function is CheckPermChar (73.7.2).

4. The table automorphisms of *tbl* (see AutomorphismsOfTable (71.9.4)) are used in order to compute only orbit representatives. (But note that the list returned by PossibleClassFusions contains the full orbits.)

5. For each character χ of G, the restriction to H via the class fusion is a character of H. This condition is checked for all characters specified below, the corresponding function is FusionsAllowedByRestrictions (73.7.4).

6. The class multiplication coefficients in *subtbl* do not exceed the corresponding coefficients in *tbl*. This is checked in ConsiderStructureConstants (73.3.7), see also the comment on the parameter verify below.

If *subtbl* and *tbl* are Brauer tables then the possibilities are computed from those for the underlying ordinary tables.

The optional argument *options* must be a record that may have the following components:

chars
: a list of characters of *tbl* which are used for the check of 5.; the default is Irr(*tbl*),

subchars
: a list of characters of *subtbl* which are constituents of the restrictions of chars, the default is Irr(*subtbl*),

fusionmap
: a parametrized map which is an approximation of the desired map,

`decompose`
: a Boolean; a `true` value indicates that all constituents of the restrictions of `chars` computed for criterion 5. lie in `subchars`, so the restrictions can be decomposed into elements of `subchars`; the default value of `decompose` is `true` if `subchars` is not bound and `Irr( subtbl )` is known, otherwise `false`,

`permchar`
: (a values list of) a permutation character; only those fusions affording that permutation character are computed,

`quick`
: a Boolean; if `true` then the subroutines are called with value `true` for the argument *`quick`*; especially, as soon as only one possibility remains then this possibility is returned immediately; the default value is `false`,

`verify`
: a Boolean; if `false` then `ConsiderStructureConstants` (73.3.7) is called only if more than one orbit of possible class fusions exists, under the action of the groups of table automorphisms; the default value is `false` (because the computation of the structure constants is usually very time consuming, compared with checking the other criteria),

`parameters`
: a record with components `maxamb`, `minamb` and `maxlen` which control the subroutine `FusionsAllowedByRestrictions` (73.7.4); it only uses characters with current indeterminateness up to `maxamb`, tests decomposability only for characters with current indeterminateness at least `minamb`, and admits a branch according to a character only if there is one with at most `maxlen` possible restrictions.

Example

```
gap> subtbl:= CharacterTable( "U3(3)" );;  tbl:= CharacterTable( "J4" );;
gap> PossibleClassFusions( subtbl, tbl );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 5, 5, 6, 10, 13, 12, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 16, 16, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 13, 12, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 13, 12, 16, 16, 22, 22 ] ]
```

73.3.7 ConsiderStructureConstants

▷ `ConsiderStructureConstants(`*`subtbl, tbl, fusions, quick`*`)` (function)

Let *`subtbl`* and *`tbl`* be ordinary character tables and *`fusions`* be a list of possible class fusions from *`subtbl`* to *`tbl`*. `ConsiderStructureConstants` returns the list of those maps σ in *`fusions`* with the property that for all triples (i,j,k) of class positions, `ClassMultiplicationCoefficient`$(subtbl,i,j,k)$ is not bigger than `ClassMultiplicationCoefficient`$(tbl,\sigma[i],\sigma[j],\sigma[k])$; see `ClassMultiplicationCoefficient` (71.12.7) for the definition of class multiplication coefficients/structure constants.

The argument *`quick`* must be a Boolean; if it is `true` then only those triples are checked for which for which at least two entries in *`fusions`* have different images.

73.4 Orbits on Sets of Possible Class Fusions

The permutation groups of table automorphisms (see AutomorphismsOfTable (71.9.4)) of the subgroup table *subtbl* and the supergroup table *tbl* act on the possible class fusions from *subtbl* to *tbl* that are returned by PossibleClassFusions (73.3.6), the former by permuting a list via Permuted (21.20.18), the latter by mapping the images via OnPoints (41.2.1).

If a set of possible fusions with certain properties was computed that are not invariant under the full groups of table automorphisms then only a smaller group acts on this set. This may happen for example if a permutation character or if an explicit approximation of the fusion map was prescribed in the call of PossibleClassFusions (73.3.6).

73.4.1 OrbitFusions

▷ OrbitFusions(*subtblautomorphisms, fusionmap, tblautomorphisms*) (function)

returns the orbit of the class fusion map *fusionmap* under the actions of the permutation groups *subtblautomorphisms* and *tblautomorphisms* of automorphisms of the character table of the subgroup and the supergroup, respectively.

73.4.2 RepresentativesFusions

▷ RepresentativesFusions(*subtbl, listofmaps, tbl*) (function)

Let *listofmaps* be a list of class fusions from the character table *subtbl* to the character table *tbl*. RepresentativesFusions returns a list of orbit representatives of the class fusions under the action of maximal admissible subgroups of the table automorphism groups of these character tables.

Instead of the character tables *subtbl* and *tbl*, also the permutation groups of their table automorphisms (see AutomorphismsOfTable (71.9.4)) may be entered.

Example

```
gap> fus:= GetFusionMap( subtbl, tbl );
[ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ]
gap> orb:= OrbitFusions( AutomorphismsOfTable( subtbl ), fus,
>              AutomorphismsOfTable( tbl ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 5, 5, 6, 10, 13, 12, 14, 14, 21, 21 ] ]
gap> rep:= RepresentativesFusions( subtbl, orb, tbl );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ] ]
```

73.5 Parametrized Maps

A *parametrized map* is a list whose *i*-th entry is either unbound (which means that nothing is known about the image(s) of the *i*-th class) or the image of the *i*-th class (i.e., an integer for fusion maps, power maps, element orders etc., and a cyclotomic for characters), or a list of possible images of the *i*-th class. In this sense, maps are special parametrized maps. We often identify a parametrized map *paramap* with the set of all maps *map* with the property that either *map*[i] = *paramap*[i] or *map*[i] is contained in the list *paramap*[i]; we say then that *map* is contained in *paramap*.

This definition implies that parametrized maps cannot be used to describe sets of maps where lists are possible images. An exception are strings which naturally arise as images when class names are

considered. So strings and lists of strings are allowed in parametrized maps, and character constants (see Chapter 27) are not allowed in maps.

73.5.1 CompositionMaps

▷ CompositionMaps(*paramap2, paramap1[, class]*) (function)

The composition of two parametrized maps *paramap1*, *paramap2* is defined as the parametrized map *comp* that contains all compositions $f_2 \circ f_1$ of elements f_1 of *paramap1* and f_2 of *paramap2*. For example, the composition of a character χ of a group G by a parametrized class fusion map from a subgroup H to G is the parametrized map that contains all restrictions of χ by elements of the parametrized fusion map.

CompositionMaps(*paramap2*, *paramap1*) is a parametrized map with entry CompositionMaps(*paramap2*, *paramap1*, *class*) at position *class*. If *paramap1*[*class*] is an integer then CompositionMaps(*paramap2*, *paramap1*, *class*) is equal to *paramap2*[*paramap1*[*class*]]. Otherwise it is the union of *paramap2*[*i*] for *i* in *paramap1*[*class*].

Example

```
gap> map1:= [ 1, [ 2 .. 4 ], [ 4, 5 ], 1 ];;
gap> map2:= [ [ 1, 2 ], 2, 2, 3, 3 ];;
gap> CompositionMaps( map2, map1 );
[ [ 1, 2 ], [ 2, 3 ], 3, [ 1, 2 ] ]
gap> CompositionMaps( map1, map2 );
[ [ 1, 2, 3, 4 ], [ 2, 3, 4 ], [ 2, 3, 4 ], [ 4, 5 ], [ 4, 5 ] ]
```

73.5.2 InverseMap

▷ InverseMap(*paramap*) (function)

For a parametrized map *paramap*, InverseMap returns a mutable parametrized map whose *i*-th entry is unbound if *i* is not in the image of *paramap*, equal to *j* if *i* is (in) the image of *paramap*[*j*] exactly for *j*, and equal to the set of all preimages of *i* under *paramap* otherwise.

We have CompositionMaps(*paramap*, InverseMap(*paramap*)) the identity map.

Example

```
gap> tbl:= CharacterTable( "2.A5" );;  f:= CharacterTable( "A5" );;
gap> fus:= GetFusionMap( tbl, f );
[ 1, 1, 2, 3, 3, 4, 4, 5, 5 ]
gap> inv:= InverseMap( fus );
[ [ 1, 2 ], 3, [ 4, 5 ], [ 6, 7 ], [ 8, 9 ] ]
gap> CompositionMaps( fus, inv );
[ 1, 2, 3, 4, 5 ]
gap> # transfer a power map ‘‘up’’ to the factor group
gap> pow:= PowerMap( tbl, 2 );
[ 1, 1, 2, 4, 4, 8, 8, 6, 6 ]
gap> CompositionMaps( fus, CompositionMaps( pow, inv ) );
[ 1, 1, 3, 5, 4 ]
gap> last = PowerMap( f, 2 );
true
gap> # transfer a power map of the factor group ‘‘down’’ to the group
gap> CompositionMaps( inv, CompositionMaps( PowerMap( f, 2 ), fus ) );
```

```
  [ [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], [ 4, 5 ], [ 4, 5 ], [ 8, 9 ],
    [ 8, 9 ], [ 6, 7 ], [ 6, 7 ] ]
```

73.5.3 ProjectionMap

▷ `ProjectionMap(`*fusionmap*`)` (function)

For a map *fusionmap*, `ProjectionMap` returns a parametrized map whose *i*-th entry is unbound if *i* is not in the image of *fusionmap*, and equal to *j* if *j* is the smallest position such that *i* is the image of *fusionmap*`[`*j*`]`.

We have `CompositionMaps(` *fusionmap* `, ProjectionMap(` *fusionmap* `) )` the identity map, i.e., first projecting and then fusing yields the identity. Note that *fusionmap* must *not* be a parametrized map.

Example

```
gap> ProjectionMap( [ 1, 1, 1, 2, 2, 2, 3, 4, 5, 5, 5, 6, 6, 6 ] );
[ 1, 4, 7, 8, 9, 12 ]
```

73.5.4 Indirected

▷ `Indirected(`*character, paramap*`)` (function)

For a map *character* and a parametrized map *paramap*, `Indirected` returns a parametrized map whose entry at position *i* is *character*`[` *paramap*`[`*i*`]` `]` if *paramap*`[`*i*`]` is an integer, and an unknown (see Chapter 74) otherwise.

Example

```
gap> tbl:= CharacterTable( "M12" );;
gap> fus:= [ 1, 3, 4, [ 6, 7 ], 8, 10, [ 11, 12 ], [ 11, 12 ],
>            [ 14, 15 ], [ 14, 15 ] ];;
gap> List( Irr( tbl ){ [ 1 .. 6 ] }, x -> Indirected( x, fus ) );
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 11, 3, 2, Unknown(9), 1, 0, Unknown(10), Unknown(11), 0, 0 ],
  [ 11, 3, 2, Unknown(12), 1, 0, Unknown(13), Unknown(14), 0, 0 ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, Unknown(15), Unknown(16) ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, Unknown(17), Unknown(18) ],
  [ 45, -3, 0, 1, 0, 0, -1, -1, 1, 1 ] ]
```

73.5.5 Parametrized

▷ `Parametrized(`*list*`)` (function)

For a list *list* of (parametrized) maps of the same length, `Parametrized` returns the smallest parametrized map containing all elements of *list*.

`Parametrized` is the inverse function to `ContainedMaps` (73.5.6).

Example

```
gap> Parametrized( [ [ 1, 2, 3, 4, 5 ], [ 1, 3, 2, 4, 5 ],
>                    [ 1, 2, 3, 4, 6 ] ] );
[ 1, [ 2, 3 ], [ 2, 3 ], 4, [ 5, 6 ] ]
```

73.5.6 ContainedMaps

▷ ContainedMaps(*paramap*) (function)

For a parametrized map *paramap*, ContainedMaps returns the set of all maps contained in *paramap*.

ContainedMaps is the inverse function to Parametrized (73.5.5) in the sense that Parametrized(ContainedMaps(*paramap*)) is equal to *paramap*.

Example

```
gap> ContainedMaps( [ 1, [ 2, 3 ], [ 2, 3 ], 4, [ 5, 6 ] ] );
[ [ 1, 2, 2, 4, 5 ], [ 1, 2, 2, 4, 6 ], [ 1, 2, 3, 4, 5 ],
  [ 1, 2, 3, 4, 6 ], [ 1, 3, 2, 4, 5 ], [ 1, 3, 2, 4, 6 ],
  [ 1, 3, 3, 4, 5 ], [ 1, 3, 3, 4, 6 ] ]
```

73.5.7 UpdateMap

▷ UpdateMap(*character, paramap, indirected*) (function)

Let *character* be a map, *paramap* a parametrized map, and *indirected* a parametrized map that is contained in CompositionMaps(*character*, *paramap*).

Then UpdateMap changes *paramap* to the parametrized map containing exactly the maps whose composition with *character* is equal to *indirected*.

If a contradiction is detected then false is returned immediately, otherwise true.

Example

```
gap> subtbl:= CharacterTable("S4(4).2");; tbl:= CharacterTable("He");;
gap> fus:= InitFusion( subtbl, tbl );;
gap> fus;
[ 1, 2, 2, [ 2, 3 ], 4, 4, [ 7, 8 ], [ 7, 8 ], 9, 9, 9, [ 10, 11 ],
  [ 10, 11 ], 18, 18, 25, 25, [ 26, 27 ], [ 26, 27 ], 2, [ 6, 7 ],
  [ 6, 7 ], [ 6, 7, 8 ], 10, 10, 17, 17, 18, [ 19, 20 ], [ 19, 20 ] ]
gap> chi:= Irr( tbl )[2];
Character( CharacterTable( "He" ), [ 51, 11, 3, 6, 0, 3, 3, -1, 1, 2,
  0, 3*E(7)+3*E(7)^2+3*E(7)^4, 3*E(7)^3+3*E(7)^5+3*E(7)^6, 2,
  E(7)+E(7)^2+2*E(7)^3+E(7)^4+2*E(7)^5+2*E(7)^6,
  2*E(7)+2*E(7)^2+E(7)^3+2*E(7)^4+E(7)^5+E(7)^6, 1, 1, 0, 0,
  -E(7)-E(7)^2-E(7)^4, -E(7)^3-E(7)^5-E(7)^6, E(7)+E(7)^2+E(7)^4,
  E(7)^3+E(7)^5+E(7)^6, 1, 0, 0, -1, -1, 0, 0, E(7)+E(7)^2+E(7)^4,
  E(7)^3+E(7)^5+E(7)^6 ] )
gap> filt:= Filtered( Irr( subtbl ), x -> x[1] = 50 );
[ Character( CharacterTable( "S4(4).2" ),
    [ 50, 10, 10, 2, 5, 5, -2, 2, 0, 0, 0, 1, 1, 0, 0, 0, 0, -1, -1,
      10, 2, 2, 2, 1, 1, 0, 0, 0, -1, -1 ] ),
  Character( CharacterTable( "S4(4).2" ),
    [ 50, 10, 10, 2, 5, 5, -2, 2, 0, 0, 0, 1, 1, 0, 0, 0, 0, -1, -1,
      -10, -2, -2, -2, -1, -1, 0, 0, 0, 1, 1 ] ) ]
gap> UpdateMap( chi, fus, filt[1] + TrivialCharacter( subtbl ) );
true
gap> fus;
[ 1, 2, 2, 3, 4, 4, 8, 7, 9, 9, 9, 10, 10, 18, 18, 25, 25,
  [ 26, 27 ], [ 26, 27 ], 2, [ 6, 7 ], [ 6, 7 ], [ 6, 7 ], 10, 10,
  17, 17, 18, [ 19, 20 ], [ 19, 20 ] ]
```

73.5.8 MeetMaps

▷ MeetMaps(*paramap1, paramap2*) (function)

For two parametrized maps *paramap1* and *paramap2*, MeetMaps changes *paramap1* such that the image of class *i* is the intersection of *paramap1*[*i*] and *paramap2*[*i*].

If this implies that no images remain for a class, the position of such a class is returned. If no such inconsistency occurs, MeetMaps returns true.

Example

```
gap> map1:= [ [ 1, 2 ], [ 3, 4 ], 5, 6, [ 7, 8, 9 ] ];;
gap> map2:= [ [ 1, 3 ], [ 3, 4 ], [ 5, 6 ], 6, [ 8, 9, 10 ] ];;
gap> MeetMaps( map1, map2 );  map1;
true
[ 1, [ 3, 4 ], 5, 6, [ 8, 9 ] ]
```

73.5.9 CommutativeDiagram

▷ CommutativeDiagram(*paramap1, paramap2, paramap3, paramap4[, improvements]*) (function)

Let *paramap1*, *paramap2*, *paramap3*, *paramap4* be parametrized maps covering parametrized maps f_1, f_2, f_3, f_4 with the property that CompositionMaps(f_2, f_1) is equal to CompositionMaps(f_4, f_3).

CommutativeDiagram checks this consistency, and changes the arguments such that all possible images are removed that cannot occur in the parametrized maps f_i.

The return value is fail if an inconsistency was found. Otherwise a record with the components imp1, imp2, imp3, imp4 is returned, each bound to the list of positions where the corresponding parametrized map was changed,

The optional argument *improvements* must be a record with components imp1, imp2, imp3, imp4. If such a record is specified then only diagrams are considered where entries of the *i*-th component occur as preimages of the *i*-th parametrized map.

When an inconsistency is detected, CommutativeDiagram immediately returns fail. Otherwise a record is returned that contains four lists imp1, ..., imp4: The *i*-th component is the list of classes where the *i*-th argument was changed.

Example

```
gap> map1:= [[ 1, 2, 3 ], [ 1, 3 ]];; map2:= [[ 1, 2 ], 1, [ 1, 3 ]];;
gap> map3:= [ [ 2, 3 ], 3 ];;  map4:= [ , 1, 2, [ 1, 2 ] ];;
gap> imp:= CommutativeDiagram( map1, map2, map3, map4 );
rec( imp1 := [ 2 ], imp2 := [ 1 ], imp3 := [  ], imp4 := [  ] )
gap> map1;  map2;  map3;  map4;
[ [ 1, 2, 3 ], 1 ]
[ 2, 1, [ 1, 3 ] ]
[ [ 2, 3 ], 3 ]
[ , 1, 2, [ 1, 2 ] ]
gap> imp2:= CommutativeDiagram( map1, map2, map3, map4, imp );
rec( imp1 := [  ], imp2 := [  ], imp3 := [  ], imp4 := [  ] )
```

73.5.10 CheckFixedPoints

▷ CheckFixedPoints(*inside1, between, inside2*) (function)

Let *inside1*, *between*, *inside2* be parametrized maps, where *between* is assumed to map each fixed point of *inside1* (that is, *inside1*[*i*] = *i*) to a fixed point of *inside2* (that is, *between*[*i*] is either an integer that is fixed by *inside2* or a list that has nonempty intersection with the union of its images under *inside2*). CheckFixedPoints changes *between* and *inside2* by removing all those entries violate this condition.

When an inconsistency is detected, CheckFixedPoints immediately returns fail. Otherwise the list of positions is returned where changes occurred.

Example

```
gap> subtbl:= CharacterTable( "L4(3).2_2" );;
gap> tbl:= CharacterTable( "O7(3)" );;
gap> fus:= InitFusion( subtbl, tbl );;  fus{ [ 48, 49 ] };
[ [ 54, 55, 56, 57 ], [ 54, 55, 56, 57 ] ]
gap> CheckFixedPoints( ComputedPowerMaps( subtbl )[5], fus,
>        ComputedPowerMaps( tbl )[5] );
[ 48, 49 ]
gap> fus{ [ 48, 49 ] };
[ [ 56, 57 ], [ 56, 57 ] ]
```

73.5.11 TransferDiagram

▷ TransferDiagram(*inside1, between, inside2[, improvements]*) (function)

Let *inside1*, *between*, *inside2* be parametrized maps covering parametrized maps m_1, f, m_2 with the property that CompositionMaps(m_2, f) is equal to CompositionMaps(f, m_1).

TransferDiagram checks this consistency, and changes the arguments such that all possible images are removed that cannot occur in the parametrized maps m_i and f.

So TransferDiagram is similar to CommutativeDiagram (73.5.9), but *between* occurs twice in each diagram checked.

If a record *improvements* with fields impinside1, impbetween, and impinside2 is specified, only those diagrams with elements of impinside1 as preimages of *inside1*, elements of impbetween as preimages of *between* or elements of impinside2 as preimages of *inside2* are considered.

When an inconsistency is detected, TransferDiagram immediately returns fail. Otherwise a record is returned that contains three lists impinside1, impbetween, and impinside2 of positions where the arguments were changed.

Example

```
gap> subtbl:= CharacterTable( "2F4(2)" );;  tbl:= CharacterTable( "Ru" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> tr:= TransferDiagram(PowerMap( subtbl, 2), fus, PowerMap(tbl, 2));
rec( impbetween := [ 12, 23 ], impinside1 := [  ], impinside2 := [  ]
```

```
 )
gap> tr:= TransferDiagram(PowerMap(subtbl, 3), fus, PowerMap( tbl, 3 ));
rec( impbetween := [ 14, 24, 25 ], impinside1 := [  ],
  impinside2 := [  ] )
gap> tr:= TransferDiagram( PowerMap(subtbl, 3), fus, PowerMap(tbl, 3),
>             tr );
rec( impbetween := [  ], impinside1 := [  ], impinside2 := [  ] )
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, [ 25, 26 ],
  [ 25, 26 ], 5, 5, 6, 8, 14, 13, 19, 19, [ 25, 26 ], [ 25, 26 ], 27,
  27 ]
```

73.5.12 TestConsistencyMaps

▷ `TestConsistencyMaps(`*powermap1, fusionmap, powermap2[, fusimp]*`)` (function)

Let *powermap1* and *powermap2* be lists of parametrized maps, and *fusionmap* a parametrized map, such that for each *i*, the *i*-th entry in *powermap1*, *fusionmap*, and the *i*-th entry in *powermap2* (if bound) are valid arguments for `TransferDiagram` (73.5.11). So a typical situation for applying `TestConsistencyMaps` is that *fusionmap* is an approximation of a class fusion, and *powermap1*, *powermap2* are the lists of power maps of the subgroup and the group.

`TestConsistencyMaps` repeatedly applies `TransferDiagram` (73.5.11) to these arguments for all *i* until no more changes occur.

If a list *fusimp* is specified then only those diagrams with elements of *fusimp* as preimages of *fusionmap* are considered.

When an inconsistency is detected, `TestConsistencyMaps` immediately returns `false`. Otherwise `true` is returned.

Example

```
gap> subtbl:= CharacterTable( "2F4(2)" );;  tbl:= CharacterTable( "Ru" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> TestConsistencyMaps( ComputedPowerMaps( subtbl ), fus,
>        ComputedPowerMaps( tbl ) );
true
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, [ 25, 26 ],
  [ 25, 26 ], 5, 5, 6, 8, 14, 13, 19, 19, [ 25, 26 ], [ 25, 26 ], 27,
  27 ]
gap> Indeterminateness( fus );
16
```

73.5.13 Indeterminateness

▷ `Indeterminateness(`*paramap*`)` (function)

For a parametrized map *paramap*, Indeterminateness returns the number of maps contained in *paramap*, that is, the product of lengths of lists in *paramap* denoting lists of several images.

Example

```
gap> Indeterminateness([ 1, [ 2, 3 ], [ 4, 5 ], [ 6, 7, 8, 9, 10 ], 11 ]);
20
```

73.5.14 PrintAmbiguity

▷ PrintAmbiguity(*list, paramap*) (function)

For each map in the list *list*, PrintAmbiguity prints its position in *list*, the indeterminateness (see Indeterminateness (73.5.13)) of the composition with the parametrized map *paramap*, and the list of positions where a list of images occurs in this composition.

Example

```
gap> paramap:= [ 1, [ 2, 3 ], [ 3, 4 ], [ 2, 3, 4 ], 5 ];;
gap> list:= [ [ 1, 1, 1, 1, 1 ], [ 1, 1, 2, 2, 3 ], [ 1, 2, 3, 4, 5 ] ];;
gap> PrintAmbiguity( list, paramap );
1 1 [  ]
2 4 [ 2, 4 ]
3 12 [ 2, 3, 4 ]
```

73.5.15 ContainedSpecialVectors

▷ ContainedSpecialVectors(*tbl, chars, paracharacter, func*) (function)
▷ IntScalarProducts(*tbl, chars, candidate*) (function)
▷ NonnegIntScalarProducts(*tbl, chars, candidate*) (function)
▷ ContainedPossibleVirtualCharacters(*tbl, chars, paracharacter*) (function)
▷ ContainedPossibleCharacters(*tbl, chars, paracharacter*) (function)

Let *tbl* be an ordinary character table, *chars* a list of class functions (or values lists), *paracharacter* a parametrized class function of *tbl*, and *func* a function that expects the three arguments *tbl*, *chars*, and a values list of a class function, and that returns either true or false.

ContainedSpecialVectors returns the list of all those elements *vec* of *paracharacter* that have integral norm, have integral scalar product with the principal character of *tbl*, and that satisfy *func*(*tbl*, *chars*, *vec*) = true.

Two special cases of *func* are the check whether the scalar products in *tbl* between the vector *vec* and all lists in *chars* are integers or nonnegative integers, respectively. These functions are accessible as global variables IntScalarProducts and NonnegIntScalarProducts, and ContainedPossibleVirtualCharacters and ContainedPossibleCharacters provide access to these special cases of ContainedSpecialVectors.

Example

```
gap> subtbl:= CharacterTable( "HSM12" );;  tbl:= CharacterTable( "HS" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> rest:= CompositionMaps( Irr( tbl )[8], fus );
[ 231, [ -9, 7 ], [ -9, 7 ], [ -9, 7 ], 6, 15, 15, [ -1, 15 ],
  [ -1, 15 ], 1, [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ -2, 0 ],
  [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], 0, 0, 1, 0, 0, 0, 0 ]
gap> irr:= Irr( subtbl );;
```

```
gap> # no further condition
gap> cont1:= ContainedSpecialVectors( subtbl, irr, rest,
>               function( tbl, chars, vec ) return true; end );;
gap> Length( cont1 );
24
gap> # require scalar products to be integral
gap> cont2:= ContainedSpecialVectors( subtbl, irr, rest,
>               IntScalarProducts );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
gap> # additionally require scalar products to be nonnegative
gap> cont3:= ContainedSpecialVectors( subtbl, irr, rest,
>               NonnegIntScalarProducts );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
gap> cont2 = ContainedPossibleVirtualCharacters( subtbl, irr, rest );
true
gap> cont3 = ContainedPossibleCharacters( subtbl, irr, rest );
true
```

73.5.16 CollapsedMat

▷ CollapsedMat(*mat, maps*) (function)

is a record with the components

`fusion`
: fusion that collapses those columns of *mat* that are equal in *mat* and also for all maps in the list *maps*,

`mat` the image of *mat* under that fusion.

Example

```
gap> mat:= [ [ 1, 1, 1, 1 ], [ 2, -1, 0, 0 ], [ 4, 4, 1, 1 ] ];;
gap> coll:= CollapsedMat( mat, [] );
rec( fusion := [ 1, 2, 3, 3 ],
  mat := [ [ 1, 1, 1 ], [ 2, -1, 0 ], [ 4, 4, 1 ] ] )
gap> List( last.mat, x -> x{ last.fusion } ) = mat;
true
gap> coll:= CollapsedMat( mat, [ [ 1, 1, 1, 2 ] ] );
rec( fusion := [ 1, 2, 3, 4 ],
  mat := [ [ 1, 1, 1, 1 ], [ 2, -1, 0, 0 ], [ 4, 4, 1, 1 ] ] )
```

73.5.17 ContainedDecomposables

▷ ContainedDecomposables(*constituents, moduls, parachar, func*) (function)
▷ ContainedCharacters(*tbl, constituents, parachar*) (function)

For these functions, let *constituents* be a list of *rational* class functions, *moduls* a list of positive integers, *parachar* a parametrized rational class function, *func* a function that returns either true or false when called with (a values list of) a class function, and *tbl* a character table.

ContainedDecomposables returns the set of all elements χ of *parachar* that satisfy *func*$(\chi) =$ true and that lie in the $\mathbb{Z}$-lattice spanned by *constituents*, modulo *moduls*. The latter means they lie in the $\mathbb{Z}$-lattice spanned by *constituents* and the set $\{$*moduls*$[i] \cdot e_i; 1 \leq i \leq n\}$ where n is the length of *parachar* and e_i is the i-th standard basis vector.

One application of ContainedDecomposables is the following. *constituents* is a list of (values lists of) rational characters of an ordinary character table *tbl*, *moduls* is the list of centralizer orders of *tbl* (see SizesCentralizers (71.9.2)), and *func* checks whether a vector in the lattice mentioned above has nonnegative integral scalar product in *tbl* with all entries of *constituents*. This situation is handled by ContainedCharacters. Note that the entries of the result list are *not* necessary linear combinations of *constituents*, and they are *not* necessarily characters of *tbl*.

Example

```
gap> subtbl:= CharacterTable( "HSM12" );;  tbl:= CharacterTable( "HS" );;
gap> rat:= RationalizedMat( Irr( subtbl ) );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> rest:= CompositionMaps( Irr( tbl )[8], fus );
[ 231, [ -9, 7 ], [ -9, 7 ], [ -9, 7 ], 6, 15, 15, [ -1, 15 ],
  [ -1, 15 ], 1, [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ -2, 0 ],
  [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], 0, 0, 1, 0, 0, 0, 0 ]
gap> # compute all vectors in the lattice
gap> ContainedDecomposables( rat, SizesCentralizers( subtbl ), rest,
>        ReturnTrue );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
gap> # compute only those vectors that are characters
gap> ContainedDecomposables( rat, SizesCentralizers( subtbl ), rest,
>        x -> NonnegIntScalarProducts( subtbl, Irr( subtbl ), x ) );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
```

73.6 Subroutines for the Construction of Power Maps

In the argument lists of the functions Congruences (73.6.2), ConsiderKernels (73.6.3), and ConsiderSmallerPowerMaps (73.6.4), *tbl* is an ordinary character table, *chars* a list of (values

lists of) characters of *tbl*, *prime* a prime integer, *approxmap* a parametrized map that is an approximation for the *prime*-th power map of *tbl* (e.g., a list returned by InitPowerMap (73.6.1), and *quick* a Boolean.

The *quick* value true means that only those classes are considered for which *approxmap* lists more than one possible image.

73.6.1 InitPowerMap

▷ InitPowerMap(*tbl, prime*) (function)

For an ordinary character table *tbl* and a prime *prime*, InitPowerMap returns a parametrized map that is a first approximation of the *prime*-th powermap of *tbl*, using the conditions 1. and 2. listed in the description of PossiblePowerMaps (73.1.2).

If there are classes for which no images are possible, according to these criteria, then fail is returned.

Example

```
gap> t:= CharacterTable( "U4(3).4" );;
gap> pow:= InitPowerMap( t, 2 );
[ 1, 1, 3, 4, 5, [ 2, 16 ], [ 2, 16, 17 ], 8, 3, [ 3, 4 ],
  [ 11, 12 ], [ 11, 12 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ], 14,
  [ 9, 20 ], 1, 1, 2, 2, 3, [ 3, 4, 5 ], [ 3, 4, 5 ],
  [ 6, 7, 18, 19, 30, 31, 32, 33 ], 8, 9, 9, [ 9, 10, 20, 21, 22 ],
  [ 11, 12 ], [ 11, 12 ], 16, 16, [ 2, 16 ], [ 2, 16 ], 17, 17,
  [ 6, 18, 30, 31, 32, 33 ], [ 6, 18, 30, 31, 32, 33 ],
  [ 6, 7, 18, 19, 30, 31, 32, 33 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ],
  20, 20, [ 9, 20 ], [ 9, 20 ], [ 9, 10, 20, 21, 22 ],
  [ 9, 10, 20, 21, 22 ], 24, 24, [ 15, 25, 26, 40, 41, 42, 43 ],
  [ 15, 25, 26, 40, 41, 42, 43 ], [ 28, 29 ], [ 28, 29 ], [ 28, 29 ],
  [ 28, 29 ] ]
```

73.6.2 Congruences (for character tables)

▷ Congruences(*tbl, chars, approxmap, prime, quick*) (function)

Congruences replaces the entries of *approxmap* by improved values, according to condition 3. listed in the description of PossiblePowerMaps (73.1.2).

For each class for which no images are possible according to the tests, the new value of *approxmap* is an empty list. Congruences returns true if no such inconsistencies occur, and false otherwise.

Example

```
gap> Congruences( t, Irr( t ), pow, 2, false );  pow;
true
[ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, [ 6, 7 ], 14, 9, 1, 1, 2, 2,
  3, 4, 5, [ 6, 7 ], 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18,
  18, [ 18, 19 ], [ 18, 19 ], 20, 20, 20, 20, 22, 22, 24, 24,
  [ 25, 26 ], [ 25, 26 ], 28, 28, 29, 29 ]
```

73.6.3 ConsiderKernels

▷ ConsiderKernels(*tbl, chars, approxmap, prime, quick*) (function)

ConsiderKernels replaces the entries of *approxmap* by improved values, according to condition 4. listed in the description of PossiblePowerMaps (73.1.2).

Congruences (73.6.2) returns true if the orders of the kernels of all characters in *chars* divide the order of the group of *tbl*, and false otherwise.

Example

```
gap> t:= CharacterTable( "A7.2" );;  init:= InitPowerMap( t, 2 );
[ 1, 1, 3, 4, [ 2, 9, 10 ], 6, 3, 8, 1, 1, [ 2, 9, 10 ], 3, [ 3, 4 ],
  6, [ 7, 12 ] ]
gap> ConsiderKernels( t, Irr( t ), init, 2, false );
true
gap> init;
[ 1, 1, 3, 4, 2, 6, 3, 8, 1, 1, 2, 3, [ 3, 4 ], 6, 7 ]
```

73.6.4 ConsiderSmallerPowerMaps

▷ ConsiderSmallerPowerMaps(*tbl, approxmap, prime, quick*) (function)

ConsiderSmallerPowerMaps replaces the entries of *approxmap* by improved values, according to condition 5. listed in the description of PossiblePowerMaps (73.1.2).

ConsiderSmallerPowerMaps returns true if each class admits at least one image after the checks, otherwise false is returned. If no element orders of *tbl* are stored (see OrdersClassRepresentatives (71.9.1)) then true is returned without any tests.

Example

```
gap> t:= CharacterTable( "3.A6" );;  init:= InitPowerMap( t, 5 );
[ 1, [ 2, 3 ], [ 2, 3 ], 4, [ 5, 6 ], [ 5, 6 ], [ 7, 8 ], [ 7, 8 ],
  9, [ 10, 11 ], [ 10, 11 ], 1, [ 2, 3 ], [ 2, 3 ], 1, [ 2, 3 ],
  [ 2, 3 ] ]
gap> Indeterminateness( init );
4096
gap> ConsiderSmallerPowerMaps( t, init, 5, false );
true
gap> Indeterminateness( init );
256
```

73.6.5 MinusCharacter

▷ MinusCharacter(*character, primepowermap, prime*) (function)

Let *character* be (the list of values of) a class function χ, *prime* a prime integer p, and *primepowermap* a parametrized map that is an approximation of the p-th power map for the character table of χ. MinusCharacter returns the parametrized map of values of χ^{p-}, which is defined by $\chi^{p-}(g) = (\chi(g)^p - \chi(g^p))/p$.

Example

```
gap> tbl:= CharacterTable( "S7" );;  pow:= InitPowerMap( tbl, 2 );;
gap> pow;
```

```
[ 1, 1, 3, 4, [ 2, 9, 10 ], 6, 3, 8, 1, 1, [ 2, 9, 10 ], 3, [ 3, 4 ],
  6, [ 7, 12 ] ]
gap> chars:= Irr( tbl ){ [ 2 .. 5 ] };;
gap> List( chars, x -> MinusCharacter( x, pow, 2 ) );
[ [ 0, 0, 0, 0, [ 0, 1 ], 0, 0, 0, 0, 0, [ 0, 1 ], 0, 0, 0, [ 0, 1 ] ]
    ,
  [ 15, -1, 3, 0, [ -2, -1, 0 ], 0, -1, 1, 5, -3, [ 0, 1, 2 ], -1, 0,
      0, [ 0, 1 ] ],
  [ 15, -1, 3, 0, [ -1, 0, 2 ], 0, -1, 1, 5, -3, [ 1, 2, 4 ], -1, 0,
      0, 1 ],
  [ 190, -2, 1, 1, [ 0, 2 ], 0, 1, 1, -10, -10, [ 0, 2 ], -1, -1, 0,
      [ -1, 0 ] ] ]
```

73.6.6 PowerMapsAllowedBySymmetrizations

▷ PowerMapsAllowedBySymmetrizations(*tbl*, *subchars*, *chars*, *approxmap*, *prime*, *parameters*) (function)

Let *tbl* be an ordinary character table, *prime* a prime integer, *approxmap* a parametrized map that is an approximation of the *prime*-th power map of *tbl* (e.g., a list returned by InitPowerMap (73.6.1), *chars* and *subchars* two lists of (values lists of) characters of *tbl*, and *parameters* a record with components maxlen, minamb, maxamb (three integers), quick (a Boolean), and contained (a function). Usual values of contained are ContainedCharacters (73.5.17) or ContainedPossibleCharacters (73.5.15).

PowerMapsAllowedBySymmetrizations replaces the entries of *approxmap* by improved values, according to condition 6. listed in the description of PossiblePowerMaps (73.1.2).

More precisely, the strategy used is as follows.

First, for each $\chi \in$ *chars*, let minus:= MinusCharacter(χ, *approxmap*, *prime*).

- If Indeterminateness(minus)$= 1$ and *parameters*.quick = false then the scalar products of minus with *subchars* are checked; if not all scalar products are nonnegative integers then an empty list is returned, otherwise χ is deleted from the list of characters to inspect.
- Otherwise if Indeterminateness(minus) is smaller than *parameters*.minamb then χ is deleted from the list of characters.
- If *parameters*.minamb $\leq$ Indeterminateness(minus) $\leq$ *parameters*.maxamb then construct the list of contained class functions poss:= *parameters*.contained(*tbl*, *subchars*, minus) and Parametrized(poss), and improve the approximation of the power map using UpdateMap (73.5.7).

If this yields no further immediate improvements then we branch. If there is a character from *chars* left with less or equal *parameters*.maxlen possible symmetrizations, compute the union of power maps allowed by these possibilities. Otherwise we choose a class C such that the possible symmetrizations of a character in *chars* differ at C, and compute recursively the union of all allowed power maps with image at C fixed in the set given by the current approximation of the power map.

Example

```
gap> tbl:= CharacterTable( "U4(3).4" );;
gap> pow:= InitPowerMap( tbl, 2 );;
gap> Congruences( tbl, Irr( tbl ), pow, 2 );;  pow;
```

```
[ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, [ 6, 7 ], 14, 9, 1, 1, 2, 2,
  3, 4, 5, [ 6, 7 ], 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18,
  18, [ 18, 19 ], [ 18, 19 ], 20, 20, 20, 20, 22, 22, 24, 24,
  [ 25, 26 ], [ 25, 26 ], 28, 28, 29, 29 ]
gap> PowerMapsAllowedBySymmetrizations( tbl, Irr( tbl ), Irr( tbl ),
>       pow, 2, rec( maxlen:= 10, contained:= ContainedPossibleCharacters,
>       minamb:= 2, maxamb:= infinity, quick:= false ) );
[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4,
      5, 6, 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18,
      18, 20, 20, 20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29 ] ]
```

73.7 Subroutines for the Construction of Class Fusions

73.7.1 InitFusion

▷ `InitFusion(`*subtbl, tbl*`)` (function)

For two ordinary character tables *subtbl* and *tbl*, `InitFusion` returns a parametrized map that is a first approximation of the class fusion from *subtbl* to *tbl*, using condition 1. listed in the description of `PossibleClassFusions` (73.3.6).

If there are classes for which no images are possible, according to this criterion, then `fail` is returned.

Example

```
gap> subtbl:= CharacterTable( "2F4(2)" );;  tbl:= CharacterTable( "Ru" );;
gap> fus:= InitFusion( subtbl, tbl );
[ 1, 2, 2, 4, [ 5, 6 ], [ 5, 6, 7, 8 ], [ 5, 6, 7, 8 ], [ 9, 10 ],
  11, 14, 14, [ 13, 14, 15 ], [ 16, 17 ], [ 18, 19 ], 20, [ 25, 26 ],
  [ 25, 26 ], [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], [ 5, 6, 7, 8 ],
  [ 13, 14, 15 ], [ 13, 14, 15 ], [ 18, 19 ], [ 18, 19 ], [ 25, 26 ],
  [ 25, 26 ], [ 27, 28, 29 ], [ 27, 28, 29 ] ]
```

73.7.2 CheckPermChar

▷ `CheckPermChar(`*subtbl, tbl, approxmap, permchar*`)` (function)

`CheckPermChar` replaces the entries of the parametrized map *approxmap* by improved values, according to condition 3. listed in the description of `PossibleClassFusions` (73.3.6).

`CheckPermChar` returns `true` if no inconsistency occurred, and `false` otherwise.

Example

```
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar ); fus;
true
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
```

73.7.3 ConsiderTableAutomorphisms

▷ ConsiderTableAutomorphisms(*approxmap, grp*) (function)

ConsiderTableAutomorphisms replaces the entries of the parametrized map *approxmap* by improved values, according to condition 4. listed in the description of PossibleClassFusions (73.3.6).

Afterwards exactly one representative of fusion maps (contained in *approxmap*) in each orbit under the action of the permutation group *grp* is contained in the modified parametrized map.

ConsiderTableAutomorphisms returns the list of positions where *approxmap* was changed.

Example

```
gap> ConsiderTableAutomorphisms( fus, AutomorphismsOfTable( tbl ) );
[ 16 ]
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  25, [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ], [ 18, 19 ],
  [ 25, 26 ], [ 25, 26 ], 27, 27 ]
```

73.7.4 FusionsAllowedByRestrictions

▷ FusionsAllowedByRestrictions(*subtbl, tbl, subchars, chars, approxmap, parameters*) (function)

Let *subtbl* and *tbl* be ordinary character tables, *subchars* and *chars* two lists of (values lists of) characters of *subtbl* and *tbl*, respectively, *approxmap* a parametrized map that is an approximation of the class fusion of *subtbl* in *tbl*, and *parameters* a record with components maxlen, minamb, maxamb (three integers), quick (a Boolean), and contained (a function). Usual values of contained are ContainedCharacters (73.5.17) or ContainedPossibleCharacters (73.5.15).

FusionsAllowedByRestrictions replaces the entries of *approxmap* by improved values, according to condition 5. listed in the description of PossibleClassFusions (73.3.6).

More precisely, the strategy used is as follows.

First, for each $\chi \in$ *chars*, let restricted:= CompositionMaps(χ, *approxmap*).

- If Indeterminateness(restricted)$= 1$ and *parameters*.quick = false then the scalar products of restricted with *subchars* are checked; if not all scalar products are non-negative integers then an empty list is returned, otherwise χ is deleted from the list of characters to inspect.
- Otherwise if Indeterminateness(minus) is smaller than *parameters*.minamb then χ is deleted from the list of characters.
- If *parameters*.minamb $\leq$ Indeterminateness(restricted) $\leq$ *parameters*.maxamb then construct poss:= *parameters*.contained(*subtbl*, *subchars*, restricted) and Parametrized(poss), and improve the approximation of the fusion map using UpdateMap (73.5.7).

If this yields no further immediate improvements then we branch. If there is a character from *chars* left with less or equal *parameters*.maxlen possible restrictions, compute the union of fusion maps allowed by these possibilities. Otherwise we choose a class C such that the possible restrictions of a character in *chars* differ at C, and compute recursively the union of all allowed fusion maps with image at C fixed in the set given by the current approximation of the fusion map.

Example

```
gap> subtbl:= CharacterTable( "U3(3)" );;  tbl:= CharacterTable( "J4" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> TestConsistencyMaps( ComputedPowerMaps( subtbl ), fus,
>        ComputedPowerMaps( tbl ) );
true
gap> fus;
[ 1, 2, 4, 4, [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], 10, [ 12, 13 ],
  [ 12, 13 ], [ 14, 15, 16 ], [ 14, 15, 16 ], [ 21, 22 ], [ 21, 22 ] ]
gap> ConsiderTableAutomorphisms( fus, AutomorphismsOfTable( tbl ) );
[ 9 ]
gap> fus;
[ 1, 2, 4, 4, [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], 10, 12, [ 12, 13 ],
  [ 14, 15, 16 ], [ 14, 15, 16 ], [ 21, 22 ], [ 21, 22 ] ]
gap> FusionsAllowedByRestrictions( subtbl, tbl, Irr( subtbl ),
>        Irr( tbl ), fus, rec( maxlen:= 10,
>        contained:= ContainedPossibleCharacters, minamb:= 2,
>        maxamb:= infinity, quick:= false ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 16, 16, 22, 22 ] ]
```

Chapter 74

Unknowns

Sometimes the result of an operation does not allow further computations with it. In many cases, then an error is signalled, and the computation is stopped.

This is not appropriate for some applications in character theory. For example, if one wants to induce a character of a group to a supergroup (see `InducedClassFunction` (72.9.3)) but the class fusion is only a parametrized map (see Chapter 73), there may be values of the induced character which are determined by the fusion map, whereas other values are not known.

For this and other situations, GAP provides the data type *unknown*. An object of this type, further on called an *unknown*, may stand for any cyclotomic (see Chapter 18), in particular its family (see 13.1) is `CyclotomicsFamily`.

Unknowns are parametrized by positive integers. When a GAP session is started, no unknowns exist.

The only ways to create unknowns are to call the function `Unknown` (74.1.1) or a function that calls it, or to do arithmetical operations with unknowns.

GAP objects containing unknowns will contain *fixed* unknowns when they are printed to files, i.e., function calls `Unknown(`*n*`)` instead of `Unknown()`. So be careful to read files printed in different GAP sessions, since there may be the same unknown at different places.

The rest of this chapter contains information about the unknown constructor, the category, and comparison of and arithmetical operations for unknowns. More is not known about unknowns in GAP.

74.1 More about Unknowns

74.1.1 Unknown

▷ `Unknown(`*[n]*`)` (operation)

Called without argument, `Unknown` returns a new unknown value, i.e., the first one that is larger than all unknowns which exist in the current GAP session.

Called with a positive integer *n*, `Unknown` returns the *n*-th unknown; if this did not exist yet, it is created.

74.1.2 LargestUnknown

▷ `LargestUnknown` (global variable)

`LargestUnknown` is the largest *n* that is used in any `Unknown( n )` in the current GAP session. This is used in `Unknown` (74.1.1) which increments this value when asked to make a new unknown.

74.1.3 IsUnknown

▷ `IsUnknown(obj)` (Category)

is the category of unknowns in GAP.

Example

```
gap> Unknown();  List( [ 1 .. 20 ], i -> Unknown() );;
Unknown(1)
gap> Unknown();   # note that we have already created 21 unknowns.
Unknown(22)
gap> Unknown(2000);  Unknown();
Unknown(2000)
Unknown(2001)
gap> LargestUnknown;
2001
gap> IsUnknown( Unknown );  IsUnknown( Unknown() );
false
true
```

74.1.4 Comparison of Unknowns

Unknowns can be *compared* via = and < with all cyclotomics and with certain other GAP objects (see 4.12). We have `Unknown( n ) >= Unknown( m )` if and only if `n >= m` holds, unknowns are larger than all cyclotomics that are not unknowns.

Example

```
gap> Unknown() >= Unknown();  Unknown(2) < Unknown(3);
false
true
gap> Unknown() > 3;  Unknown() > E(3);
true
true
gap> Unknown() > Z(8);  Unknown() > [];
false
false
```

74.1.5 Arithmetical Operations for Unknowns

The usual arithmetic operations `+`, `-`, `*` and `/` are defined for addition, subtraction, multiplication and division of unknowns and cyclotomics. The result will be a new unknown except in one of the following cases.

Multiplication with zero yields zero, and multiplication with one or addition of zero yields the old unknown. *Note* that division by an unknown causes an error, since an unknown might stand for zero.

As unknowns are cyclotomics, dense lists of unknowns and other cyclotomics are row vectors and they can be added and multiplied in the usual way. Consequently, lists of such row vectors of equal length are (ordinary) matrices (see `IsOrdinaryMatrix` (24.2.2)).

Chapter 75

Monomiality Questions

This chapter describes functions dealing with the monomiality of finite (solvable) groups and their characters.

All these functions assume *characters* to be class function objects as described in Chapter 72, lists of character *values* are not allowed.

The usual *property tests* of GAP that return either `true` or `false` are not sufficient for us. When we ask whether a group character χ has a certain property, such as quasiprimitivity, we usually want more information than just yes or no. Often we are interested in the reason *why* a group character χ was proved to have a certain property, e.g., whether monomiality of χ was proved by the observation that the underlying group is nilpotent, or whether it was necessary to construct a linear character of a subgroup from which χ can be induced. In the latter case we also may be interested in this linear character. Therefore we need test functions that return a record containing such useful information. For example, the record returned by the function `TestQuasiPrimitive` (75.3.3) contains the component `isQuasiPrimitive` (which is the known boolean property flag), and additionally the component `comment`, a string telling the reason for the value of the `isQuasiPrimitive` component, and in the case that the argument χ was *not* quasiprimitive also the component `character`, which is an irreducible constituent of a nonhomogeneous restriction of χ to a normal subgroup. Besides these test functions there are also the known properties, e.g., the property `IsQuasiPrimitive` (75.3.3) which will call the attribute `TestQuasiPrimitive` (75.3.3), and return the value of the `isQuasiPrimitive` component of the result.

A few words about how to use the monomiality functions seem to be necessary. Monomiality questions usually involve computations in many subgroups and factor groups of a given group, and for these groups often expensive calculations such as that of the character table are necessary. So one should be careful not to construct the same group over and over again, instead the same group object should be reused, such that its character table need to be computed only once. For example, suppose you want to restrict a character to a normal subgroup N that was constructed as a normal closure of some group elements, and suppose that you have already computed with normal subgroups (by calls to `NormalSubgroups` (39.19.8) or `MaximalNormalSubgroups` (39.19.9)) and their character tables. Then you should look in the lists of known normal subgroups whether N is contained, and if so you can use the known character table. A mechanism that supports this for normal subgroups is described in 71.23.

Also the following hint may be useful in this context. If you know that sooner or later you will compute the character table of a group G then it may be advisable to compute it as soon as possible. For example, if you need the normal subgroups of G then they can be computed more efficiently if

the character table of G is known, and they can be stored compatibly to the contained G-conjugacy classes. This correspondence of classes list and normal subgroup can be used very often.

Several *examples* in this chapter use the symmetric group S_4 and the special linear group $SL(2,3)$. For running the examples, you must first define the groups, for example as follows.

Example

```
gap> S4:= SymmetricGroup( 4 );;  SetName( S4, "S4" );
gap> Sl23:= SL( 2, 3 );;
```

75.1 InfoMonomial (Info Class)

75.1.1 InfoMonomial

▷ `InfoMonomial` (info class)

Most of the functions described in this chapter print some (hopefully useful) *information* if the info level of the info class `InfoMonomial` is at least 1, see 7.4 for details.

75.2 Character Degrees and Derived Length

75.2.1 Alpha

▷ `Alpha(G)` (attribute)

For a group `G`, `Alpha` returns a list whose i-th entry is the maximal derived length of groups $G/\ker(\chi)$ for $\chi \in Irr(G)$ with $\chi(1)$ at most the i-th irreducible degree of `G`.

75.2.2 Delta

▷ `Delta(G)` (attribute)

For a group `G`, `Delta` returns the list $[1, alp[2]-alp[1], \ldots, alp[n]-alp[n-1]]$, where $alp =$`Alpha( G )` (see `Alpha` (75.2.1)).

75.2.3 IsBergerCondition

▷ `IsBergerCondition(G)` (property)
▷ `IsBergerCondition(chi)` (property)

Called with an irreducible character `chi` of a group G, `IsBergerCondition` returns `true` if `chi` satisfies $M' \leq \ker(\chi)$ for every normal subgroup M of G with the property that $M \leq \ker(\psi)$ holds for all $\psi \in Irr(G)$ with $\psi(1) < \chi(1)$, and `false` otherwise.

Called with a group `G`, `IsBergerCondition` returns `true` if all irreducible characters of `G` satisfy the inequality above, and `false` otherwise.

For groups of odd order the result is always `true` by a theorem of T. R. Berger (see [Ber76, Thm. 2.2]).

In the case that `false` is returned, `InfoMonomial` (75.1.1) tells about a degree for which the inequality is violated.

Example

```
gap> Alpha( Sl23 );
[ 1, 3, 3 ]
gap> Alpha( S4 );
[ 1, 2, 3 ]
gap> Delta( Sl23 );
[ 1, 2, 0 ]
gap> Delta( S4 );
[ 1, 1, 1 ]
gap> IsBergerCondition( S4 );
true
gap> IsBergerCondition( Sl23 );
false
gap> List( Irr( Sl23 ), IsBergerCondition );
[ true, true, true, false, false, false, true ]
gap> List( Irr( Sl23 ), Degree );
[ 1, 1, 1, 2, 2, 2, 3 ]
```

75.3 Primitivity of Characters

75.3.1 TestHomogeneous

▷ TestHomogeneous(*chi, N*) (function)

For a group character *chi* of the group G, say, and a normal subgroup *N* of G, TestHomogeneous returns a record with information whether the restriction of *chi* to *N* is homogeneous, i.e., is a multiple of an irreducible character.

N may be given also as list of conjugacy class positions w.r.t. the character table of G.

The components of the result are

isHomogeneous
: true or false,

comment
: a string telling a reason for the value of the isHomogeneous component,

character
: irreducible constituent of the restriction, only bound if the restriction had to be checked,

multiplicity
: multiplicity of the character component in the restriction of *chi*.

Example

```
gap> n:= DerivedSubgroup( Sl23 );;
gap> chi:= Irr( Sl23 )[7];
Character( CharacterTable( SL(2,3) ), [ 3, 0, 0, 3, 0, 0, -1 ] )
gap> TestHomogeneous( chi, n );
rec( character := Character( CharacterTable( Group(
    [ [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ],
      [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
      [ [ Z(3)^0, Z(3) ], [ Z(3), Z(3) ] ] ]) ), [ 1, -1, 1, -1, 1 ] )
```

```
    , comment := "restriction checked", isHomogeneous := false,
  multiplicity := 1 )
gap> chi:= Irr( Sl23 )[4];
Character( CharacterTable( SL(2,3) ), [ 2, 1, 1, -2, -1, -1, 0 ] )
gap> cln:= ClassPositionsOfNormalSubgroup( CharacterTable( Sl23 ), n );
[ 1, 4, 7 ]
gap> TestHomogeneous( chi, cln );
rec( comment := "restricts irreducibly", isHomogeneous := true )
```

75.3.2 IsPrimitiveCharacter

▷ IsPrimitiveCharacter(*chi*) (property)

For a character *chi* of the group G, say, IsPrimitiveCharacter returns true if *chi* is not induced from any proper subgroup, and false otherwise.

Example

```
gap> IsPrimitive( Irr( Sl23 )[4] );
true
gap> IsPrimitive( Irr( Sl23 )[7] );
false
```

75.3.3 TestQuasiPrimitive

▷ TestQuasiPrimitive(*chi*) (attribute)
▷ IsQuasiPrimitive(*chi*) (property)

TestQuasiPrimitive returns a record with information about quasiprimitivity of the group character *chi*, i.e., whether *chi* restricts homogeneously to every normal subgroup of its group. The result record contains at least the components isQuasiPrimitive (with value either true or false) and comment (a string telling a reason for the value of the component isQuasiPrimitive). If *chi* is not quasiprimitive then there is additionally a component character, with value an irreducible constituent of a nonhomogeneous restriction of *chi*.

IsQuasiPrimitive returns true or false, depending on whether the character *chi* is quasiprimitive.

Note that for solvable groups, quasiprimitivity is the same as primitivity (see IsPrimitiveCharacter (75.3.2)).

Example

```
gap> chi:= Irr( Sl23 )[4];
Character( CharacterTable( SL(2,3) ), [ 2, 1, 1, -2, -1, -1, 0 ] )
gap> TestQuasiPrimitive( chi );
rec( comment := "all restrictions checked", isQuasiPrimitive := true )
gap> chi:= Irr( Sl23 )[7];
Character( CharacterTable( SL(2,3) ), [ 3, 0, 0, 3, 0, 0, -1 ] )
gap> TestQuasiPrimitive( chi );
rec( character := Character( CharacterTable( Group(
    [ [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ],
      [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
      [ [ Z(3)^0, Z(3) ], [ Z(3), Z(3) ] ] ]) ), [ 1, -1, 1, -1, 1 ] )
    , comment := "restriction checked", isQuasiPrimitive := false )
```

75.3.4 TestInducedFromNormalSubgroup

▷ `TestInducedFromNormalSubgroup(chi[, N])` (function)
▷ `IsInducedFromNormalSubgroup(chi)` (property)

`TestInducedFromNormalSubgroup` returns a record with information whether the irreducible character *chi* of the group *G*, say, is induced from a proper normal subgroup of *G*. If the second argument *N* is present, which must be a normal subgroup of *G* or the list of class positions of a normal subgroup of *G*, it is checked whether *chi* is induced from *N*.

The result contains always the components `isInduced` (either `true` or `false`) and `comment` (a string telling a reason for the value of the component `isInduced`). In the `true` case there is a component `character` which contains a character of a maximal normal subgroup from which *chi* is induced.

`IsInducedFromNormalSubgroup` returns `true` if *chi* is induced from a proper normal subgroup of *G*, and `false` otherwise.

Example

```
gap> List( Irr( Sl23 ), IsInducedFromNormalSubgroup );
[ false, false, false, false, false, false, true ]
gap> List( Irr( S4 ){ [ 1, 3, 4 ] },
>          TestInducedFromNormalSubgroup );
[ rec( comment := "linear character", isInduced := false ),
  rec( character := Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
        [ 1, 1, E(3)^2, E(3) ] ),
      comment := "induced from component '.character'",
      isInduced := true ),
  rec( comment := "all maximal normal subgroups checked",
      isInduced := false ) ]
```

75.4 Testing Monomiality

A character χ of a finite group *G* is called *monomial* if χ is induced from a linear character of a subgroup of *G*. A finite group *G* is called *monomial* (or *M-group*) if each ordinary irreducible character of *G* is monomial.

75.4.1 TestMonomial

▷ `TestMonomial(chi)` (attribute)
▷ `TestMonomial(G)` (attribute)
▷ `TestMonomial(chi, uselattice)` (operation)
▷ `TestMonomial(G, uselattice)` (operation)

Called with a group character *chi* of a group *G*, `TestMonomial` returns a record containing information about monomiality of the group *G* or the group character *chi*, respectively.

If `TestMonomial` proves the character *chi* to be monomial then the result contains components `isMonomial` (with value `true`), `comment` (a string telling a reason for monomiality), and if it was necessary to compute a linear character from which *chi* is induced, also a component `character`.

If TestMonomial proves *chi* or *G* to be nonmonomial then the value of the component isMonomial is false, and in the case of *G* a nonmonomial character is the value of the component character if it had been necessary to compute it.

A Boolean can be entered as the second argument *uselattice*; if the value is true then the subgroup lattice of the underlying group is used if necessary, if the value is false then the subgroup lattice is used only for groups of order at most TestMonomialUseLattice (75.4.2). The default value of *uselattice* is false.

For a group whose lattice must not be used, it may happen that TestMonomial cannot prove or disprove monomiality; then the result record contains the component isMonomial with value "?". This case occurs in the call for a character *chi* if and only if *chi* is not induced from the inertia subgroup of a component of any reducible restriction to a normal subgroup. It can happen that *chi* is monomial in this situation. For a group, this case occurs if no irreducible character can be proved to be nonmonomial, and if no decision is possible for at least one irreducible character.

Example

```
gap> TestMonomial( S4 );
rec( comment := "abelian by supersolvable group", isMonomial := true )
gap> TestMonomial( Sl23 );
rec( comment := "list Delta( G ) contains entry > 1",
  isMonomial := false )
```

75.4.2 TestMonomialUseLattice

▷ TestMonomialUseLattice (global variable)

This global variable controls for which groups the operation TestMonomial (75.4.1) may compute the subgroup lattice. The value can be set to a positive integer or infinity (18.2.1), the default is 1000.

75.4.3 IsMonomialNumber

▷ IsMonomialNumber(*n*) (property)

For a positive integer *n*, IsMonomialNumber returns true if every solvable group of order *n* is monomial, and false otherwise. One can also use IsMonomial instead.

Let $\nu_p(n)$ denote the multiplicity of the prime p as factor of n, and $ord(p,q)$ the multiplicative order of $p \pmod q$.

Then there exists a solvable nonmonomial group of order n if and only if one of the following conditions is satisfied.

1. $\nu_2(n) \geq 2$ and there is a p such that $\nu_p(n) \geq 3$ and $p \equiv -1 \pmod 4$,

2. $\nu_2(n) \geq 3$ and there is a p such that $\nu_p(n) \geq 3$ and $p \equiv 1 \pmod 4$,

3. there are odd prime divisors p and q of n such that $ord(p,q)$ is even and $ord(p,q) < \nu_p(n)$ (especially $\nu_p(n) \geq 3$),

4. there is a prime divisor q of n such that $\nu_2(n) \geq 2ord(2,q)+2$ (especially $\nu_2(n) \geq 4$),

5. $v_2(n) \geq 2$ and there is a p such that $p \equiv 1 \pmod 4$, $ord(p,q)$ is odd, and $2ord(p,q) < v_p(n)$ (especially $v_p(n) \geq 3$).

These five possibilities correspond to the five types of solvable minimal nonmonomial groups (see `MinimalNonmonomialGroup` (75.5.2)) that can occur as subgroups and factor groups of groups of order `n`.

Example

```
gap> Filtered( [ 1 .. 111 ], x -> not IsMonomial( x ) );
[ 24, 48, 72, 96, 108 ]
```

75.4.4 TestMonomialQuick

▷ `TestMonomialQuick(chi)` (attribute)
▷ `TestMonomialQuick(G)` (attribute)

`TestMonomialQuick` does some cheap tests whether the irreducible character `chi` or the group `G`, respectively, is monomial. Here "cheap" means in particular that no computations of character tables are involved. The return value is a record with components

`isMonomial`
: either `true` or `false` or the string `"?"`, depending on whether (non)monomiality could be proved, and

`comment`
: a string telling the reason for the value of the `isMonomial` component.

A group `G` is proved to be monomial by `TestMonomialQuick` if `G` is nilpotent or Sylow abelian by supersolvable, or if `G` is solvable and its order is not divisible by the third power of a prime, Nonsolvable groups are proved to be nonmonomial by `TestMonomialQuick`.

An irreducible character `chi` is proved to be monomial if it is linear, or if its codegree is a prime power, or if its group knows to be monomial, or if the factor group modulo the kernel can be proved to be monomial by `TestMonomialQuick`.

Example

```
gap> TestMonomialQuick( Irr( S4 )[3] );
rec( comment := "whole group is monomial", isMonomial := true )
gap> TestMonomialQuick( S4 );
rec( comment := "abelian by supersolvable group", isMonomial := true )
gap> TestMonomialQuick( Sl23 );
rec( comment := "no decision by cheap tests", isMonomial := "?" )
```

75.4.5 TestSubnormallyMonomial

▷ `TestSubnormallyMonomial(G)` (attribute)
▷ `TestSubnormallyMonomial(chi)` (attribute)
▷ `IsSubnormallyMonomial(G)` (property)
▷ `IsSubnormallyMonomial(chi)` (property)

A character of the group G is called *subnormally monomial* (SM for short) if it is induced from a linear character of a subnormal subgroup of G. A group G is called SM if all its irreducible characters are SM.

`TestSubnormallyMonomial` returns a record with information whether the group `G` or the irreducible character `chi` of `G` is SM.

The result has the components `isSubnormallyMonomial` (either `true` or `false`) and `comment` (a string telling a reason for the value of the component `isSubnormallyMonomial`); in the case that the `isSubnormallyMonomial` component has value `false` there is also a component `character`, with value an irreducible character of G that is not SM.

`IsSubnormallyMonomial` returns `true` if the group `G` or the group character `chi` is subnormally monomial, and `false` otherwise.

Example

```
gap> TestSubnormallyMonomial( S4 );
rec( character := Character( CharacterTable( S4 ), [ 3, -1, -1, 0, 1
     ] ), comment := "found non-SM character",
  isSubnormallyMonomial := false )
gap> TestSubnormallyMonomial( Irr( S4 )[4] );
rec( comment := "all subnormal subgroups checked",
  isSubnormallyMonomial := false )
gap> TestSubnormallyMonomial( DerivedSubgroup( S4 ) );
rec( comment := "all irreducibles checked",
  isSubnormallyMonomial := true )
```

75.4.6 TestRelativelySM

▷ `TestRelativelySM(G)` (attribute)
▷ `TestRelativelySM(chi)` (attribute)
▷ `TestRelativelySM(G, N)` (operation)
▷ `TestRelativelySM(chi, N)` (operation)
▷ `IsRelativelySM(G)` (property)
▷ `IsRelativelySM(chi)` (property)

In the first two cases, `TestRelativelySM` returns a record with information whether the argument, which must be a SM group `G` or an irreducible character `chi` of a SM group G, is relatively SM with respect to every normal subgroup of `G`.

In the second two cases, a normal subgroup `N` of `G` is the second argument. Here `TestRelativelySM` returns a record with information whether the first argument is relatively SM with respect to `N`, i.e, whether there is a subnormal subgroup H of G that contains `N` such that the character `chi` resp. every irreducible character of G is induced from a character ψ of H such that the restriction of ψ to `N` is irreducible.

The result record has the components `isRelativelySM` (with value either `true` or `false`) and `comment` (a string that describes a reason). If the argument is a group `G` that is not relatively SM with respect to a normal subgroup then additionally the component `character` is bound, with value a not relatively SM character of such a normal subgroup.

`IsRelativelySM` returns `true` if the SM group `G` or the irreducible character `chi` of the SM group `G` is relatively SM with respect to every normal subgroup of `G`, and `false` otherwise.

Note that it is *not* checked whether `G` is SM.

Example

```
gap> IsSubnormallyMonomial( DerivedSubgroup( S4 ) );
true
gap> TestRelativelySM( DerivedSubgroup( S4 ) );
rec(
  comment := "normal subgroups are abelian or have nilpotent factor gr\
oup", isRelativelySM := true )
```

75.5 Minimal Nonmonomial Groups

75.5.1 IsMinimalNonmonomial

▷ `IsMinimalNonmonomial(`*G*`)` (property)

A group *G* is called *minimal nonmonomial* if it is nonmonomial, and all proper subgroups and factor groups are monomial.

Example

```
gap> IsMinimalNonmonomial( Sl23 );  IsMinimalNonmonomial( S4 );
true
false
```

75.5.2 MinimalNonmonomialGroup

▷ `MinimalNonmonomialGroup(`*p, factsize*`)` (function)

is a solvable minimal nonmonomial group described by the parameters *factsize* and *p* if such a group exists, and `false` otherwise.

Suppose that the required group K exists. Then *factsize* is the size of the Fitting factor $K/F(K)$, and this value is 4, 8, an odd prime, twice an odd prime, or four times an odd prime. In the case that *factsize* is twice an odd prime, the centre $Z(K)$ is cyclic of order 2^{p+1}. In all other cases *p* is the (unique) prime that divides the order of $F(K)$.

The solvable minimal nonmonomial groups were classified by van der Waall, see [vdW76].

Example

```
gap> MinimalNonmonomialGroup(  2,  3 ); # the group SL(2,3)
2^(1+2):3
gap> MinimalNonmonomialGroup(  3,  4 );
3^(1+2):4
gap> MinimalNonmonomialGroup(  5,  8 );
5^(1+2):Q8
gap> MinimalNonmonomialGroup( 13, 12 );
13^(1+2):2.D6
gap> MinimalNonmonomialGroup(  1, 14 );
2^(1+6):D14
gap> MinimalNonmonomialGroup(  2, 14 );
(2^(1+6)Y4):D14
```

Chapter 76

Using GAP Packages

The functionality of GAP can be extended by loading GAP packages. GAP distribution already contains all currently redistributed with GAP packages in the `gap4r7/pkg` directory.

GAP packages are written by (groups of) GAP users which may not be members of the GAP developer team. The responsibility and copyright of a GAP package remains with the original author(s).

GAP packages have their own documentation which is smoothly integrated into the GAP help system.

All GAP users who develop new code are invited to share the results of their efforts with other GAP users by making the code and its documentation available in form of a package. Information how to do this is available from the GAP Web pages (`http://www.gap-system.org`) and in the GAP package Example (see `http://www.gap-system.org/Packages/example.html`). There are possibilities to get a package distributed together with GAP and it is possible to submit a package to a formal refereeing process.

In this chapter we describe how to use existing packages.

76.1 Installing a GAP Package

Before a package can be used it must be installed. With a standard installation of GAP there should be all currently redistributed with GAP packages already available. But since GAP packages are released independently of the main GAP system it may be sensible to upgrade or install new packages between upgrades of your GAP installation.

A package consists of a collection of files within a single directory that must be a subdirectory of the `pkg` directory in one of the GAP root directories, see 9.2. (If you don't have access to the `pkg` directory in your main GAP installation you can add private root directories as explained in that section.)

Whenever you get from somewhere an archive of a GAP package it should be accompanied with a `README` file that explains its installation. Some packages just consist of GAP code and the installation is done by unpacking the archive in one of the places described above. There are also packages that need further installation steps, there may be for example some external programs which have to be compiled (this is often done by just saying `./configure; make` inside the unpacked package directory, but check the individual `README` files). Note that if you use Windows you may not be able to use some or all external binaries.

76.2 Loading a GAP Package

Some GAP packages are prepared for *automatic loading*, that is they will be loaded automatically with GAP, others must in each case be separately loaded by a call to LoadPackage (76.2.1).

76.2.1 LoadPackage

▷ LoadPackage(*name[, version][, banner]*) (function)

loads the GAP package with name *name*. If the optional version string *version* is given, the package will only be loaded in a version number at least as large as *version*, or equal to *version* if its first character is = (see CompareVersionNumbers (76.3.6)). The argument *name* is case insensitive.

LoadPackage will return true if the package has been successfully loaded and will return fail if the package could not be loaded. The latter may be the case if the package is not installed, if necessary binaries have not been compiled, or if the version number of the available version is too small. If the package cannot be loaded, TestPackageAvailability (76.3.2) can be used to find the reasons. Also, DisplayPackageLoadingLog (76.2.4) can be used to find out more about the failure reasons. To see the problems directly, one can change the verbosity using the user preference InfoPackageLoadingLevel, see InfoPackageLoading (76.2.4) for details.

If the package *name* has already been loaded in a version number at least or equal to *version*, respectively, LoadPackage returns true without doing anything else.

The argument *name* may be the prefix of a package name. If no package with name *name* is installed, the behaviour is as follows. If *name* is the prefix of exactly one name of an installed package then LoadPackage is called with this name; if the names of several installed packages start with *name* then the these names are printed, and LoadPackage returns fail. Thus the names of *all* installed packages can be shown by calling LoadPackage with an empty string.

If the optional argument *banner* is present then it must be either true or false; in the latter case, the effect is that no package banner is printed.

After a package has been loaded its code and documentation should be available as other parts of the GAP library are.

When GAP is started then some packages are loaded automatically. These are the packages listed in GAPInfo.Dependencies.NeededOtherPackages and (if this is not disabled, see below) UserPreference("PackagesToLoad").

A GAP package may also install only its documentation automatically but still need loading by LoadPackage. In this situation the online help displays (not loaded) in the header lines of the manual pages belonging to this GAP package.

If for some reason you don't want certain packages to be automatically loaded, GAP provides three levels for disabling autoloading:

The autoloading of specific packages can be overwritten *for the whole GAP installation* by putting a file NOAUTO into a pkg directory that contains lines with the names of packages which should not be automatically loaded.

Furthermore, *individual users* can disable the autoloading of specific packages by putting the names of these packages into the list that is assigned to the user preference "ExcludeFromAutoload", for example in the user's gap.ini file (see 3.2.1).

Using the -A command line option when starting up GAP (see 3.1), automatic loading of packages is switched off *for this GAP session*.

In any of the above three cases, the packages listed in GAPInfo.Dependencies.NeededOtherPackages are still loaded automatically, and an error is signalled if not all of these packages are available.

See SetPackagePath (76.2.2) for a possibility to force that a prescribed package version will be loaded. See also ExtendRootDirectories (76.2.3) for a possibility to add directories containing packages after GAP has been started.

76.2.2 SetPackagePath

▷ SetPackagePath(*pkgname, pkgpath*) (function)

Let *pkgname* and *pkgpath* be strings denoting the name of a GAP package and the path to a directory where a version of this package can be found (i. e., calling Directory (9.3.2) with the argument *pkgpath* will yield a directory that contains the file PackageInfo.g of the package).

If the package *pkgname* is already loaded with an installation path different from *pkgpath* then SetPackagePath signals an error. If the package *pkgname* is not yet loaded then SetPackagePath erases the information about available versions of the package *pkgname*, and stores the record that is contained in the PackageInfo.g file at *pkgpath* instead, such that only the version installed at *pkgpath* can be loaded with LoadPackage (76.2.1).

This function can be used to force GAP to load a particular version of a package, although newer versions of the package might be available.

76.2.3 ExtendRootDirectories

▷ ExtendRootDirectories(*paths*) (function)

Let *paths* be a list of strings that denote paths to intended GAP root directories (see 9.2). The function ExtendRootDirectories adds these paths to the global list GAPInfo.RootPaths and calls the initialization of available GAP packages, such that later calls to LoadPackage (76.2.1) will find the GAP packages that are contained in pkg subdirectories of the directories given by *paths*.

Note that the purpose of this function is to make GAP packages in the given directories available. It cannot be used to influence the start of GAP, because the GAP library is loaded before ExtendRootDirectories can be called (and because GAPInfo.RootPaths is not used for reading the GAP library).

76.2.4 DisplayPackageLoadingLog

▷ DisplayPackageLoadingLog(*[severity]*) (function)
▷ InfoPackageLoading (info class)
▷ PACKAGE_ERROR (global variable)
▷ PACKAGE_WARNING (global variable)
▷ PACKAGE_INFO (global variable)
▷ PACKAGE_DEBUG (global variable)
▷ LogPackageLoadingMessage(*severity, message[, name]*) (function)

Whenever GAP considers loading a package, log messages are collected in a global list. The messages for the current GAP session can be displayed with DisplayPackageLoadingLog. To each

message, a "severity" is assigned, which is one of PACKAGE_ERROR (76.2.4), PACKAGE_WARNING (76.2.4), PACKAGE_INFO (76.2.4), PACKAGE_DEBUG (76.2.4), in increasing order. The function DisplayPackageLoadingLog shows only the messages whose severity is at most *severity*, the default for *severity* is PACKAGE_WARNING (76.2.4).

The intended meaning of the severity levels is as follows.

PACKAGE_ERROR
should be used whenever GAP will run into an error during package loading, where the reason of the error shall be documented in the global list.

PACKAGE_WARNING
should be used whenever GAP has detected a reason why a package cannot be loaded, and where the message describes how to solve this problem, for example if a package binary is missing.

PACKAGE_INFO
should be used whenever GAP has detected a reason why a package cannot be loaded, and where it is not clear how to solve this problem, for example if the package is not compatible with other installed packages.

PACKAGE_DEBUG
should be used for other messages reporting what GAP does when it loads packages (checking dependencies, reading files, etc.). One purpose is to record in which order packages have been considered for loading or have actually been loaded.

The log messages are created either by the functions of GAP's package loading mechanism or in the code of your package, for example in the AvailabilityTest function of the package's PackageInfo.g file (see 76.3.11), using LogPackageLoadingMessage. The arguments of this function are *severity* (which must be one of the above severity levels), *message* (which must be either a string or a list of strings), and optionally *name* (which must be the name of the package to which the message belongs). The argument *name* is not needed if the function is called from a call of a package's AvailabilityTest function (see 76.3.11) or is called from a package file that is read from init.g or read.g; in these cases, the name of the current package (stored in the record GAPInfo.PackageCurrent) is taken. According to the above list, the *severity* argument of LogPackageLoadingMessage calls in a package's AvailabilityTest function is either PACKAGE_WARNING (76.2.4) or PACKAGE_INFO (76.2.4).

If you want to see the log messages already during the package loading process, you can set the level of the info class InfoPackageLoading to one of the severity values listed above; afterwards the messages with at most this severity are shown immediately when they arise. In order to make this work already for autoloaded packages, you can call SetUserPreference("InfoPackageLoadingLevel", *lev*); to set the desired severity level *lev*. This can for example be done in your gap.ini file, see Section 3.2.1.

76.3 Functions for GAP Packages

The following functions are mainly used in files contained in a package and not by users of a package.

76.3.1 ReadPackage

▷ ReadPackage([name,]file) (function)
▷ RereadPackage([name,]file) (function)

Called with two strings *name* and *file*, ReadPackage reads the file *file* of the GAP package *name*, where *file* is given as a path relative to the home directory of *name*. Note that *file* is read in the namespace of the package, see Section 4.10 for details.

If only one argument *file* is given, this should be the path of a file relative to the pkg subdirectory of GAP root paths (see 9.2). Note that in this case, the package name is assumed to be equal to the first part of *file*, *so the one argument form is not recommended*.

The absolute path is determined as follows. If the package in question has already been loaded then the file in the directory of the loaded version is read. If the package is available but not yet loaded then the directory given by TestPackageAvailability (76.3.2) is used, without prescribed version number. (Note that the ReadPackage call does *not* force the package to be loaded.)

If the file is readable then true is returned, otherwise false.

Each of *name* and *file* should be a string. The *name* argument is case insensitive.

RereadPackage does the same as ReadPackage, except that also read-only global variables are overwritten (cf. Reread (9.7.9)).

76.3.2 TestPackageAvailability

▷ TestPackageAvailability(name[, version][, checkall]) (function)

For strings *name* and *version*, this function tests whether the GAP package *name* is available for loading in a version that is at least *version*, or equal to *version* if the first character of *version* is = (see CompareVersionNumbers (76.3.6) for further details about version numbers).

The result is true if the package is already loaded, fail if it is not available, and the string denoting the GAP root path where the package resides if it is available, but not yet loaded. So the package *name* is available if the result of TestPackageAvailability is not equal to fail.

If the optional argument *checkall* is true then all dependencies are checked, even if some have turned out to be not satisfied. This is useful when one is interested in the reasons why the package *name* cannot be loaded. In this situation, calling first TestPackageAvailability and then DisplayPackageLoadingLog (76.2.4) with argument PACKAGE_INFO (76.2.4) will give an overview of these reasons.

You should *not* call TestPackageAvailability in the test function of a package (the value of the component AvailabilityTest in the PackageInfo.g file of the package, see 76.3.11), because TestPackageAvailability calls this test function.

The argument *name* is case insensitive.

76.3.3 InstalledPackageVersion

▷ InstalledPackageVersion(name) (function)

If the GAP package with name *name* has already been loaded then InstalledPackageVersion returns the string denoting the version number of this version of the package. If the package is available but has not yet been loaded then the version number string for that version of the package that currently would be loaded. (Note that loading *another* package might force loading another version

of the package *name*, so the result of InstalledPackageVersion will be different afterwards.) If the package is not available then fail is returned.

The argument *name* is case insensitive.

76.3.4 DirectoriesPackageLibrary

▷ DirectoriesPackageLibrary(*name[, path]*) (function)

takes the string *name*, a name of a GAP package, and returns a list of directory objects for those sub-directory/ies containing the library functions of this GAP package, for the version that is already loaded or is currently going to be loaded or would be the first version GAP would try to load if no other version is explicitly prescribed. (If the package *name* is not yet loaded then we cannot guarantee that the returned directories belong to a version that really can be loaded.)

The default is that the library functions are in the subdirectory lib of the GAP package's home directory. If this is not the case, then the second argument *path* needs to be present and must be a string that is a path name relative to the home directory of the GAP package with name *name*.

Note that DirectoriesPackageLibrary is likely to be called in the AvailabilityTest function in the package's PackageInfo.g file (see 76.3.11).

As an example, the following returns a directory object for the library functions of the GAP package Example:

Example

```
gap> DirectoriesPackageLibrary( "Example", "gap" );
[ dir("/home/werner/gap/4.0/pkg/example/gap/") ]
```

Observe that we needed the second argument "gap" here, since Example's library functions are in the subdirectory gap rather than lib.

In order to find a subdirectory deeper than one level in a package directory, the second argument is again necessary whether or not the desired subdirectory relative to the package's directory begins with lib. The directories in *path* should be separated by / (even on systems, like Windows, which use \ as the directory separator). For example, suppose there is a package somepackage with a subdirectory m11 in the directory data, then we might expect the following:

Example

```
gap> DirectoriesPackageLibrary( "somepackage", "data/m11" );
[ dir("/home/werner/gap/4.0/pkg/somepackage/data/m11") ]
```

76.3.5 DirectoriesPackagePrograms

▷ DirectoriesPackagePrograms(*name*) (function)

returns a list of the bin/*architecture* subdirectories of all packages *name* where *architecture* is the architecture on which GAP has been compiled (this can be accessed as GAPInfo.Architecture, see GAPInfo (3.5.1)) and the version of the installed package coincides with the version of the package *name* that is already loaded or is currently going to be loaded or would be the first version GAP would try to load if no other version is explicitly prescribed. (If the package *name* is not yet loaded then we cannot guarantee that the returned directories belong to a version that really can be loaded.)

Note that DirectoriesPackagePrograms is likely to be called in the AvailabilityTest function in the package's PackageInfo.g file (see 76.3.11).

The directories returned by DirectoriesPackagePrograms are the place where external binaries of the GAP package *name* for the current package version and the current architecture should be located.

Example

```
gap> DirectoriesPackagePrograms( "nq" );
[ dir("/home/gap/4.0/pkg/nq/bin/x86_64-unknown-linux-gnu-gcc/64-bit/"),
  dir("/home/gap/4.0/pkg/nq/bin/x86_64-unknown-linux-gnu-gcc/") ]
```

76.3.6 CompareVersionNumbers

▷ CompareVersionNumbers(*supplied, required[, "equal"]*) (function)

A version number is a string which contains nonnegative integers separated by non-numeric characters. Examples of valid version numbers are for example:

Example

```
"1.0"   "3.141.59"  "2-7-8.3" "5 release 2 patchlevel 666"
```

CompareVersionNumbers compares two version numbers, given as strings. They are split at non-digit characters, the resulting integer lists are compared lexicographically. The routine tests whether *supplied* is at least as large as *required*, and returns true or false accordingly. A version number ending in dev is considered to be infinite.

76.3.7 IsPackageMarkedForLoading

▷ IsPackageMarkedForLoading(*name, version*) (function)

This function can be used in the code of a package *A*, say, for testing whether the package *name* in version *version* will be loaded after the LoadPackage (76.2.1) call for the package *A* has been executed. This means that the package *name* had been loaded before, or has been (directly or indirectly) requested as a needed or suggested package of the package *A* or of a package whose loading requested that *A* was loaded.

76.3.8 DeclareAutoreadableVariables

▷ DeclareAutoreadableVariables(*pkgname, filename, varlist*) (function)

Let *pkgname* be the name of a package, let *filename* be the name of a file relative to the home directory of this package, and let *varlist* be a list of strings that are the names of global variables which get bound when the file is read. DeclareAutoreadableVariables notifies the names in *varlist* such that the first attempt to access one of the variables causes the file to be read.

76.3.9 Kernel modules in GAP packages

If the package has a kernel module, then it can be compiled using the gac script. A kernel module is implemented in C and follows certain conventions to comply with the GAP kernel interface, which

we plan to document later. In the meantime, we advice to get in touch with GAP developers if you plan to develop such a package.

To use the gac script to produce dynamically loadable modules, call it with the `-d` option, for example:

Example

```
$ gap4/bin/i386-ibm-linux-gcc2/gac -d test.c
```

This will produce a file `test.so`, which then can be loaded into GAP with `LoadDynamicModule` (76.3.10).

76.3.10 LoadDynamicModule

▷ `LoadDynamicModule(`*`filename[, crc]`*`)` (function)

To load a compiled file, the command `LoadDynamicModule` is used. This command loads *`filename`* as module. If given, the CRC checksum *`crc`* must match the value of the module (see 9.7.7).

Example

```
gap> LoadDynamicModule("./test.so");
gap> CrcFile("test.so");
2906458206
gap> LoadDynamicModule("./test.so",1);
Error, <crc> mismatch (or no support for dynamic loading) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> LoadDynamicModule("./test.so",2906458206);
```

On some operating systems, once you have loaded a dynamic module with a certain filename, loading another with the same filename will have no effect, even if the file on disk has changed.

76.3.11 The PackageInfo.g File

Each package has the file `PackageInfo.g` which contains meta-information about the package (package name, version, author(s), relations to other packages, homepage, download archives, banner, etc.). This file is used by the package loading mechanism and also for the distribution of a package to other users.

76.3.12 ValidatePackageInfo

▷ `ValidatePackageInfo(`*`info`*`)` (function)

This function is intended to support package authors who create or modify `PackageInfo.g` files. (It is *not* called when these files are read during the startup of GAP or when packages are actually loaded.)

The argument *`info`* must be either a record as is contained in a `PackageInfo.g` file or a a string which describes the path to such a file. The result is `true` if the record or the contents of the file,

respectively, has correct format, and `false` otherwise; in the latter case information about the incorrect components is printed.

Note that the components used for package loading are checked as well as the components that are needed for composing the package overview web page or for updating the package archives.

If *info* is a string then `ValidatePackageInfo` checks additionally whether those package files exist that are mentioned in the file `info`, for example the `manual.six` file of the package documentation.

76.3.13 ShowPackageVariables

▷ `ShowPackageVariables(`*pkgname[, version][, arec]*`)` (function)
▷ `PackageVariablesInfo(`*pkgname, version*`)` (function)

Let *pkgname* be the name of a GAP package. If the package *pkgname* is available but not yet loaded then `ShowPackageVariables` prints a list of global variables that become bound and of methods that become installed when the package is loaded. (For that, GAP actually loads the package.)

If a version number *version* is given (see Section (**Example: Version Numbers**)) then this version of the package is considered.

An error message is printed if (the given version of) the package is not available or already loaded.

Information is printed about new and redeclared global variables, and about names of global variables introduced in the package that differ from existing globals only by case; note that the GAP help system is case insensitive, so it is difficult to document identifiers that differ only by case.

Info lines for undocumented variables are marked with an asterisk `*`.

The following entries are omitted from the list: default setter methods for attributes and properties that are declared in the package, and `Set`*attr* and `Has`*attr* type variables where *attr* is an attribute or property.

The output can be customized using the optional record *arec*, the following components of this record are supported.

`show`
: a list of strings describing those kinds of variables which shall be shown, such as `"new global functions"`; the default are all kinds that appear in the package,

`showDocumented`
: `true` (the default) if documented variables shall be shown, and `false` otherwise,

`showUndocumented`
: `true` (the default) if undocumented variables shall be shown, and `false` otherwise,

`showPrivate`
: `true` (the default) if variables from the package's name space (see Section 4.10) shall be shown, and `false` otherwise,

`Display`
: a function that takes a string and shows it on the screen; the default is `Print` (6.3.4), another useful value is `Pager` (2.4.1).

An interactive variant of `ShowPackageVariables` is the function `BrowsePackageVariables` (**Browse: BrowsePackageVariables**) that is provided by the GAP package Browse. For this function, it is not sensible to assume that the package *pkgname* is not yet loaded before the function call,

because one might be interested in packages that must be loaded before Browse itself can be loaded. The solution is that BrowsePackageVariables (**Browse: BrowsePackageVariables**) takes the output of PackageVariablesInfo as its second argument. The function PackageVariablesInfo is used by both ShowPackageVariables and BrowsePackageVariables (**Browse: BrowsePackageVariables**) for collecting the information about the package in question, and can be called before the package Browse is loaded.

76.3.14 BibEntry

▷ BibEntry(*pkgname[, key]*) (function)

Returns: a string in BibXMLext format (see (**GAPDoc: The BibXMLext Format**)) that can be used for referencing the GAP system or a GAP package.

If the argument *pkgname* is the string "GAP", the function returns an entry for the current version of GAP.

Otherwise, if a string *pkgname* is given, which is the name of a GAP package, an entry for this package is returned; this entry is computed from the PackageInfo.g file of *the current version* of the package, see InstalledPackageVersion (76.3.3). If no package with name *pkgname* is installed then the empty string is returned.

A string for *a different version* of GAP or a package can be computed by entering, as the argument *pkgname*, the desired record from the PackageInfo.g file. (One can access these records using the function PackageInfo.)

In each of the above cases, an optional argument *key* can be given, a string which is then used as the key of the BibTeX entry instead of the default key that is generated from the system/package name and the version number.

BibEntry requires the functions FormatParagraph (**GAPDoc: FormatParagraph**) and NormalizedNameAndKey (**GAPDoc: NormalizedNameAndKey**) from the GAP package GAPDoc.

The functions ParseBibXMLextString (**GAPDoc: ParseBibXMLextString**) and StringBibXMLEntry (**GAPDoc: StringBibXMLEntry**) can be used to create for example a BibTeX entry from the return value, as follows.

Example

```
gap> bib:= BibEntry( "GAP", "GAP4.5" );;
gap> Print( bib, "\n" );
<entry id="GAP4.5"><misc>
  <title><C>GAP</C> &ndash; <C>G</C>roups, <C>A</C>lgorithms,
         and <C>P</C>rogramming, <C>V</C>ersion 4.5.1</title>
  <howpublished><URL>http://www.gap-system.org</URL></howpublished>
  <key>GAP</key>
  <keywords>groups; *; gap; manual</keywords>
  <other type="organization">The GAP <C>G</C>roup</other>
</misc></entry>
gap> parse:= ParseBibXMLextString( bib );;
gap> Print( StringBibXMLEntry( parse.entries[1], "BibTeX" ) );
@misc{ GAP4.5,
  title =            {{GAP}   {\textendash}   {G}roups,   {A}lgorithms,  and
                      {P}rogramming, {V}ersion 4.5.1},
  organization =     {The GAP {G}roup},
  howpublished =     {\href                        {http://www.gap-system.org}
                      {\texttt{http://www.gap-system.org}}},
  key =              {GAP},
```

```
  keywords =          {groups; *; gap; manual}
}
```

76.3.15 Cite

▷ `Cite(`*`[pkgname[, key]]`*`)` (function)

Used with no arguments or with argument `"GAP"` (case-insensitive), `Cite` displays instructions on citing the version of GAP that is being used. Suggestions are given in plain text, HTML, BibXML and BibTeX formats. The same instructions are also contained in the `CITATION` file in the GAP root directory.

If *pkgname* is the name of a GAP package, instructions on citing this package will be displayed. They will be produced from the `PackageInfo.g` file of the working version of this package that must be available in the GAP installation being used. Otherwise, one will get a warning that no working version of the package is available.

The optional 2nd argument *key* has the same meaning as in `BibEntry` (76.3.14).

Chapter 77

Replaced and Removed Command Names

In general we try to keep GAP 4 compatible with former releases as much as possible. Nevertheless, from time to time it seems appropriate to remove some commands or to change the names of some commands or variables. There are various reasons for that: Some functionality was improved and got another (hopefully better) interface, names turned out to be too special or too general for the underlying functionality, or names are found to be unintuitive or inconsistent with other names.

In this chapter we collect such old names while pointing to the sections which explain how to substitute them. Usually, old names will be available for several releases; they may be removed when they don't seem to be used any more.

The obsolete GAP code is collected in two library files, `lib/obsolete.gd` and `lib/obsolete.gi`. By default, these files are read when GAP is started. It may be useful to omit reading these files, for example in order to make sure that one's own GAP code does not rely on the obsolete variables. For that, one can use the `-O` command line option (see 3.1) or set the component `ReadObsolete` in the file `gap.ini` to `false` (see 3.2). Note that `-O` command line option overrides `ReadObsolete`.

(Note that the condition whether the library files with the obsolete GAP code shall be read has changed. In GAP 4.3 and 4.4, the global variables `GAP_OBSOLESCENT` and `GAPInfo.ReadObsolete` –to be set in the user's `.gaprc` file– were used to control this behaviour.)

77.1 Group Actions – Name Changes

The concept of a group action is sometimes referred to as a "group operation". In GAP 3 as well as in older versions of GAP 4 the term `Operation` was used instead of `Action`. We decided to change the names to avoid confusion with the term "operation" as in `DeclareOperation` (79.18.6) and "operations for Xyz".

Here are some examples of such name changes.

OLD	*NOW USE*
`Operation`	`Action` (41.7.2)
`RepresentativeOperation`	`RepresentativeAction` (41.6.1)
`OperationHomomorphism`	`ActionHomomorphism` (41.7.1)
`FunctionOperation`	`FunctionAction` (41.12.4)

77.2 Package Interface – Obsolete Functions and Name Changes

With GAP 4.4 the package interface was changed. Thereby some functions became obsolete and the names of some others were made more consistent.

The following functions are no longer needed: `DeclarePackage`, `DeclareAutoPackage`, `DeclarePackageDocumentation` and `DeclarePackageAutoDocumentation`. They are substituted by entries in the packages' `PackageInfo.g` files, see 76.3.11.

Furthermore, the global variable `PACKAGES_VERSIONS` is no longer needed, since this information is now contained in the `GAPInfo.PackagesInfo` record (see 3.5.1). The global variable `Revisions` is also no longer needed, since the function `DisplayRevision` was made obsolete in GAP 4.5.

The following function names were changed.

OLD	*NOW USE*
`RequirePackage`	`LoadPackage` (76.2.1)
`ReadPkg`	`ReadPackage` (76.3.1)
`RereadPkg`	`RereadPackage` (76.3.1)

77.3 Normal Forms of Integer Matrices – Name Changes

Former versions of GAP 4 documented several functions for computing the Smith or Hermite normal form of integer matrices. Some of them were never implemented and it was unclear which commands to use. The functionality of all of these commands is now available with `NormalFormIntMat` (25.2.9) and a few interface functions.

77.4 Miscellaneous Name Changes or Removed Names

In former releases of GAP 4 there were some global variable names bound to general information about the running GAP, such as path names or command line options. Although they were not officially documented they were used by several users and in some packages. We mention here `BANNER` and `QUIET`. This type of information is now collected in the global record `GAPInfo` (3.5.1).

Here are some further name changes.

OLD	*NOW USE*
`MonomialTotalDegreeLess`	`MonomialExtGrlexLess` (66.17.14)
`NormedVectors`	`NormedRowVectors` (61.9.11)
`MutableIdentityMat`	`IdentityMat` (24.5.1)
`MutableNullMat`	`NullMat` (24.5.2)

77.4.1 InfoObsolete

▷ `InfoObsolete` (global variable)

is an info class to display warnings when an obsolete variable is used. By default, these warnings are switched off since the info level for this class is 0. Setting it to 1 will trigger warnings if GAP will detect that an obsolete variable is used at runtime (this detection is possible, however, only for obsolete variables declared using `DeclareObsoleteSynonym`).

To check that the GAP code does not use obsolete variables at the parsing time, and not at a runtime, use `-O` command line option, see 3.1.

77.5 The former .gaprc file

Up to GAP 4.4, a file `.gaprc` in the user's home directory (if available, and GAP was started without `-r` option) was read automatically during startup, early enough for influencing the autoloading of packages and late enough for being allowed to execute any GAP code. On Windows machines this file was called `gap.rc`.

In GAP 4.5 the startup mechanism has changed, see 3.2 for details. These new configuration files are now contained in a directory `GAPInfo.UserGapRoot`.

For the sake of partial backwards compatibility, also the former file `~/.gaprc` is still supported for such initializations, but this file is read only if the directory `GAPInfo.UserGapRoot` does not exist. In that case the `~/.gaprc` is read at the same time as `gaprc` would be read, i. e., too late for influencing the startup of GAP.

As before, the command line option `-r` disables reading `~/.gaprc`, see 3.1.

To migrate from the old setup to the new one introduced with GAP 4.5, first have a look at the function `WriteGapIniFile` (3.2.3). Many users will find that all or most of what was set in the old `~/.gaprc` file can now be done via the user preferences in a `gap.ini` file. If you had code for new functions or abbreviations in your old `~/.gaprc` file or you were reading additional files, then move this into the file `gaprc` (without the leading dot, same name for all operating systems) in the directory `GAPInfo.UserGapRoot`.

Chapter 78

Method Selection

This chapter explains how GAP decides which function to call for which types of objects. It assumes that you have read the chapters about objects (Chapter 12) and types (Chapter 13).

An *operation* is a special GAP function that bundles a set of functions, its *methods*.

All methods of an operation compute the same result. But each method is installed for specific types of arguments.

If an operation is called with a tuple of arguments, one of the applicable methods is selected and called.

Special cases of methods are partial methods, immediate methods, and logical implications.

78.1 Operations and Methods

Operations are functions in the category `IsOperation` (5.4.2).

So on the one hand, *operations* are GAP functions, that is, they can be applied to arguments and return a result or cause a side-effect.

On the other hand, operations are more. Namely, an operation corresponds to a set of GAP functions, called the *methods* of the operation.

Each call of an operation causes a suitable method to be selected and then called. The choice of which method to select is made according to the types of the arguments, the underlying mechanism is described in the following sections.

Examples of operations are the binary infix operators =, + etc., and `PrintObj` (6.3.5) is the operation that is called for each argument of `Print` (6.3.4).

Also all attributes and properties are operations. Each attribute has a special method which is called if the attribute value is already stored; this method of course simply returns this value.

The setter of an attribute is called automatically if an attribute value has been computed. Attribute setters are operations, too. They have a default method that ignores the request to store the value. Depending on the type of the object, there may be another method to store the value in a suitable way, and then set the attribute tester for the object to `true`.

78.2 Method Installation

In order to describe what it means to select a method of an operation, we must describe how the methods are connected to their operations.

For attributes and properties there is `InstallImmediateMethod` (78.6.1).

For declaring that a filter is implied by other filters there is `InstallTrueMethod` (78.7.1).

78.2.1 InstallMethod

▷ `InstallMethod(`*`opr[, info][, famp], args-filts[, val], method`*`)` (function)

installs a function method *`method`* for the operation *`opr`*; *`args-filts`* should be a list of requirements for the arguments, each entry being a filter; if supplied *`info`* should be a short but informative string that describes for what situation the method is installed, *`famp`* should be a function to be applied to the families of the arguments, and *`val`* should be an integer that measures the priority of the method.

The default values for *`info`*, *`famp`*, and *`val`* are the empty string, the function `ReturnTrue` (5.3.1), and the integer zero, respectively.

The exact meaning of the arguments *`famp`*, *`args-filts`*, and *`val`* is explained in Section 78.3.

`opr` expects its methods to require certain filters for their arguments. For example, the argument of a method for the operation `Zero` (31.10.3) must be in the category `IsAdditiveElementWithZero` (31.14.5). It is not possible to use `InstallMethod` to install a method for which the entries of *`args-filts`* do not imply the respective requirements of the operation *`opr`*. If one wants to override this restriction, one has to use `InstallOtherMethod` (78.2.2) instead.

78.2.2 InstallOtherMethod

▷ `InstallOtherMethod(`*`opr[, info][, famp], args-filts[, val], method`*`)` (function)

installs a function method *`method`* for the operation *`opr`*, in the same way as for `InstallMethod` (78.2.1), but without the restriction that the number of arguments must match a declaration of *`opr`* and without the restriction that *`args-filts`* imply the respective requirements of the operation *`opr`*.

78.3 Applicable Methods and Method Selection

A method installed as above is *applicable* for an arguments tuple if the following conditions are satisfied.

The number of arguments equals the length of the list *`args-filts`*, the *i*-th argument lies in the filter *`args-filts`*$[i]$, and *`famp`* returns `true` when applied to the families of the arguments. The maximal number of arguments supported for methods is six, one gets an error message if one tries to install a method with at least seven arguments.

So *`args-filt`* describes conditions for each argument, and *`famp`* describes a relation between the arguments.

For unary operations such as attributes and properties, there is no such relation to postulate, *`famp`* is `ReturnTrue` (5.3.1) for these operations, a function that always returns `true`. For binary operations, the usual value of *`famp`* is `IsIdenticalObj` (12.5.1), which means that both arguments must lie in the same family.

Note that any properties which occur among the filters in the filter list will *not* be tested by the method selection if they are not yet known. (More exact: if *`prop`* is a property then the filter implicitly uses not *`prop`* but `Has`*`prop`* `and` *`prop`*.) If this is desired you must explicitly enforce a test (see section 78.5) below.

If no method is applicable, the error message “no method found” is signaled.

Otherwise, the applicable method with highest *rank* is selected and then called. This rank is given by the sum of the ranks of the filters in the list *args-filt*, *including involved filters*, plus the number *val* used in the call of InstallMethod (78.2.1). So the argument *val* can be used to raise the priority of a method relative to other methods for *opr*.

Note that from the applicable methods, an efficient one shall be selected. This is a method that needs only little time and storage for the computations.

It seems to be impossible for GAP to select an optimal method in all cases. The present ranking of methods is based on the assumption that a method installed for a special situation shall be preferred to a method installed for a more general situation.

For example, a method for computing a Sylow subgroup of a nilpotent group is expected to be more efficient than a method for arbitrary groups. So the more specific method will be selected if GAP knows that the group given as argument is nilpotent.

Of course there is no obvious way to decide between the efficiency of incommensurable methods. For example, take an operation with one method for permutation groups, another method for nilpotent groups, but no method for nilpotent permutation groups, and call this operation with a permutation group known to be nilpotent.

78.4 Partial Methods

78.4.1 TryNextMethod

▷ TryNextMethod() (function)

After a method has been selected and called, the method may recognize that it cannot compute the desired result, and give up by calling TryNextMethod().

In effect, the execution of the method is terminated, and the method selection calls the next method that is applicable w.r.t. the original arguments. In other words, the applicable method is called that is subsequent to the one that called TryNextMethod, according to decreasing rank of the methods.

For example, since every finite group of odd order is solvable, one may install a method for the property IsSolvableGroup (39.15.6) that checks whether the size of the argument is an odd integer, returns true if so, and gives up otherwise.

Care is needed if a partial method might modify the type of one of its arguments, for example by computing an attribute or property. If this happens, and the type has really changed, then the method should not exit using TryNextMethod() but should call the operation again, as the new information in the type may cause some methods previously judged inapplicable to be applicable. For example, if the above method for IsSolvableGroup (39.15.6) actually computes the size, (rather than just examining a stored size), then it must take care to check whether the type of the group has changed.

78.5 Redispatching

As mentioned above the method selection will not test unknown properties. In situations, in which algorithms are only known (or implemented) under certain conditions, however such a test might be actually desired.

One way to achieve this would be to install the method under weaker conditions and explicitly test the properties first, exiting via TryNextMethod (78.4.1) if some of them are not fulfilled. A problem

of this approach however is that such methods then automatically are ranked lower and that the code does not look nice.

A much better way is to use redispatching: Before deciding that no method has been found one tests these properties and if they turn out to be true the method selection is started anew (and will then find a method).

This can be achieved via the following function:

78.5.1 RedispatchOnCondition

▷ `RedispatchOnCondition(`*`oper, fampred, reqs, cond, val`*`)` (function)

This function installs a method for the operation *oper* under the conditions *fampred* and *reqs* which has absolute value *val*; that is, the value of the filters *reqs* is disregarded. *cond* is a list of filters. If not all the values of properties involved in these filters are already known for actual arguments of the method, they are explicitly tested and if they are fulfilled *and* stored after this test, the operation is dispatched again. Otherwise the method exits with `TryNextMethod` (78.4.1). This can be used to enforce tests like `IsFinite` (30.4.2) in situations when all existing methods require this property. The list *cond* may have unbound entries in which case the corresponding argument is ignored for further tests.

78.6 Immediate Methods

Usually a method is called only if its operation has been called and if this method has been selected, see `InstallMethod` (78.2.1).

For attributes and properties, one can install also *immediate methods*.

78.6.1 InstallImmediateMethod

▷ `InstallImmediateMethod(`*`opr[, info], filter, rank, method`*`)` (function)

`InstallImmediateMethod` installs *method* as an immediate method for *opr*, which must be an attribute or a property, with requirement *filter* and rank *rank*. The rank must be an integer value that measures the priority of *method* among the immediate methods for *opr*. If supplied, *info* should be a short but informative string that describes the situation in which the method is called.

An immediate method is called automatically as soon as the object lies in *filter*, provided that the value is not yet known. Afterwards the attribute setter is called in order to store the value, unless the method exits via `TryNextMethod` (78.4.1).

Note the difference to `InstallMethod` (78.2.1) that no family predicate occurs because *opr* expects only one argument, and that *filter* is not a list of requirements but the argument requirement itself.

Immediate methods are thought of as a possibility for objects to gain useful knowledge. They must not be used to force the storing of "defining information" in an object. In other words, GAP should work even if all immediate methods are completely disabled. Therefore, the call to `InstallImmediateMethod` installs *method* also as an ordinary method for *opr* with requirement *filter*.

Note that in such a case GAP executes a computation for which it was not explicitly asked by the user. So one should install only those methods as immediate methods that are *extremely cheap*. To

emphasize this, immediate methods are also called *zero cost methods*. The time for their execution should really be approximately zero.

For example, the size of a permutation group can be computed very cheaply if a stabilizer chain of the group is known. So it is reasonable to install an immediate method for `Size` (30.4.6) with requirement `IsGroup and Tester( stab )`, where *stab* is the attribute corresponding to the stabilizer chain.

Another example would be the implementation of the conclusion that every finite group of prime power order is nilpotent. This could be done by installing an immediate method for the attribute `IsNilpotentGroup` (39.15.3) with requirement `IsGroup and Tester( Size )`. This method would then check whether the size is a finite prime power, return `true` in this case and otherwise call `TryNextMethod` (78.4.1). But this requires factoring of an integer, which cannot be guaranteed to be very cheap, so one should not install this method as an immediate method.

78.7 Logical Implications

78.7.1 InstallTrueMethod

▷ `InstallTrueMethod(`*newfil, filt*`)` (function)

It may happen that a filter *newfil* shall be implied by another filter *filt*, which is usually a meet of other properties, or the meet of some properties and some categories. Such a logical implication can be installed as an "immediate method" for *newfil* that requires *filt* and that always returns `true`. (This should not be mixed up with the methods installed via `InstallImmediateMethod` (78.6.1), which have to be called at runtime for the actual objects.)

`InstallTrueMethod` has the effect that *newfil* becomes an implied filter of *filt*, see 13.2.

For example, each cyclic group is abelian, each finite vector space is finite dimensional, and each division ring is integral. The first of these implications is installed as follows.

Example

```
InstallTrueMethod( IsCommutative, IsGroup and IsCyclic );
```

Contrary to the immediate methods installed with `InstallImmediateMethod` (78.6.1), logical implications cannot be switched off. This means that after the above implication has been installed, one can rely on the fact that every object in the filter `IsGroup and IsCyclic` will also be in the filter `IsCommutative` (35.4.9).

78.8 Operations and Mathematical Terms

Usually an operation stands for a mathematical concept, and the name of the operation describes this uniquely. Examples are the property `IsFinite` (30.4.2) and the attribute `Size` (30.4.6). But there are cases where the same mathematical term is used to denote different concepts, for example `Degree` is defined for polynomials, group characters, and permutation actions, and `Rank` is defined for matrices, free modules, p-groups, and transitive permutation actions.

It is in principle possible to install methods for the operation `Rank` that are applicable to the different types of arguments, corresponding to the different contexts. But this is not the approach taken in the GAP library. Instead there are operations such as `RankMat` (24.7.1) for matrices and

`DegreeOfCharacter` (72.8.4) (in fact these are attributes) which are installed as methods of the "ambiguous" operations `Rank` and `Degree`.

The idea is to distinguish between on the one hand different ways to compute the same thing (e.g. different methods for \= (31.11.1), `Size` (30.4.6), etc.), and on the other hand genuinely different things (such as the degree of a polynomial and a permutation action).

The former is the basic purpose of operations and attributes. The latter is provided as a user convenience where mathematical usage forces it on us *and* where no conflicts arise. In programming the library, we use the underlying mathematically precise operations or attributes, such as `RankMat` (24.7.1) and `RankOperation`. These should be attributes if appropriate, and the only role of the operation `Rank` is to decide which attribute the user meant. That way, stored information is stored with "full mathematical precision" and is less likely to be retrieved for a wrong purpose later.

One word about possible conflicts. A typical example is the mathematical term "centre", which is defined as $\{x \in M | a * x = x * a \forall a \in M\}$ for a magma M, and as $\{x \in L | l * x = 0 \forall l \in L\}$ for a Lie algebra L. Here it is *not* possible to introduce an operation `Centre` (35.4.5) that delegates to attributes `CentreOfMagma` and `CentreOfLieAlgebra`, depending on the type of the argument. This is because any Lie algebra in GAP is also a magma, so both `CentreOfMagma` and `CentreOfLieAlgebra` would be defined for a Lie algebra, with different meaning if the characteristic is two. So we cannot achieve that one operation in GAP corresponds to the mathematical term "centre".

"Ambiguous" operations such as `Rank` are declared in the library file `lib/overload.g`.

Chapter 79

Creating New Objects

This chapter is divided into three parts.

In the first part, it is explained how to create filters (see 79.1, 79.2, 79.3, 79.4), operations (see 79.5), families (see 79.7), types (see 79.8), and objects with given type (see 79.9).

In the second part, first a few small examples are given, for dealing with the usual cases of component objects (see 79.10) and positional objects (see 79.11), and for the implementation of new kinds of lists (see 79.12 and 79.15). Finally, the external representation of objects is introduced (see 79.16), as a tool for representation independent access to an object.

The third part deals with some rules concerning the organization of the GAP library; namely, some commands for creating global variables are explained (see 79.18) that correspond to the ones discussed in the first part of the chapter, and the idea of distinguishing declaration and implementation part of GAP packages is outlined (see 79.19).

See also Chapter 81 for examples how the functions from the first part are used, and why it is useful to have a declaration part and an implementation part.

79.1 Creating Categories

79.1.1 NewCategory

▷ `NewCategory(`*`name, super[, rank]`*`)` (function)

`NewCategory` returns a new category *`cat`* that has the name *`name`* and is contained in the filter *`super`*, see 13.2. This means that every object in *`cat`* lies automatically also in *`super`*. We say also that *`super`* is an implied filter of *`cat`*.

For example, if one wants to create a category of group elements then *`super`* should be `IsMultiplicativeElementWithInverse` (31.14.13) or a subcategory of it. If no specific supercategory of *`cat`* is known, *`super`* may be `IsObject` (12.1.1).

The optional third argument *`rank`* denotes the incremental rank (see 13.2) of *`cat`*, the default value is 1.

79.1.2 CategoryFamily

▷ `CategoryFamily(`*`cat`*`)` (function)

For a category `cat`, `CategoryFamily` returns the *family category* of `cat`. This is a category in which all families lie that know from their creation that all their elements are in the category `cat`, see 79.7.

For example, a family of associative words is in the category `CategoryFamily( IsAssocWord )`, and one can distinguish such a family from others by this category. So it is possible to install methods for operations that require one argument to be a family of associative words.

`CategoryFamily` is quite technical, and in fact of minor importance.

See also `CategoryCollections` (30.2.4).

79.2 Creating Representations

79.2.1 NewRepresentation

▷ `NewRepresentation(`*`name, super, slots[, req]`*`)` (function)

`NewRepresentation` returns a new representation *`rep`* that has the name *`name`* and is a subrepresentation of the representation *`super`*. This means that every object in *`rep`* lies automatically also in *`super`*. We say also that *`super`* is an implied filter of *`rep`*.

Each representation in GAP is a subrepresentation of exactly one of the four representations `IsInternalRep`, `IsDataObjectRep`, `IsComponentObjectRep`, `IsPositionalObjectRep`. The data describing objects in the former two can be accessed only via GAP kernel functions, the data describing objects in the latter two is accessible also in library functions, see 79.10 and 79.11 for the details.

The third argument *`slots`* is a list either of integers or of strings. In the former case, *`rep`* must be `IsPositionalObjectRep` or a subrepresentation of it, and *`slots`* tells what positions of the objects in the representation *`rep`* may be bound. In the latter case, *`rep`* must be `IsComponentObjectRep` or a subrepresentation of, and *`slots`* lists the admissible names of components that objects in the representation *`rep`* may have. The admissible positions resp. component names of *`super`* need not be be listed in *`slots`*.

The incremental rank (see 13.2) of *`rep`* is 1.

Note that for objects in the representation *`rep`*, of course some of the component names and positions reserved via *`slots`* may be unbound.

Examples for the use of `NewRepresentation` can be found in 79.10, 79.11, and also in 81.3.

79.3 Creating Attributes and Properties

Each method that is installed for an attribute or a property via `InstallMethod` (78.2.1) must require exactly one argument, and this must lie in the filter *`filter`* that was entered as second argument of `NewAttribute` (79.3.1) resp. `NewProperty` (79.3.2).

As for any operation (see 79.5), for attributes and properties one can install a method taking an argument that does not lie in *`filt`* via `InstallOtherMethod` (78.2.2), or a method for more than one argument; in the latter case, clearly the result value is *not* stored in any of the arguments.

79.3.1 NewAttribute

▷ `NewAttribute(`*`name, filter[, "mutable"][, rank]`*`)` (function)

NewAttribute returns a new attribute getter with name *name* that is applicable to objects with the property *filter*.

Contrary to the situation with categories and representations, the tester of the new attribute does *not* imply *filter*. This is exactly because of the possibility to install methods that do not require *filter*.

For example, the attribute Size (30.4.6) was created with second argument a list or a collection, but there is also a method for Size (30.4.6) that is applicable to a character table, which is neither a list nor a collection.

If the optional third argument is given then there are two possibilities. Either it is an integer *rank*, then the attribute tester has this incremental rank (see 13.2). Or it is the string "mutable", then the values of the attribute shall be mutable; more precisely, when a value of such a mutable attribute is set then this value itself is stored, not an immutable copy of it. (So it is the user's responsibility to set an object that is in fact mutable.) This is useful for an attribute whose value is some partial information that may be completed later. For example, there is an attribute ComputedSylowSubgroups for the list holding those Sylow subgroups of a group that have been computed already by the function SylowSubgroup (39.13.1), and this list is mutable because one may want to enter groups into it as they are computed.

If no third argument is given then the rank of the tester is 1.

Each method for the new attribute that does *not* require its argument to lie in *filter* must be installed using InstallOtherMethod (78.2.2).

79.3.2 NewProperty

▷ NewProperty(*name, filter[, rank]*) (function)

NewProperty returns a new property *prop* with name *name* (see also 13.7). The filter *filter* describes the involved filters of *prop*. As in the case of attributes, *filter* is not implied by *prop*.

The optional third argument *rank* denotes the incremental rank (see 13.2) of the property *prop* itself, i.e. *not* of its tester; the default value is 1.

79.4 Creating Other Filters

In order to change the value of *filt* for an object *obj*, one can use logical implications (see 78.7) or SetFilterObj (79.4.2), ResetFilterObj (79.4.3).

79.4.1 NewFilter

▷ NewFilter(*name[, rank]*) (function)

NewFilter returns a simple filter with name *name* (see 13.8). The optional second argument *rank* denotes the incremental rank (see 13.2) of the filter, the default value is 1.

The default value of the new simple filter for each object is false.

79.4.2 SetFilterObj

▷ SetFilterObj(*obj, filter*) (function)

SetFilterObj sets the value of *filter* (and of all filters implied by *filter*) for *obj* to true,

79.4.3 ResetFilterObj

▷ ResetFilterObj(*obj, filter*) (function)

ResetFilterObj sets the value of *filter* for *obj* to false. (Implied filters of *filt* are not touched. This might create inconsistent situations if applied carelessly).

79.5 Creating Operations

79.5.1 NewOperation

▷ NewOperation(*name, args-filts*) (function)

NewOperation returns an operation *opr* with name *name*. The list *args-filts* describes requirements about the arguments of *opr*, namely the number of arguments must be equal to the length of *args-filts*, and the i-th argument must lie in the filter *args-filts*$[i]$.

Each method that is installed for *opr* via InstallMethod (78.2.1) must require that the i-th argument lies in the filter *args-filts*$[i]$.

One can install methods for other arguments tuples via InstallOtherMethod (78.2.2), this way it is also possible to install methods for a different number of arguments than the length of *args-filts*.

79.6 Creating Constructors

79.6.1 NewConstructor

▷ NewConstructor(*name, args-filts*) (function)

NewConstructor returns a constructor *cons* with name *name*. The list *args-filts* describes requirements about the arguments of *constr*, namely the number of arguments must be equal to the length of *args-filts*, and the i-th argument must lie in the filter *args-filts*$[i]$. Additionally a constructor expects the first argument to be a filter to then select a method to construct an object.

Each method that is installed for *cons* via InstallMethod (78.2.1) must require that the i-th argument lies in the filter *args-filts*$[i]$.

One can install methods for other arguments tuples via InstallOtherMethod (78.2.2), this way it is also possible to install methods for a different number of arguments than the length of *args-filts*.

79.7 Creating Families

Families are probably the least obvious part of the GAP type system, so some remarks about the role of families are necessary. When one uses GAP as it is, one will (better: should) not meet families at all. The two situations where families come into play are the following.

First, since families are used to describe relations between arguments of operations in the method selection mechanism (see Chapter 78, and also Chapter 13), one has to prescribe such a relation in each method installation (see 78.2); usual relations are ReturnTrue (5.3.1) (which means that any

relation of the actual arguments is admissible), IsIdenticalObj (12.5.1) (which means that there are two arguments that lie in the same family), and IsCollsElms (which means that there are two arguments, the first being a collection of elements that lie in the same family as the second argument).

Second –and this is the more complicated situation– whenever one creates a new kind of objects, one has to decide what its family shall be. If the new object shall be equal to existing objects, for example if it is just represented in a different way, there is no choice: The new object must lie in the same family as all objects that shall be equal to it. So only if the new object is different (w.r.t. the equality "=") from all other GAP objects, we are likely to create a new family for it. Note that enlarging an existing family by such new objects may be problematic because of implications that have been installed for all objects of the family in question. The choice of families depends on the applications one has in mind. For example, if the new objects in question are not likely to be arguments of operations for which family relations are relevant (for example binary arithmetic operations), one could create one family for all such objects, and regard it as "the family of all those GAP objects that would in fact not need a family". On the other extreme, if one wants to create domains of the new objects then one has to choose the family in such a way that all intended elements of a domain do in fact lie in the same family. (Remember that a domain is a collection, see Chapter 12.4, and that a collection consists of elements in the same family, see Chapter 30 and Section 13.1.)

Let us look at an example. Suppose that no permutations are available in GAP, and that we want to implement permutations. Clearly we want to support permutation groups, but it is not a priori clear how to distribute the new permutations into families. We can put all permutations into one family; this is how in fact permutations are implemented in GAP. But it would also be possible to put all permutations of a given degree into a family of their own; this would for example mean that for each degree, there would be distinguished trivial permutations, and that the stabilizer of the point 5 in the symmetric group on the points 1, 2, ..., 5 is not regarded as equal to the symmetric group on 1, 2, 3, 4. Note that the latter approach would have the advantage that it is no problem to construct permutations and permutation groups acting on arbitrary (finite) sets, for example by constructing first the symmetric group on the set and then generating any desired permutation group as a subgroup of this symmetric group.

So one aspect concerning a reasonable choice of families is to make the families large enough for being able to form interesting domains of elements in the family. But on the other hand, it is useful to choose the families small enough for admitting meaningful relations between objects. For example, the elements of different free groups in GAP lie in different families; the multiplication of free group elements is installed only for the case that the two operands lie in the same family, with the effect that one cannot erroneously form the product of elements from different free groups. In this case, families appear as a tool for providing useful restrictions.

As another example, note that an element and a collection containing this element never lie in the same family, by the general implementation of collections; namely, the family of a collection of elements in the family *Fam* is the collections family of *Fam* (see CollectionsFamily (30.2.1)). This means that for a collection, we need not (because we cannot) decide about its family.

A few functions in GAP return families, see CollectionsFamily (30.2.1) and ElementsFamily (30.2.3).

79.7.1 NewFamily

▷ NewFamily(*name[, req[, imp[, famfilter]]]*) (function)

NewFamily returns a new family *fam* with name *name*. The argument *req*, if present, is a filter

of which *fam* shall be a subset. If one tries to create an object in *fam* that does not lie in the filter *req*, an error message is printed. Also the argument *imp*, if present, is a filter of which *fam* shall be a subset. Any object that is created in the family *fam* will lie automatically in the filter *imp*.

The filter *famfilter*, if given, specifies a filter that will hold for the family *fam* (not for objects in *fam*).

Families are always represented as component objects (see 79.10). This means that components can be used to store and access useful information about the family.

79.8 Creating Types

79.8.1 NewType

▷ NewType(*family, filter[, data]*) (function)

NewType returns the type given by the family *family* and the filter *filter*. The optional third argument *data* is any object that denotes defining data of the desired type.

For examples where NewType is used, see 79.10, 79.11, and the example in Chapter 81.

79.9 Creating Objects

79.9.1 Objectify

▷ Objectify(*type, data*) (function)

New objects are created by Objectify. *data* is a list or a record, and *type* is the type that the desired object shall have. Objectify turns *data* into an object with type *type*. That is, *data* is changed, and afterwards it will not be a list or a record unless *type* is of type list resp. record.

If *data* is a list then Objectify turns it into a positional object, if *data* is a record then Objectify turns it into a component object (for examples, see 79.10 and 79.11).

Objectify does also return the object that it made out of *data*.

For examples where Objectify is used, see 79.10, 79.11, and the example in Chapter 81.

79.9.2 ObjectifyWithAttributes

▷ ObjectifyWithAttributes(*obj, type, attr1, val1, attr2, val2, ...*) (function)

Attribute assignments will change the type of an object. If you create many objects, code of the form

Example

```
o:=Objectify(type,rec());
SetMyAttribute(o,value);
```

will take a lot of time for type changes. You can avoid this by setting the attributes immediately while the object is created, as follows. ObjectifyWithAttributes changes the type of object *obj* to type *type* and sets attribute *attr1* to *val1*, sets attribute *attr2* to *val2* and so forth.

If the filter list of *type* includes that these attributes are set (and the properties also include values of the properties) and if no special setter methods are installed for any of the involved attributes then they are set simultaneously without type changes. This can produce a substantial speedup.

If the conditions of the last sentence are not fulfilled, an ordinary `Objectify` (79.9.1) with subsequent setter calls for the attributes is performed instead.

79.10 Component Objects

A *component object* is an object in the representation `IsComponentObjectRep` or a subrepresentation of it. Such an object *cobj* is built from subobjects that can be accessed via *cobj*`!.`*name*, similar to components of a record. Also analogously to records, values can be assigned to components of *cobj* via *cobj*`!.`*name*`:=` *val*. For the creation of component objects, see 79.9. One must be *very careful* when using the `!.` operator, in order to interpret the component in the right way, and even more careful when using the assignment to components using `!.`, in order to keep the information stored in *cobj* consistent.

First of all, in the access or assignment to a component as shown above, *name* must be among the admissible component names for the representation of *cobj*, see 79.2. Second, preferably only few low level functions should use `!.`, whereas this operator should not occur in "user interactions".

Note that even if *cobj* claims that it is immutable, i.e., if *cobj* is not in the category `IsMutable` (12.6.2), access and assignment via `!.` and `!.:=` work. This is necessary for being able to store newly discovered information in immutable objects.

The following example shows the implementation of an iterator (see 30.8) for the domain of integers, which is represented as component object. See 79.11 for an implementation using positional objects. (In practice, such an iterator can be implemented more elegantly using `IteratorByFunctions` (30.8.8), see 79.14.)

The used succession of integers is $0,1,-1,2,-2,3,-3,\ldots$, that is, $a_n = n/2$ if n is even, and $a_n = (1-n)/2$ otherwise.

Example

```
IsIntegersIteratorCompRep := NewRepresentation( "IsIntegersIteratorRep",
    IsComponentObjectRep, [ "counter" ] );
```

The above command creates a new representation (see `NewRepresentation` (79.2.1)) `IsIntegersIteratorCompRep`, as a subrepresentation of `IsComponentObjectRep`, and with one admissible component `counter`. So no other components than `counter` will be needed.

Example

```
InstallMethod( Iterator,
    "method for 'Integers'",
    [ IsIntegers ],
    function( Integers )
    return Objectify( NewType( IteratorsFamily,
                                   IsIterator
                               and IsIntegersIteratorCompRep ),
                      rec( counter := 0 ) );
    end );
```

After the above method installation, one can already ask for `Iterator( Integers )`. Note that exactly the domain of integers is described by the filter `IsIntegers` (14.1.2).

By the call to NewType (79.8.1), the returned object lies in the family containing all iterators, which is IteratorsFamily, it lies in the category IsIterator (30.8.3) and in the representation IsIntegersIteratorCompRep; furthermore, it has the component counter with value 0.

What is missing now are methods for the two basic operations of iterators, namely IsDoneIterator (30.8.4) and NextIterator (30.8.5). The former must always return false, since there are infinitely many integers. The latter must return the next integer in the iteration, and update the information stored in the iterator, that is, increase the value of the component counter.

Example

```
InstallMethod( IsDoneIterator,
    "method for iterator of ‘Integers’",
    [ IsIterator and IsIntegersIteratorCompRep ],
    ReturnFalse );

InstallMethod( NextIterator,
    "method for iterator of ‘Integers’",
    [ IsIntegersIteratorCompRep ],
    function( iter )
    iter!.counter:= iter!.counter + 1;
    if iter!.counter mod 2 = 0 then
      return iter!.counter / 2;
    else
      return ( 1 - iter!.counter ) / 2;
    fi;
    end );
```

79.10.1 NamesOfComponents

▷ NamesOfComponents(*comobj*) (function)

For a component object *comobj*, NamesOfComponents returns a list of strings, which are the names of components currently bound in *comobj*.

For a record *comobj*, NamesOfComponents returns the result of RecNames (29.1.2).

79.11 Positional Objects

A *positional object* is an object in the representation IsPositionalObjectRep or a subrepresentation of it. Such an object *pobj* is built from subobjects that can be accessed via *pobj*![*pos*], similar to positions in a list. Also analogously to lists, values can be assigned to positions of *pobj* via *pobj*![*pos*]:= *val*. For the creation of positional objects, see 79.9.

One must be *very careful* when using the ![] operator, in order to interpret the position in the right way, and even more careful when using the assignment to positions using ![], in order to keep the information stored in *pobj* consistent.

First of all, in the access or assignment to a position as shown above, *pos* must be among the admissible positions for the representation of *pobj*, see 79.2. Second, preferably only few low level functions should use ![], whereas this operator should not occur in “user interactions”.

Note that even if *pobj* claims that it is immutable, i.e., if *pobj* is not in the category IsMutable (12.6.2), access and assignment via ![] work. This is necessary for being able to store newly discovered information in immutable objects.

The following example shows the implementation of an iterator (see 30.8) for the domain of integers, which is represented as positional object. See 79.10 for an implementation using component objects, and more details.

Example

```
IsIntegersIteratorPosRep := NewRepresentation( "IsIntegersIteratorRep",
    IsPositionalObjectRep, [ 1 ] );
```

The above command creates a new representation (see NewRepresentation (79.2.1)) IsIntegersIteratorPosRep, as a subrepresentation of IsComponentObjectRep, and with only the first position being admissible for storing data.

Example

```
InstallMethod( Iterator,
    "method for 'Integers'",
    [ IsIntegers ],
    function( Integers )
    return Objectify( NewType( IteratorsFamily,
                                   IsIterator
                               and IsIntegersIteratorRep ),
                      [ 0 ] );
    end );
```

After the above method installation, one can already ask for Iterator(Integers). Note that exactly the domain of integers is described by the filter IsIntegers (14.1.2).

By the call to NewType (79.8.1), the returned object lies in the family containing all iterators, which is IteratorsFamily, it lies in the category IsIterator (30.8.3) and in the representation IsIntegersIteratorPosRep; furthermore, the first position has value 0.

What is missing now are methods for the two basic operations of iterators, namely IsDoneIterator (30.8.4) and NextIterator (30.8.5). The former must always return false, since there are infinitely many integers. The latter must return the next integer in the iteration, and update the information stored in the iterator, that is, increase the value stored in the first position.

Example

```
InstallMethod( IsDoneIterator,
    "method for iterator of 'Integers'",
    [ IsIterator and IsIntegersIteratorPosRep ],
    ReturnFalse );

InstallMethod( NextIterator,
    "method for iterator of 'Integers'",
    [ IsIntegersIteratorPosRep ],
    function( iter )
    iter![1]:= iter![1] + 1;
    if iter![1] mod 2 = 0 then
      return iter![1] / 2;
    else
      return ( 1 - iter![1] ) / 2;
    fi;
    end );
```

It should be noted that one can of course install both the methods shown in Section 79.10 and 79.11. The call Iterator(Integers) will cause one of the methods to be selected, and for the

returned iterator, which will have one of the representations we constructed, the right NextIterator (30.8.5) method will be chosen.

79.12 Implementing New List Objects

This section gives some hints for the quite usual situation that one wants to implement new objects that are lists. More precisely, one either wants to deal with lists that have additional features, or one wants that some objects also behave as lists. An example can be found in 79.13.

A *list* in GAP is an object in the category IsList (21.1.1). Basic operations for lists are Length (21.17.5), \[\] (21.2.1), and IsBound\[\] (21.2.1) (see 21.2).

Note that the access to the position *pos* in the list *list* via *list*[*pos*] is handled by the call \[\](*list*, *pos*) to the operation \[\] (21.2.1). To explain the somewhat strange name \[\] of this operation, note that non-alphanumeric characters like [and] may occur in GAP variable names only if they are escaped by a \ character.

Analogously, the check IsBound(*list*[*pos*]) whether the position *pos* of the list *list* is bound is handled by the call IsBound\[\](*list*, *pos*) to the operation IsBound\[\] (21.2.1).

For mutable lists, also assignment to positions and unbinding of positions via the operations \[\]\:\= (21.2.1) and Unbind\[\] (21.2.1) are basic operations. The assignment *list*[*pos*]:= *val* is handled by the call \[\]\:\=(*list*, *pos*, *val*), and Unbind(*list*[*pos*]) is handled by the call Unbind\[\](*list*, *pos*).

All other operations for lists, e.g., Add (21.4.2), Append (21.4.5), Sum (21.20.26), are based on these operations. This means that it is sufficient to install methods for the new list objects only for the basic operations.

So if one wants to implement new list objects then one creates them as objects in the category IsList (21.1.1), and installs methods for Length (21.17.5), \[\] (21.2.1), and IsBound\[\] (21.2.1). If the new lists shall be mutable, one needs to install also methods for \[\]\:\= (21.2.1) and Unbind\[\] (21.2.1).

One application for this is the implementation of *enumerators* for domains. An enumerator for the domain D is a dense list whose entries are in bijection with the elements of D. If D is large then it is not useful to write down all elements. Instead one can implement such a bijection implicitly. This works also for infinite domains.

In this situation, one implements a new representation of the lists that are already available in GAP, in particular the family of such a list is the same as the family of the domain D.

But it is also possible to implement new kinds of lists that lie in new families, and thus are not equal to lists that were available in GAP before. An example for this is the implementation of matrices whose multiplication via "*" is the Lie product of matrices.

In this situation, it makes no sense to put the new matrices into the same family as the original matrices. Note that the product of two Lie matrices shall be defined but not the product of an ordinary matrix and a Lie matrix. So it is possible to have two lists that have the same entries but that are not equal w.r.t. "=" because they lie in different families.

79.13 Example – Constructing Enumerators

When dealing with countable sets, a usual task is to define enumerations, i.e., bijections to the positive integers. In GAP, this can be implemented via *enumerators* (see 21.23). These are lists containing the elements in a specified ordering, and the operations Position (21.16.1) and list access via \[\]

(21.2.1) define the desired bijection. For implementing such an enumerator, one mainly needs to install the appropriate functions for these operations.

A general setup for creating such lists is given by EnumeratorByFunctions (30.3.4).

If the set in question is a domain *D* for which a Size (30.4.6) method is available then all one has to do is to write down the functions for computing the *n*-th element of the list and for computing the position of a given GAP object in the list, to put them into the components ElementNumber and NumberElement of a record, and to call EnumeratorByFunctions (30.3.4) with the domain *D* and this record as arguments. For example, the following lines of code install an Enumerator (30.3.2) method for the case that *D* is the domain of rational integers. (Note that IsIntegers (14.1.2) is a filter that describes exactly the domain of rational integers.)

Example

```
InstallMethod( Enumerator,
    "for integers",
    [ IsIntegers ],
    Integers -> EnumeratorByFunctions( Integers, rec(
                    ElementNumber := function( e, n ) ... end,
                    NumberElement := function( e, x ) ... end ) ) );
```

The bodies of the functions have been omitted above; here is the code that is actually used in GAP. (The ordering coincides with that for the iterators for the domain of rational integers that have been discussed in 79.10 and 79.11.)

Example

```
gap> enum:= Enumerator( Integers );
<enumerator of Integers>
gap> Print( enum!.NumberElement, "\n" );
function ( e, x )
    local  pos;
    if not IsInt( x )  then
        return fail;
    elif 0 < x  then
        pos := 2 * x;
    else
        pos := -2 * x + 1;
    fi;
    return pos;
end
gap> Print( enum!.ElementNumber, "\n" );
function ( e, n )
    if n mod 2 = 0  then
        return n / 2;
    else
        return (1 - n) / 2;
    fi;
    return;
end
```

The situation becomes slightly more complicated if the set *S* in question is not a domain. This is because one must provide also at least a method for computing the length of the list, and because one has to determine the family in which it lies (see 79.9). The latter should usually not be a problem since

either S is nonempty and all its elements lie in the same family –in this case one takes the collections family of any element in S– or the family of the enumerator must be `ListsFamily`.

An example in the GAP library is an enumerator for the set of k-tuples over a finite set; the function is called `EnumeratorOfTuples` (16.2.9).

Example

```
gap> Print( EnumeratorOfTuples, "\n" );
function ( set, k )
    local  enum;
    if k = 0  then
        return Immutable( [ [  ] ] );
    elif IsEmpty( set )  then
        return Immutable( [  ] );
    fi;
    enum
     := EnumeratorByFunctions( CollectionsFamily( FamilyObj( set ) ),
       rec(
          ElementNumber := function ( enum, n )
                local  nn, t, i;
                nn := n - 1;
                t := [  ];
                for i  in [ 1 .. enum!.k ]  do
                    t[i] := RemInt( nn, Length( enum!.set ) ) + 1;
                    nn := QuoInt( nn, Length( enum!.set ) );
                od;
                if nn <> 0  then
                    Error( "<enum>[", n,
                     "] must have an assigned value" );
                fi;
                nn := enum!.set{Reversed( t )};
                MakeImmutable( nn );
                return nn;
            end,
          NumberElement := function ( enum, elm )
                local  n, i;
                if not IsList( elm )  then
                    return fail;
                fi;
                elm := List( elm, function ( x )
                        return Position( enum!.set, x );
                    end );
                if fail in elm or Length( elm ) <> enum!.k  then
                    return fail;
                fi;
                n := 0;
                for i  in [ 1 .. enum!.k ]  do
                    n := Length( enum!.set ) * n + elm[i] - 1;
                od;
                return n + 1;
            end,
          Length := function ( enum )
                return Length( enum!.set ) ^ enum!.k;
            end,
          PrintObj := function ( enum )
```

```
                Print( "EnumeratorOfTuples( ", enum!.set, ", ",
                 enum!.k, " )" );
                return;
            end,
          set := Set( set ),
          k := k ) );
    SetIsSSortedList( enum, true );
    return enum;
end
```

We see that the enumerator is a homogeneous list that stores individual functions `ElementNumber`, `NumberElement`, `Length`, and `PrintObj`; besides that, the data components S and k are contained.

79.14 Example – Constructing Iterators

Iterators are a kind of objects that is implemented for several collections in the GAP library and which might be interesting also in other cases, see 30.8. A general setup for implementing new iterators is provided by `IteratorByFunctions` (30.8.8).

All one has to do is to write down the functions for `NextIterator` (30.8.5), `IsDoneIterator` (30.8.4), and `ShallowCopy` (12.7.1), and to call `IteratorByFunctions` (30.8.8) with this record as argument. For example, the following lines of code install an `Iterator` (30.8.1) method for the case that the argument is the domain of rational integers.

(Note that `IsIntegers` (14.1.2) is a filter that describes exactly the domain of rational integers.)

Example

```
InstallMethod( Iterator,
    "for integers",
    [ IsIntegers ],
    Integers -> IteratorByFunctions( rec(
                    NextIterator:= function( iter ) ... end,
                    IsDoneIterator := ReturnFalse,
                    ShallowCopy := function( iter ) ... end ) ) );
```

The bodies of two of the functions have been omitted above; here is the code that is actually used in GAP. (The ordering coincides with that for the iterators for the domain of rational integers that have been discussed in 79.10 and 79.11.)

Example

```
gap> iter:= Iterator( Integers );
<iterator>
gap> Print( iter!.NextIterator, "\n" );
function ( iter )
    iter!.counter := iter!.counter + 1;
    if iter!.counter mod 2 = 0  then
        return iter!.counter / 2;
    else
        return (1 - iter!.counter) / 2;
    fi;
    return;
end
gap> Print( iter!.ShallowCopy, "\n" );
```

```
function ( iter )
    return rec(
        counter := iter!.counter );
end
```

Note that the ShallowCopy component of the record must be a function that does not return an iterator but a record that can be used as the argument of IteratorByFunctions (30.8.8) in order to create the desired shallow copy.

79.15 Arithmetic Issues in the Implementation of New Kinds of Lists

When designing a new kind of list objects in GAP, defining the arithmetic behaviour of these objects is an issue.

There are situations where arithmetic operations of list objects are unimportant in the sense that adding two such lists need not be represented in a special way. In such cases it might be useful either to support no arithmetics at all for the new lists, or to enable the default arithmetic methods. The former can be achieved by not setting the filters IsGeneralizedRowVector (21.12.1) and IsMultiplicativeGeneralizedRowVector (21.12.2) in the types of the lists, the latter can be achieved by setting the filter IsListDefault (21.12.3). (for details, see 21.12). An example for "wrapped lists" with default behaviour are vector space bases; they are lists with additional properties concerning the computation of coefficients, but arithmetic properties are not important. So it is no loss to enable the default methods for these lists.

However, often the arithmetic behaviour of new list objects is important, and one wants to keep these lists away from default methods for addition, multiplication etc. For example, the sum and the product of (compatible) block matrices shall be represented as a block matrix, so the default methods for sum and product of matrices shall not be applicable, although the results will be equal to those of the default methods in the sense that their entries at corresponding positions are equal.

So one does not set the filter IsListDefault (21.12.3) in such cases, and thus one can implement one's own methods for arithmetic operations. (Of course "can" means on the other hand that one *must* implement such methods if one is interested in arithmetics of the new lists.)

The specific binary arithmetic methods for the new lists will usually cover the case that both arguments are of the new kind, and perhaps also the interaction between a list of the new kind and certain other kinds of lists may be handled if this appears to be useful.

For the last situation, interaction between a new kind of lists and other kinds of lists, GAP provides already a setup. Namely, there are the categories IsGeneralizedRowVector (21.12.1) and IsMultiplicativeGeneralizedRowVector (21.12.2), which are concerned with the additive and the multiplicative behaviour, respectively, of lists. For lists in these filters, the structure of the results of arithmetic operations is prescribed (see 21.13 and 21.14).

For example, if one implements block matrices in IsMultiplicativeGeneralizedRowVector (21.12.2) then automatically the product of such a block matrix and a (plain) list of such block matrices will be defined as the obvious list of matrix products, and a default method for plain lists will handle this multiplication. (Note that this method will rely on a method for computing the product of the block matrices, and of course no default method is available for that.) Conversely, if the block matrices are not in IsMultiplicativeGeneralizedRowVector (21.12.2) then the product of a block matrix and a (plain) list of block matrices is not defined. (There is no default method for it, and one can define the result and provide a method for computing it.)

Thus if one decides to set the filters `IsGeneralizedRowVector` (21.12.1) and `IsMultiplicativeGeneralizedRowVector` (21.12.2) for the new lists, on the one hand one loses freedom in defining arithmetic behaviour, but on the other hand one gains several default methods for a more or less natural behaviour.

If a list in the filter `IsGeneralizedRowVector` (21.12.1) (`IsMultiplicativeGeneralizedRowVector` (21.12.2)) lies in `IsAttributeStoringRep`, the values of additive (multiplicative) nesting depth is stored in the list and need not be calculated for each arithmetic operation. One can then store the value(s) already upon creation of the lists, with the effect that the default arithmetic operations will access elements of these lists only if this is unavoidable. For example, the sum of two plain lists of "wrapped matrices" with stored nesting depths are computed via the method for adding two such wrapped lists, and without accessing any of their rows (which might be expensive). In this sense, the wrapped lists are treated as black boxes.

79.16 External Representation

An operation is defined for elements rather than for objects in the sense that if the arguments are replaced by objects that are equal to the old arguments w.r.t. the equivalence relation "=" then the result must be equal to the old result w.r.t. "=".

But the implementation of many methods is representation dependent in the sense that certain representation dependent subobjects are accessed.

For example, a method that implements the addition of univariate polynomials may access coefficients lists of its arguments only if they are really stored, while in the case of sparsely represented polynomials a different approach is needed.

In spite of this, for many operations one does not want to write an own method for each possible representations of each argument, for example because none of the methods could in fact take advantage of the actually given representations of the objects. Another reason could be that one wants to install first a representation independent method, and then add specific methods as they are needed to gain more efficiency, by really exploiting the fact that the arguments have certain representations.

For the purpose of admitting representation independent code, one can define an *external representation* of objects in a given family, install methods to compute this external representation for each representation of the objects, and then use this external representation of the objects whenever they occur.

We cannot provide conversion functions that allow us to first convert any object in question to one particular "standard representation", and then access the data in the way defined for this representation, simply because it may be impossible to choose such a "standard representation" uniformly for all objects in the given family.

So the aim of an external representation of an object `obj` is a different one, namely to describe the data from which `obj` is composed. In particular, the external representation of `obj` is *not* one possible ("standard") representation of `obj`, in fact the external representation of `obj` is in general different from `obj` w.r.t. "=", first of all because the external representation of `obj` does in general not lie in the same family as `obj`.

For example the external representation of a rational function is a list of length two or three, the first entry being the zero coefficient, the second being a list describing the coefficients and monomials of the numerator, and the third, if bound, being a list describing the coefficients and monomials of the denominator. In particular, the external representation of a polynomial is a list and not a polynomial.

The other way round, the external representation of `obj` encodes `obj` in such a way that from

this data and the family of *obj*, one can create an object that is equal to *obj*. Usually the external representation of an object is a list or a record.

Although the external representation of *obj* is by definition independent of the actually available representations for *obj*, it is usual that a representation of *obj* exists for which the computation of the external representation is obtained by just "unpacking" *obj*, in the sense that the desired data is stored in a component or a position of *obj*, if *obj* is a component object (see 79.10) or a positional object (see 79.11).

To implement an external representation means to install methods for the following two operations.

79.16.1 ExtRepOfObj

▷ `ExtRepOfObj(`*obj*`)` (function)
▷ `ObjByExtRep(`*fam*`, `*data*`)` (function)

`ExtRepOfObj` returns the external representation of its argument, and `ObjByExtRep` returns an object in the family *fam* that has external representation *data*.

Of course, `ObjByExtRep( FamilyObj(` *obj* `), ExtRepOfObj(` *obj* `) )` must be equal to *obj* w.r.t. the operation `\=` (31.11.1). But it is *not* required that equal objects have equal external representations.

Note that if one defines a new representation of objects for which an external representation does already exist then one *must* install a method to compute this external representation for the objects in the new representation.

79.17 Mutability and Copying

Any GAP object is either mutable or immutable. This can be tested with the function `IsMutable` (12.6.2). The intended meaning of (im)mutability is a mathematical one: an immutable object should never change in such a way that it represents a different Element. Objects *may* change in other ways, for instance to store more information, or represent an element in a different way.

Immutability is enforced in different ways for built-in objects (like records, or lists) and for external objects (made using `Objectify` (79.9.1)).

For built-in objects which are immutable, the kernel will prevent you from changing them. Thus

Example

```
gap> l := [1,2,4];
[ 1, 2, 4 ]
gap> MakeImmutable(l);
[ 1, 2, 4 ]
gap> l[3] := 5;
Error, Lists Assignment: <list> must be a mutable list
```

For external objects, the situation is different. An external object which claims to be immutable (i.e. its type does not contain `IsMutable` (12.6.2)) should not admit any methods which change the element it represents. The kernel does *not* prevent the use of `!.` and `![` to change the underlying data structure. This is used for instance by the code that stores attribute values for reuse. In general, these `!` operations should only be used in methods which depend on the representation of the object.

Furthermore, we would *not* recommend users to install methods which depend on the representations of objects created by the library or by GAP packages, as there is certainly no guarantee of the representations being the same in future versions of GAP.

Here we see an immutable object (the group S_4), in which we improperly install a new component.

Example

```
gap> g := SymmetricGroup(IsPermGroup,4);
Sym( [ 1 .. 4 ] )
gap> IsMutable(g);
false
gap> NamesOfComponents(g);
[ "Size", "NrMovedPoints", "MovedPoints",
  "GeneratorsOfMagmaWithInverses" ]
gap> g!.silly := "rubbish";
"rubbish"
gap> NamesOfComponents(g);
[ "Size", "NrMovedPoints", "MovedPoints",
  "GeneratorsOfMagmaWithInverses", "silly" ]
gap> g!.silly;
"rubbish"
```

On the other hand, if we form an immutable externally represented list, we find that GAP will not let us change the object.

Example

```
gap> e := Enumerator(g);
<enumerator of perm group>
gap> IsMutable(e);
false
gap> IsList(e);
true
gap> e[3];
(1,2,4)
gap> e[3] := false;
Error, The list you are trying to assign to is immutable
```

When we consider copying objects, another filter `IsCopyable` (12.6.1), enters the game and we find that `ShallowCopy` (12.7.1) and `StructuralCopy` (12.7.2) behave quite differently. Objects can be divided for this purpose into three: mutable objects, immutable but copyable objects, and non-copyable objects (called constants).

A mutable or copyable object should have a method for the operation `ShallowCopy` (12.7.1), which should make a new mutable object, sharing its top-level subobjects with the original. The exact definition of top-level subobject may be defined by the implementor for new kinds of object.

`ShallowCopy` (12.7.1) applied to a constant simply returns the constant.

`StructuralCopy` (12.7.2) is expected to be much less used than `ShallowCopy` (12.7.1). Applied to a mutable object, it returns a new mutable object which shares no mutable sub-objects with the input. Applied to an immutable object (even a copyable one), it just returns the object. It is not an operation (indeed, it's a rather special kernel function).

Example

```
gap> e1 := StructuralCopy(e);
<enumerator of perm group>
```

```
gap> IsMutable(e1);
false
gap> e2 := ShallowCopy(e);
[ (), (1,4), (1,2,4), (1,3,4), (2,4), (1,4,2), (1,2), (1,3,4,2),
  (2,3,4), (1,4,2,3), (1,2,3), (1,3)(2,4), (3,4), (1,4,3), (1,2,4,3),
  (1,3), (2,4,3), (1,4,3,2), (1,2)(3,4), (1,3,2), (2,3), (1,4)(2,3),
  (1,2,3,4), (1,3,2,4) ]
gap>
```

There are two other related functions: `Immutable` (12.6.3), which makes a new immutable object which shares no mutable subobjects with its input and `MakeImmutable` (12.6.4) which changes an object and its mutable subobjects *in place* to be immutable. It should only be used on "new" objects that you have just created, and which cannot share mutable subobjects with anything else.

Both `Immutable` (12.6.3) and `MakeImmutable` (12.6.4) work on external objects by just resetting the `IsMutable` (12.6.2) filter in the object's type. This should make ineligible any methods that might change the object. As a consequence, you must allow for the possibility of immutable versions of any objects you create.

So, if you are implementing your own external objects. The rules amount to the following:

1. You decide if your objects should be mutable or copyable or constants, by fixing whether their type includes `IsMutable` (12.6.2) or `IsCopyable` (12.6.1).
2. You install methods for your objects respecting that decision:

   for constants:
   no methods change the underlying elements;

   for copyables:
   you provide a method for `ShallowCopy` (12.7.1);

   for mutables:
   you may have methods that change the underlying elements and these should explicitly require `IsMutable` (12.6.2).

79.18 Global Variables in the Library

Global variables in the GAP library are usually read-only in order to avoid their being overwritten accidentally. See also Section 4.9.

79.18.1 DeclareCategory

▷ `DeclareCategory(`*`name, super[, rank]`*`)` (function)

does the same as `NewCategory` (79.1.1) and additionally makes the variable *`name`* read-only.

79.18.2 DeclareRepresentation

▷ `DeclareRepresentation(`*`name, super, slots[, req]`*`)` (function)

does the same as `NewRepresentation` (79.2.1) and additionally makes the variable *`name`* read-only.

79.18.3 DeclareAttribute

▷ DeclareAttribute(*name, filter[, "mutable"][, rank]*) (function)

does the same as NewAttribute (79.3.1), additionally makes the variable *name* read-only and also binds read-only global variables with names Has*name* and Set*name* for the tester and setter of the attribute (see Section 13.6).

79.18.4 DeclareProperty

▷ DeclareProperty(*name, filter[, rank]*) (function)

does the same as NewProperty (79.3.2), additionally makes the variable *name* read-only and also binds read-only global variables with names Has*name* and Set*name* for the tester and setter of the property (see Section 13.6).

79.18.5 DeclareFilter

▷ DeclareFilter(*name[, rank]*) (function)

does the same as NewFilter (79.4.1) and additionally makes the variable *name* read-only.

79.18.6 DeclareOperation

▷ DeclareOperation(*name, filters*) (function)

does the same as NewOperation (79.5.1) and additionally makes the variable *name* read-only.

79.18.7 DeclareGlobalFunction

▷ DeclareGlobalFunction(*name, info*) (function)
▷ InstallGlobalFunction(*oper, func*) (function)

DeclareGlobalFunction GAP functions that are not operations and that are intended to be called by users should be notified to GAP in the declaration part of the respective package (see Section 79.19) via DeclareGlobalFunction, which returns a function that serves as a place holder for the function that will be installed later, and that will print an error message if it is called. See also DeclareSynonym (79.18.10).

A global function declared with DeclareGlobalFunction can be given its value *func* via InstallGlobalFunction; *gvar* is the global variable (or a string denoting its name) named with the *name* argument of the call to DeclareGlobalFunction. For example, a declaration like

Example

```
DeclareGlobalFunction( "SumOfTwoCubes" );
```

in the "declaration part" (see Section 79.19) might have a corresponding "implementation part" of:

Example

```
InstallGlobalFunction( SumOfTwoCubes, function(x, y) return x^3 + y^3; end);
```

79.18.8 DeclareGlobalVariable

▷ DeclareGlobalVariable(*name[, description]*) (function)

For global variables that are *not* functions, instead of using BindGlobal (4.9.7) one can also declare the variable with DeclareGlobalVariable which creates a new global variable named by the string *name*. If the second argument *description* is entered then this must be a string that describes the meaning of the global variable. DeclareGlobalVariable shall be used in the declaration part of the respective package (see 79.19), values can then be assigned to the new variable with InstallValue (79.18.9) or InstallFlushableValue (79.18.9), in the implementation part (again, see 79.19).

79.18.9 InstallValue

▷ InstallValue(*gvar, value*) (function)
▷ InstallFlushableValue(*gvar, value*) (function)

InstallValue assigns the value *value* to the global variable *gvar*. InstallFlushableValue does the same but additionally provides that each call of FlushCaches (79.18.11) will assign a structural copy of *value* to *gvar*.

InstallValue does *not* work if *value* is an "immediate object", i.e., an internally represented small integer or finite field element. It also fails for booleans. Furthermore, InstallFlushableValue works only if *value* is a list or a record. (Note that InstallFlushableValue makes sense only for *mutable* global variables.)

79.18.10 DeclareSynonym

▷ DeclareSynonym(*name, value*) (function)
▷ DeclareSynonymAttr(*name, value*) (function)

DeclareSynonym assigns the string *name* to a global variable as a synonym for *value*. Two typical intended usages are to declare an "and-filter", e.g.

Example

```
DeclareSynonym( "IsGroup", IsMagmaWithInverses and IsAssociative );
```

and to provide a previously declared global function with an alternative name, e.g.

Example

```
DeclareGlobalFunction( "SizeOfSomething" );
DeclareSynonym( "OrderOfSomething", SizeOfSomething );
```

Note: Before using DeclareSynonym in the way of this second example, one should determine whether the synonym is really needed. Perhaps an extra index entry in the documentation would be sufficient.

When *value* is actually an attribute then DeclareSynonymAttr should be used; this binds also globals variables Set*name* and Has*name* for its setter and tester, respectively.

Example

```
DeclareSynonymAttr( "IsField", IsDivisionRing and IsCommutative );
DeclareAttribute( "GeneratorsOfDivisionRing", IsDivisionRing );
DeclareSynonymAttr( "GeneratorsOfField", GeneratorsOfDivisionRing );
```

79.18.11 FlushCaches

▷ `FlushCaches()` (operation)

`FlushCaches` resets the value of each global variable that has been declared with `DeclareGlobalVariable` (79.18.8) and for which the initial value has been set with `InstallFlushableValue` (79.18.9) to this initial value.

`FlushCaches` should be used only for debugging purposes, since the involved global variables include for example lists that store finite fields and cyclotomic fields used in the current GAP session, in order to avoid that these fields are constructed anew in each call to `GF` (59.3.2) and `CF` (60.1.1).

79.19 Declaration and Implementation Part

Each package of GAP code consists of two parts, the *declaration part* that defines the new categories and operations for the objects the package deals with, and the *implementation part* where the corresponding methods are installed. The declaration part should be representation independent, representation dependent information should be dealt with in the implementation part.

GAP functions that are not operations and that are intended to be called by users should be notified to GAP in the declaration part via `DeclareGlobalFunction` (79.18.7). Values for these functions can be installed in the implementation part via `InstallGlobalFunction` (79.18.7).

Calls to the following functions belong to the declaration part.
`DeclareAttribute` (79.18.3),
`DeclareCategory` (79.18.1),
`DeclareFilter` (79.18.5),
`DeclareOperation` (79.18.6),
`DeclareGlobalFunction` (79.18.7),
`DeclareSynonym` (79.18.10),
`DeclareSynonymAttr` (79.18.10),
`DeclareProperty` (79.18.4),
`InstallTrueMethod` (78.7.1).

Calls to the following functions belong to the implementation part.
`DeclareRepresentation` (79.18.2),
`InstallGlobalFunction` (79.18.7),
`InstallMethod` (78.2.1),
`InstallImmediateMethod` (78.6.1),
`InstallOtherMethod` (78.2.2),
`NewFamily` (79.7.1),
`NewType` (79.8.1),
`Objectify` (79.9.1).

Whenever both a `New`*`Something`* and a `Declare`*`Something`* variant of a function exist (see 79.18), the use of `Declare`*`Something`* is recommended because this protects the variables in question from being overwritten. Note that there are *no* functions `DeclareFamily` and `DeclareType` since families and types are created dynamically, hence usually no global variables are associated to them. Further note that `DeclareRepresentation` (79.18.2) is regarded as belonging to the implementation part, because usually representations of objects are accessed only in very few places, and all code that involves a particular representation is contained in one file; additionally, representations

of objects are often not interesting for the user, so there is no need to provide a user interface or documentation about representations.

It should be emphasized that "declaration" means only an explicit notification of mathematical or technical terms or of concepts to GAP. For example, declaring a category or property with name `IsInteresting` does of course not tell GAP what this shall mean, and it is necessary to implement possibilities to create objects that know already that they lie in `IsInteresting` in the case that it is a category, or to install implications or methods in order to compute for a given object whether `IsInteresting` is `true` or `false` for it in the case that `IsInteresting` is a property.

Chapter 80

Examples of Extending the System

This chapter gives a few examples of how one can extend the functionality of GAP.

They are arranged in ascending difficulty. We show how to install new methods, add new operations and attributes and how to implement new features using categories and representations. (As we do not introduce completely new kinds of objects in these example it will not be necessary to declare any families.) Finally we show a simple way how to create new objects with an own arithmetic.

The examples given are all very rudimentary – no particular error checks are performed and the user interface sometimes is quite clumsy. These examples may be constructed for presentation purposes only and they do not necessarily constitute parts of the GAP library.

Even more complex examples that create whole classes of objects anew will be given in the following two chapters 81 and 82.

80.1 Addition of a Method

The easiest case is the addition of a new algorithm as a method for an existing operation for the existing structures.

For example, assume we wanted to implement a better method for computing the exponent of a nilpotent group (it is the product of the exponents of the Sylow subgroups).

The first task is to find which operation is used by GAP (it is `Exponent` (39.16.2)) and how it is declared. We can find this in the Reference Manual (in our particular case in section 39.16) and the declaration in the library file `lib/grp.gd`. The easiest way to find the place of the declaration is usually to `grep` over all `.gd` and `.g` files, see section 83.

In our example the declaration in the library is:

Example

```
DeclareAttribute("Exponent",IsGroup);
```

Similarly we find that the filter `IsNilpotentGroup` (39.15.3) represents the concept of being nilpotent.

We then write a function that implements the new algorithm which takes the right set of arguments and install it as a method. In our example this installation would be:

Example

```
InstallMethod(Exponent,"for nilpotent groups",
  [IsGroup and IsNilpotent],
function(G)
```

```
    [function body omitted]
  end);
```

We have left out the optional rank argument of `InstallMethod` (78.2.1), which normally is a wise choice –GAP automatically uses an internal ranking based on the filters that is only offset by the given rank. So our method will certainly be regarded as "better" than a method that has been installed for mere groups or for solvable groups but will be ranked lower than the library method for abelian groups.

That's all. Using 7.2.1 we can check for a nilpotent group that indeed our new method will be used.

When testing, remember that the method selection will not check for properties that are not known. (This is done internally by checking the property tester first.) Therefore the method would not be applicable for the group `g` in the following definition but only for the –mathematically identical but endowed with more knowledge by GAP– group `h`. (Section 80.3 shows a way around this.)

Example

```
gap> g:=Group((1,2),(1,3)(2,4));;
gap> h:=Group((1,2),(1,3)(2,4));;
gap> IsNilpotentGroup(h); # enforce test
true
gap> HasIsNilpotentGroup(g);
false
gap> HasIsNilpotentGroup(h);
true
```

Let's now look at a slightly more complicated example: We want to implement a better method for computing normalizers in a nilpotent permutation group. (Such an algorithm can be found for example in [LRW97].)

We already know `IsNilpotentGroup` (39.15.3), the filter `IsPermGroup` (43.1.1) represents the concept of being a group of permutations.

GAP uses `Normalizer` (39.11.1) to compute normalizers, however the declaration is a bit more complicated. In the library we find

Example

```
InParentFOA( "Normalizer", IsGroup, IsObject, NewAttribute );
```

The full mechanism of `InParentFOA` (85.2.1) is described in chapter 85, however for our purposes it is sufficient to know that for such a function the actual work is done by an operation `NormalizerOp`, an underlying operation for `Normalizer` (39.11.1) (and all the complications are just there to be able to remember certain results) and that the declaration of this operation is given by the first arguments, it would be:

Example

```
DeclareOperation( "NormalizerOp", [IsGroup, IsObject] );
```

This time we decide to enter a non-default family predicate in the call to `InstallMethod` (78.2.1). We could just leave it out as in the previous call; this would yield the default value, the function `ReturnTrue` (5.3.1) of arbitrary many arguments which always returns `true`. However, then the method might be called in some cases of inconsistent input (for example matrix groups in different characteristics) that ought to fall through the method selection to raise an error.

In our situation, we want the second group to be a subgroup of the first, so necessarily both must have the same family and we can use IsIdenticalObj (12.5.1) as family predicate.

Now we can install the method. Again this manual is lazy and does not show you the actual code:

Example

```
InstallMethod(NormalizerOp,"for nilpotent permutation groups",IsIdenticalObj,
  [IsPermGroup and IsNilpotentGroup,
   IsPermGroup and IsNilpotentGroup],
function(G,U)
  [ function body omitted ]
end);
```

80.2 Extending the Range of Definition of an Existing Operation

It might be that the operation has been defined so far only for a set of objects that is too restrictive for our purposes (or we want to install a method that takes another number of arguments). If this is the case, the call to InstallMethod (78.2.1) causes an error message. We can avoid this by using InstallOtherMethod (78.2.2) instead. It is also possible to re-declare an operation with another number of arguments and/or different filters for its arguments.

80.3 Enforcing Property Tests

As mentioned in Section 78.3, GAP does not check unknown properties to test whether a method might be applicable. In some cases one wants to enforce this, however, because the gain from knowing the property outweighs the cost of its determination.

In this situation one has to install a method *without* the additional property (so it can be tried even if the property is not yet known) and at high rank (so it will be used before other methods). The first thing to do in the actual function then is to test the property and to bail out with TryNextMethod (78.4.1) if it turns out to be false.

The above Exponent (39.16.2) example thus would become:

Example

```
InstallMethod(Exponent,"test abelianity", [IsGroup],
  50,# enforced high rank
function(G)
  if not IsAbelian(G) then
    TryNextMethod();
  fi;
  [remaining function body omitted]
end);
```

The value "50" used in this example is quite arbitrary. A better way is to use values that are given by the system inherently: We want this method still to be ranked as high, *as if it had* the IsAbelian (35.4.9) requirement. So we have GAP compute the extra rank of this:

Example

```
InstallMethod(Exponent,"test abelianity", [IsGroup],
  # enforced absolute rank of 'IsGroup and IsAbelian' installation: Subtract
  # the rank of 'IsGroup' and add the rank of 'IsGroup and IsAbelian':
  SIZE_FLAGS(FLAGS_FILTER(IsGroup and IsAbelian))
  -SIZE_FLAGS(FLAGS_FILTER(IsGroup)),
function(G)
```

the slightly complicated construction of addition and subtraction is necessary because IsGroup (39.2.7) and IsAbelian (35.4.9) might imply the *same* elementary filters which we otherwise would count twice.

A somehow similar situation occurs with matrix groups. Most methods for matrix groups are only applicable if the group is known to be finite.

However we should not enforce a finiteness test early (someone else later might install good methods for infinite groups while the finiteness test would be too expensive) but just before GAP would give a "no method found" error. This is done by redispatching, see 78.5. For example to enforce such a final finiteness test for normalizer calculations could be done by:

Example

```
RedispatchOnCondition(NormalizerOp,IsIdenticalObj,
  [IsMatrixGroup,IsMatrixGroup],[IsFinite,IsFinite],0);
```

80.4 Adding a new Operation

Next, we will consider how to add own operations. As an example we take the Sylow normalizer in a group of a given prime. This operation gets two arguments, the first has to be a group, the second a prime number.

There is a function IsPrimeInt (14.4.2), but no property for being prime (which would be pointless as integers cannot store property values anyhow). So the second argument gets specified only as positive integer:

Example

```
SylowNormalizer:=NewOperation("SylowNormalizer",[IsGroup,IsPosInt]);
```

(Note that we are using NewOperation (79.5.1) instead of DeclareOperation (79.18.6) as used in the library. The only difference other than that DeclareOperation (79.18.6) saves some typing, is that it also protects the variables against overwriting. When testing code (when one probably wants to change things) this might be restricting. If this does not bother you, you can use

Example

```
DeclareOperation("SylowNormalizer",[IsGroup,IsPosInt]);
```

as well.)

The filters IsGroup (39.2.7) and IsPosInt (14.2.2) given are *only* used to test that InstallMethod (78.2.1) installs methods with suitable arguments and will be completely ignored when using InstallOtherMethod (78.2.2). Technically one could therefore simply use IsObject (12.1.1) for all arguments in the declaration. The main point of using more specific filters here is to help documenting with which arguments the function is to be used (so for example a call SylowNormalizer(5,G) would be invalid).

Of course initially there are no useful methods for newly declared operations; you will have to write and install them yourself.

If the operation only takes one argument and has reproducible results without side effects, it might be worth declaring it as an attribute instead; see Section 80.5.

80.5 Adding a new Attribute

Now we look at an example of how to add a new attribute. As example we consider the set of all primes that divide the size of a group.

First we have to declare the attribute:

Example
```
PrimesDividingSize:=NewAttribute("PrimesDividingSize",IsGroup);
```

(See NewAttribute (79.3.1)). This implicitly declares attribute tester and setter, it is convenient however to assign these to variables as well:

Example
```
HasPrimesDividingSize:=Tester(PrimesDividingSize);
SetPrimesDividingSize:=Setter(PrimesDividingSize);
```

Alternatively, there is a declaration command DeclareAttribute (79.18.3) that executes all three assignments simultaneously and protects the variables against overwriting:

Example
```
DeclareAttribute("PrimesDividingSize",IsGroup);
```

Next we have to install method(s) for the attribute that compute its value. (This is not strictly necessary. We could use the attribute also without methods only for storing and retrieving information, but calling it for objects for which the value is not known would produce a "no method found" error.) For this purpose we can imagine the attribute simply as an one-argument operation:

Example
```
InstallMethod(PrimesDividingSize,"for finite groups",
  [IsGroup and IsFinite],
function(G)
  if Size(G)=1 then return [];
  else return Set(Factors(Size(G)));fi;
end);
```

The function installed *must* always return a value (or call TryNextMethod (78.4.1)). If the object is in the representation IsAttributeStoringRep this return value once computed will be automatically stored and retrieved if the attribute is called a second time. We don't have to call setter or tester ourselves. (This storage happens by GAP internally calling the attribute setter with the return value of the function. Retrieval is by a high-ranking method which is installed under the condition HasPrimesDividingSize. This method was installed automatically when the attribute was declared.)

80.6 Adding a new Representation

Next, we look at the implementation of a new representation of existing objects. In most cases we want to implement this representation only for efficiency reasons while keeping all the existing functionality.

For example, assume we wanted (following [Wie69]) to implement permutation groups defined by relations.

Next, we have to decide a few basics about the representation. All existing permutation groups in the library are attribute storing and we probably want to keep this for our new objects. Thus the representation must be a subrepresentation of IsComponentObjectRep and IsAttributeStoringRep. Furthermore we want each object to be a permutation group and we can imply this directly in the representation.

We also decide that we store the degree (the largest point that might be moved) in a component `degree` and the defining relations in a component `relations` (we do not specify the format of relations here. In an actual implementation one would have to design this as well, but it does not affect the declarations this chapter is about).

Example

```
IsPermutationGroupByRelations:=NewRepresentation(
  "IsPermutationGroupByRelations",
  IsComponentObjectRep and IsAttributeStoringRep and IsPermGroup,
  ["degree","relations"]);
```

(If we wanted to implement sparse matrices we might for example rather settle for a positional object in which we store a list of the nonzero entries.)

We can make the new representation a subrepresentation of an existing one. In such a case of course we have to provide all structure of this "parent" representation as well.

Next we need to check in which family our new objects will be. This will be the same family as of every other permutation group, namely the `CollectionsFamily(PermutationsFamily)` (where the family `PermutationsFamily = FamilyObj((1,2,3))` has been defined already in the library).

Now we can write a function to create our new objects. Usually it is helpful to look at functions from the library that are used in similar situations (for example `GroupByGenerators` (39.2.2) in our case) to make sure we have not forgotten any further requirements in the declaration we might have to add here. However in most cases the function is straightforward:

Example

```
PermutationGroupByRelations:=function(degree,relations)
local g
  g:=Objectify(NewType(CollectionsFamily(PermutationsFamily),
                       IsPermutationGroupByRelations),
               rec(degree:=degree,relations:=relations));
end;
```

It also is a good idea to install a `PrintObj` (6.3.5) and possibly also a `ViewObj` (6.3.5) method –otherwise testing becomes quite hard:

Example

```
InstallMethod(PrintObj,"for perm grps. given by relations",
  [IsPermutationGroupByRelations],
function(G)
  Print("PermutationGroupByRelations(", G!.degree,",",G!.relations,")");
end);
```

Next we have to write enough methods for the new representation so that the existing algorithms can be used. In particular we will have to implement methods for all operations for which library or kernel provides methods for the existing (alternative) representations. In our particular case there are no such methods. (If we would have implemented sparse matrices we would have had to implement methods for the list access and assignment functions, see 21.2.) However the existing way permutation groups are represented is by generators. To be able to use the existing machinery we want to be able to obtain a generating set also for groups in our new representation. This can be done (albeit not very effectively) by a stabilizer calculation in the symmetric group given by the `degree` component. The operation function to use is probably a bit complicated and will depend on the format of the `relations` (we have not specified in this example). In the following method we use `operationfunction` as a placeholder;

Example

```
InstallMethod(GeneratorsOfGroup,"for perm grps. given by relations",
  [IsPermutationGroupByRelations],
function(G)
local S,U;
  S:=SymmetricGroup(G!.degree);
  U:=Stabilizer(S,G!.relations,  operationfunction );
  return GeneratorsOfGroup(U);
end);
```

This is all we *must* do. Of course for performance reasons one might want to install methods for further operations as well.

80.7 Components versus Attributes

In the last section we introduced two new components, `G!.degree` and `G!.relations`. Technically, we could have used attributes instead. There is no clear distinction which variant is to be preferred: An attribute expresses part of the functionality available to certain objects (and thus could be computed later and probably even for a wider class of objects), a component is just part of the internal definition of an object.

So if the data is "of general interest", if we want the user to have access to it, attributes are preferable. Moreover, attributes can be used by the method selection (by specifying the filter `HasAttr` for an attribute `Attr`). They provide a clean interface and their immutability makes it safe to hand the data to a user who potentially could corrupt a components entries.

On the other hand more "technical" data (say the encoding of a sparse matrix) is better hidden from the user in a component, as declaring it as an attribute would not give any advantage.

Resource-wise, attributes need more memory (the attribute setter and tester are implicitly declared, and one filter bit is required), the attribute access is one further function call in the kernel, thus components might be an immeasurable bit faster.

80.8 Adding new Concepts

Now we look how to implement a new concept for existing objects and fit this in the method selection. Three examples that will be made more explicit below would be groups for which a "length" of elements (as a word in certain generators) is defined, groups that can be decomposed as a semidirect product and M-groups.

In each case we have two possibilities for the declaration. We can either declare it as a property or as a category. Both are eventually filter(s) and in this way indistinguishable for the method selection. However, the value of a property for a particular object can be unknown at first and later in the session be computed (to be `true` or `false`). This is implemented by reserving two filters for each property, one indicating whether the property value is known, and one, provided the value is known, to indicate the actual boolean value. Contrary to this, the decision whether or not an object lies in a category is taken at creation time and this is implemented using a single filter.

Property:

Properties also are attributes: If a property value is not known for an object, GAP tries to find a method to compute the property value. If no suitable method is found, an error is raised.

Category:

An object is in a category if it has been created in it. Testing the category for an object simply returns this value. Existing objects cannot enter a new category later in life. This means that in most cases one has to write own code to create objects in a new category.

If we want to implement a completely new concept so that new operations are defined only for the new objects –for example bialgebras for which a second scalar multiplication is defined– usually a category is chosen.

Technically, the behaviour of the category `IsXYZ`, declared as subcategory of `IsABC` is therefore exactly the same as if we would declare `IsXYZ` to be a property for `IsABC` and install the following method:

Example

```
InstallMethod(IsXYZ,"return false if not known",[IsABC],ReturnFalse);
```

(The word `category` also has a well-defined mathematical meaning, but this does not need to concern us at this point. The set of objects which is defined to be a (GAP) category does not need to be a category in the mathematical sense, vice versa not every mathematical category is declared as a (GAP) category.)

Eventually the choice between category and property often becomes a matter of taste or style.

Sometimes there is even a third possibility (if you have GAP 3 experience this might reflect most closely "an object whose operations record is `XYOps`"): We might want to indicate this new concept simply by the fact that certain attributes are set. In this case we could simply use the respective attribute tester(s).

The examples given below each give a short argument why the respective solution was chosen, but one could argue as well for other choices.

80.8.1 Example: M-groups

M-groups are finite groups for which all irreducible complex representations are induced from linear representations of subgroups, it turns out that they are all solvable and that every supersolvable group is an M-group. See [Isa76] for further details.

Solvability and supersolvability both are testable properties. We therefore declare `IsMGroup` as a property for solvable groups:

Example

```
IsMGroup:=NewProperty("IsMGroup",IsSolvableGroup);
```

The filter `IsSolvableGroup` (39.15.6) in this declaration *only* means that methods for `IsMGroup` by default can only be installed for groups that are (and know to be) solvable (though they could be installed for more general situations using `InstallOtherMethod` (78.2.2)). It does not yet imply that M-groups are solvable. We must do this deliberately via an implication and we use the same technique to imply that every supersolvable group is an M-group.

Example

```
InstallTrueMethod(IsSolvableGroup,IsMGroup);
InstallTrueMethod(IsMGroup,IsSupersolvableGroup);
```

Now we might install a method that tests for solvable groups whether they are M-groups:

Example

```
InstallMethod(IsMGroup,"for solvable groups",[IsSolvableGroup],
function(G)
  [... code omitted. The function must return ‘true’ or ‘false’ ...]
end);
```

Note that this example of declaring the IsMGroup property for solvable groups is not a part of the GAP library, which uses a similar but different filter IsMonomialGroup (39.15.9).

80.8.2 Example: Groups with a word length

Our second example is that of groups for whose elements a *word length* is defined. (We assume that the word length is only defined in the context of the group with respect to a preselected generating set but not for single elements alone. However we will not delve into any details of how this length is defined and how it could be computed.)

Having a word length is a feature which enables other operations (for example a “word length” function). This is exactly what categories are intended for and therefore we use one.

First, we declare the category. All objects in this category are groups and so we inherit the supercategory IsGroup (39.2.7):

Example

```
DeclareCategory("IsGroupWithWordLength",IsGroup);
```

We also define the operation which is “enabled” by this category, the word length of a group element, which is defined for a group and an element (remember that group elements are described by the category IsMultiplicativeElementWithInverse (31.14.13)):

Example

```
DeclareOperation("WordLengthOfElement",[IsGroupWithWordLength,
  IsMultiplicativeElementWithInverse]);
```

We then would proceed by installing methods to compute the word length in concrete cases and might for example add further operations to get shortest words in cosets.

80.8.3 Example: Groups with a decomposition as semidirect product

The third example is groups which have a (nontrivial) decomposition as a semidirect product. If this information has been found out, we want to be able to use it in algorithms. (Thus we do not only need the fact *that* there is a decomposition, but also the decomposition itself.)

We also want this to be applicable to every group and not only for groups which have been explicitly constructed via SemidirectProduct (49.2.1).

Instead we simply declare an attribute SemidirectProductDecomposition for groups. (Again, in this manual we don’t go in the details of how such an decomposition would look like).

Example

```
DeclareAttribute("SemidirectProductDecomposition",IsGroup);
```

If a decomposition has been found, it can be stored in a group using SetSemidirectProductDecomposition. (At the moment all groups in GAP are attribute storing.)

Methods that rely on the existence of such a decomposition then get installed for the tester filter HasSemidirectProductDecomposition.

80.9 Creating Own Arithmetic Objects

Finally let's look at a way to create new objects with a user-defined arithmetic such that one can form for example groups, rings or vector spaces of these elements. This topic is discussed in much more detail in chapter 82, in this section we present a simple approach that may be useful to get started but does not permit you to exploit all potential features.

The basic design is that the user designs some way to represent her objects in terms of GAPs built-in types, for example as a list or a record. We call this the "defining data" of the new objects. Also provided are functions that perform arithmetic on this "defining data", that is they take objects of this form and return objects that represent the result of the operation. The function `ArithmeticElementCreator` (80.9.1) then is called to provide a wrapping such that proper new GAP-objects are created which can be multiplied etc. with the default infix operations such as `*`.

80.9.1 ArithmeticElementCreator

▷ `ArithmeticElementCreator(`*spec*`)` (function)

offers a simple interface to create new arithmetic elements by providing functions that perform addition, multiplication and so forth, conforming to the specification *spec*. `ArithmeticElementCreator` creates a new category, representation and family for the new arithmetic elements being defined, and returns a function which takes the "defining data" of an element and returns the corresponding new arithmetic element.

spec is a record with one or more of the following components:

`ElementName`
: string used to identify the new type of object. A global identifier `Is`*`ElementName`* will be defined to indicate a category for these now objects. (Therefore it is not clever to have blanks in the name). Also a collections category is defined. (You will get an error message if the identifier `Is`*`ElementName`* is already defined.)

`Equality`, `LessThan`, `One`, `Zero`, `Multiplication`, `Inverse`, `Addition`, `AdditiveInverse`
: functions defining the arithmetic operations. The functions interface on the level of "defining data", the actual methods installed will perform the unwrapping and wrapping as objects. Components are optional, but of course if no multiplication is defined elements cannot be multiplied and so forth.

  There are default methods for `Equality` and `LessThan` which simply calculate on the defining data. If one is defined, it must be ensured that the other is compatible (so that $a < b$ implies not $(a = b)$)

`Print`
: a function which prints the object. By default, just the defining data is printed.

`MathInfo`
: filters determining the mathematical properties of the elements created. A typical value is for example `IsMultiplicativeElementWithInverse` for group elements.

`RepInfo`
: filters determining the representational properties of the elements created. The objects

created are always component objects, so in most cases the only reasonable option is `IsAttributeStoringRep` to permit the storing of attributes.

All components are optional and will be filled in with default values (though of course an empty record will not result in useful objects).

Note that the resulting objects are *not equal* to their defining data (even though by default they print as only the defining data). The operation `UnderlyingElement` can be used to obtain the defining data of such an element.

80.9.2 Example: ArithmeticElementCreator

As the first example we look at subsets of $\{1,\ldots,4\}$ and define an "addition" as union and "multiplication" as intersection. These operations are both commutative and we want the resulting elements to know this.

We therefore use the following specification:

Example

```
gap> # the whole set
gap> w := [1,2,3,4];
[ 1, 2, 3, 4 ]
gap> PosetElementSpec :=rec(
>   # name of the new elements
>   ElementName := "PosetOn4",
>   # arithmetic operations
>   One := a -> w,
>   Zero := a -> [],
>   Multiplication := function(a, b) return Intersection(a, b); end,
>   Addition := function(a, b) return Union(a, b); end,
>   # Mathematical properties of the elements
>   MathInfo := IsCommutativeElement and
>               IsAssociativeElement and
>               IsAdditivelyCommutativeElement
> );;
gap> mkposet := ArithmeticElementCreator(PosetElementSpec);
function( x ) ... end
```

Now we can create new elements, perform arithmetic on them and form domains:

Example

```
gap> a := mkposet([1,2,3]);
[ 1, 2, 3 ]
gap> CategoriesOfObject(a);
[ "IsExtAElement", "IsNearAdditiveElement",
  "IsNearAdditiveElementWithZero", "IsAdditiveElement",
  "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsAssociativeElement",
  "IsAdditivelyCommutativeElement", "IsCommutativeElement",
  "IsPosetOn4" ]
gap> a=[1,2,3];
false
gap> UnderlyingElement(a)=[1,2,3];
true
gap> b:=mkposet([2,3,4]);
```

```
[ 2, 3, 4 ]
gap> a+b;
[ 1, 2, 3, 4 ]
gap> a*b;
[ 2, 3 ]
gap> s:=Semigroup(a,b);
<semigroup with 2 generators>
gap> Size(s);
3
```

The categories IsPosetOn4 and IsPosetOn4Collection can be used to install methods specific to the new objects.

Example

```
gap> IsPosetOn4Collection(s);
true
```

Chapter 81

An Example – Residue Class Rings

In this chapter, we give an example how GAP can be extended by new data structures and new functionality. In order to focus on the issues of the implementation, the mathematics in the example chosen is trivial. Namely, we will discuss computations with elements of residue class rings $\mathbb{Z}/n\mathbb{Z}$.

The first attempt is straightforward (see Section 81.1), it deals with the implementation of the necessary arithmetic operations. Section 81.2 deals with the question why it might be useful to use an approach that involves creating a new data structure and integrating the algorithms dealing with these new GAP objects into the system. Section 81.3 shows how this can be done in our example, and Section 81.4, the question of further compatibility of the new objects with known GAP objects is discussed. Finally, Section 81.5 gives some hints how to improve the implementation presented before.

81.1 A First Attempt to Implement Elements of Residue Class Rings

Suppose we want to do computations with elements of a ring $\mathbb{Z}/n\mathbb{Z}$, where n is a positive integer.

First we have to decide how to represent the element $k+n\mathbb{Z}$ in GAP. If the modulus n is fixed then we can use the integer k. More precisely, we can use any integer k' such that $k-k'$ is a multiple of n. If different moduli are likely to occur then using a list of the form $[k,n]$, or a record of the form `rec( residue := k, modulus := n )` is more appropriate. In the following, let us assume the list representation $[k,n]$ is chosen. Moreover, we decide that the residue k in all such lists satisfies $0 \leq k < n$, i.e., the result of adding two residue classes represented by $[k_1,n]$ and $[k_2,n]$ (of course with same modulus n) will be $[k,n]$ with k_1+k_2 congruent to k modulo n and $0 \leq k < n$.

Now we can implement the arithmetic operations for residue classes. Note that the result of the `mod` operator is normalized as required. The division by a noninvertible residue class results in `fail`.

Example

```
gap> resclass_sum := function( c1, c2 )
>    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
>    return [ ( c1[1] + c2[1] ) mod c1[2], c1[2] ];
> end;;
gap>
gap> resclass_diff := function( c1, c2 )
>    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
>    return [ ( c1[1] - c2[1] ) mod c1[2], c1[2] ];
> end;;
gap>
```

```
gap> resclass_prod := function( c1, c2 )
>    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
>    return [ ( c1[1] * c2[1] ) mod c1[2], c1[2] ];
> end;;
gap>
gap> resclass_quo := function( c1, c2 )
>    local quo;
>    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
>    quo:= QuotientMod( c1[1], c2[1], c1[2] );
>    if quo <> fail then
>      quo:= [ quo, c1[2] ];
>    fi;
>    return quo;
> end;;
```

With these functions, we can in principle compute with residue classes.

Example

```
gap> list:= List( [ 0 .. 3 ], k -> [ k, 4 ] );
[ [ 0, 4 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ] ]
gap> resclass_sum( list[2], list[4] );
[ 0, 4 ]
gap> resclass_diff( list[1], list[2] );
[ 3, 4 ]
gap> resclass_prod( list[2], list[4] );
[ 3, 4 ]
gap> resclass_prod( list[3], list[4] );
[ 2, 4 ]
gap> List( list, x -> resclass_quo( list[2], x ) );
[ fail, [ 1, 4 ], fail, [ 3, 4 ] ]
```

81.2 Why Proceed in a Different Way?

It depends on the computations we intended to do with residue classes whether or not the implementation described in the previous section is satisfactory for us.

Probably we are mainly interested in more complex data structures than the residue classes themselves, for example in matrix algebras or matrix groups over a ring such as $\mathbb{Z}/4\mathbb{Z}$. For this, we need functions to add, multiply, invert etc. matrices of residue classes. Of course this is not a difficult task, but it requires to write additional GAP code.

And when we have implemented the arithmetic operations for matrices of residue classes, we might be interested in domain operations such as computing the order of a matrix group over $\mathbb{Z}/4\mathbb{Z}$, a Sylow 2 subgroup, and so on. The problem is that a residue class represented as a pair $[k,n]$ is not regarded as a group element by GAP. We have not yet discussed how a matrix of residue classes shall be represented, but if we choose the obvious representation of a list of lists of our residue classes then also this is not a valid group element in GAP. Hence we cannot apply the function `Group` (39.2.1) to create a group of residue classes or a group of matrices of residue classes. This is because GAP assumes that group elements can be multiplied via the infix operator $*$ (equivalently, via the operation `\*` (31.12.1)). Note that in fact the multiplication of two lists $[k_1,n]$, $[k_2,n]$ is defined, but we have $[k_1,n]*[k_2,n] = k_1*k_2+n*n$, the standard scalar product of two row vectors of same length. That

is, the multiplication with * is not compatible with the function resclass_prod introduced in the previous section. Similarly, ring elements are assumed to be added via the infix operator +; the addition of residue classes is not compatible with the available addition of row vectors.

What we have done in the previous section can be described as implementation of a "standalone" arithmetic for residue classes. In order to use the machinery of the GAP library for creating higher level objects such as matrices, polynomials, or domains over residue class rings, we have to "integrate" this implementation into the GAP library. The key step will be to create a new kind of GAP objects. This will be done in the following sections; there we assume that residue classes and residue class rings are not yet available in GAP; in fact they are available, and their implementation is very close to what is described here.

81.3 A Second Attempt to Implement Elements of Residue Class Rings

Faced with the problem to implement elements of the rings $\mathbb{Z}/n\mathbb{Z}$, we must define the *types* of these elements as far as is necessary to distinguish them from other GAP objects.

As is described in Chapter 13, the type of an object comprises several aspects of information about this object; the *family* determines the relation of the object to other objects, the *categories* determine what operations the object admits, the *representation* determines how an object is actually represented, and the *attributes* describe knowledge about the object.

First of all, we must decide about the *family* of each residue class. A natural way to do this is to put the elements of each ring $\mathbb{Z}/n\mathbb{Z}$ into a family of their own. This means that for example elements of $\mathbb{Z}/3\mathbb{Z}$ and $\mathbb{Z}/9\mathbb{Z}$ lie in different families. So the only interesting relation between the families of two residue classes is equality; binary arithmetic operations with two residue classes will be admissible only if their families are equal. Note that in the naive approach in Section 81.1, we had to take care of different moduli by a check in each function; these checks may disappear in the new approach because of our choice of families.

Note that we do not need to tell GAP anything about the above decision concerning the families of the objects that we are going to implement, that is, the *declaration part* (see 79.19) of the little GAP package we are writing contains nothing about the distribution of the new objects into families. (The actual construction of a family happens in the function MyZmodnZ shown below.)

Second, we want to describe methods to add or multiply two elements in $\mathbb{Z}/n\mathbb{Z}$, and these methods shall be not applicable to other GAP objects. The natural way to do this is to create a new *category* in which all elements of all rings $\mathbb{Z}/n\mathbb{Z}$ lie. This is done as follows.

Example

```
gap> DeclareCategory( "IsMyZmodnZObj", IsScalar );
gap> cat:= CategoryCollections( IsMyZmodnZObj );;
gap> cat:= CategoryCollections( cat );;
gap> cat:= CategoryCollections( cat );;
```

So all elements in the rings $\mathbb{Z}/n\mathbb{Z}$ will lie in the category IsMyZmodnZObj, which is a subcategory of IsScalar (31.14.20). The latter means that one can add, subtract, multiply and divide two such elements that lie in the same family, with the obvious restriction that the second operand of a division must be invertible. (The name IsMyZmodnZObj is chosen because IsZmodnZObj (14.5.4) is already defined in GAP, for an implementation of residue classes that is very similar to the one developed in this manual chapter. Using this different name, one can simply enter the GAP code of this chapter into a GAP session, either interactively or by reading a file with this code, and experiment after each step whether the expected behaviour has been achieved, and what is still missing.)

The next lines of GAP code above create the categories `CategoryCollections( IsMyZmodnZObj )` and two higher levels of collections categories of this, which will be needed later; it is important to create these categories before collections of the objects in `IsMyZmodnZObj` actually arise.

Note that the only difference between `DeclareCategory` (79.18.1) and `NewCategory` (79.1.1) is that in a call to `DeclareCategory` (79.18.1), a variable corresponding to the first argument is set to the new category, and this variable is read-only (see 79.18). The same holds for `DeclareRepresentation` (79.18.2) and `NewRepresentation` (79.2.1) etc.

There is no analogue of categories in the implementation in Section 81.1, since there it was not necessary to distinguish residue classes from other GAP objects. Note that the functions there assumed that their arguments were residue classes, and the user was responsible not to call them with other arguments. Thus an important aspect of types is to describe arguments of functions explicitly.

Third, we must decide about the *representation* of our objects. This is something we know already from Section 81.1, where we chose a list of length two. Here we may choose between two essentially different representations for the new GAP objects, namely as "component object" (record-like) or "positional object" (list-like). We decide to store the modulus of each residue class in its family, and to encode the element $k+n\mathbb{Z}$ by the unique residue in the range $[0..n-1]$ that is congruent to k modulo n, and the object itself is chosen to be a positional object with this residue at the first and only position (see 79.11).

Example

```
gap> DeclareRepresentation("IsMyModulusRep", IsPositionalObjectRep, [1]);
```

The fourth ingredients of a type, *attributes*, are usually of minor importance for element objects. In particular, we do not need to introduce special attributes for residue classes.

Having defined what the new objects shall look like, we now declare a global function (see 79.19), to create an element when family and residue are given.

Example

```
gap> DeclareGlobalFunction( "MyZmodnZObj" );
```

Now we have declared what we need, and we can start to implement the missing methods resp. functions; so the following command belongs to the *implementation part* of our package (see 79.19).

The probably most interesting function is the one to construct a residue class.

Example

```
gap> InstallGlobalFunction( MyZmodnZObj, function( Fam, residue )
>    return Objectify( NewType( Fam, IsMyZmodnZObj and IsMyModulusRep ),
>                      [ residue mod Fam!.modulus ] );
> end );
```

Note that we normalize `residue` explicitly using `mod`; we assumed that the modulus is stored in `Fam`, so we must take care of this below. If `Fam` is a family of residue classes, and `residue` is an integer, `MyZmodnZObj` returns the corresponding object in the family `Fam`, which lies in the category `IsMyZmodnZObj` and in the representation `IsMyModulusRep`.

`MyZmodnZObj` needs an appropriate family as first argument, so let us see how to get our hands on this. Of course we could write a handy function to create such a family for given modulus, but we choose another way. In fact we do not really want to call `MyZmodnZObj` explicitly when we want to create residue classes. For example, if we want to enter a matrix of residues then usually we start with a

matrix of corresponding integers, and it is more elegant to do the conversion via multiplying the matrix with the identity of the required ring $\mathbb{Z}/n\mathbb{Z}$; this is also done for the conversion of integral matrices to finite field matrices. (Note that we will have to install a method for this.) So it is often sufficient to access this identity, for example via `One( MyZmodnZ( n ) )`, where `MyZmodnZ` returns a domain representing the ring $\mathbb{Z}/n\mathbb{Z}$ when called with the argument n. We decide that constructing this ring is a natural place where the creation of the family can be hidden, and implement the function. (Note that the declaration belongs to the declaration part, and the installation belongs to the implementation part, see 79.19).

Example

```
gap> DeclareGlobalFunction( "MyZmodnZ" );
gap>
gap> InstallGlobalFunction( MyZmodnZ, function( n )
>    local F, R;
>
>    if not IsPosInt( n ) then
>      Error( "<n> must be a positive integer" );
>    fi;
>
>    # Construct the family of element objects of our ring.
>    F:= NewFamily( Concatenation( "MyZmod", String( n ), "Z" ),
>                   IsMyZmodnZObj );
>
>    # Install the data.
>    F!.modulus:= n;
>
>    # Make the domain.
>    R:= RingWithOneByGenerators( [ MyZmodnZObj( F, 1 ) ] );
>    SetIsWholeFamily( R, true );
>    SetName( R, Concatenation( "(Integers mod ", String(n), ")" ) );
>
>    # Return the ring.
>    return R;
> end );
```

Note that the modulus `n` is stored in the component `modulus` of the family, as is assumed by `MyZmodnZ`. Thus it is not necessary to store the modulus in each element. When storing `n` with the `!.` operator as value of the component `modulus`, we used that all families are in fact represented as component objects (see 79.10).

We see that we can use `RingWithOneByGenerators` (56.3.3) to construct a ring with one if we have the appropriate generators. The construction via `RingWithOneByGenerators` (56.3.3) makes sure that `IsRingWithOne` (56.3.1) (and `IsRing` (56.1.1)) is `true` for each output of `MyZmodnZ`. So the main problem is to create the identity element of the ring, which in our case suffices to generate the ring. In order to create this element via `MyZmodnZObj`, we have to construct its family first, at each call of `MyZmodnZ`.

Also note that we may enter known information about the ring. Here we store that it contains the whole family of elements; this is useful for example when we want to check the membership of an element in the ring, which can be decided from the type of the element if the ring contains its whole elements family. Giving a name to the ring causes that it will be printed via printing the name. (By the way: This name `(Integers mod n)` looks like a call to `\mod` (31.12.1) with the arguments

Integers (14) and *n*; a construction of the ring via this call seems to be more natural than by calling MyZmodnZ; later we shall install a \mod (31.12.1) method in order to admit this construction.)

Now we can read the above code into GAP, and the following works already.

Example

```
gap> R:= MyZmodnZ( 4 );
(Integers mod 4)
gap> IsRing( R );
true
gap> gens:= GeneratorsOfRingWithOne( R );
[ <object> ]
```

But of course this means just to ask for the information we have explicitly stored in the ring. Already the questions whether the ring is finite and how many elements it has, cannot be answered by GAP. Clearly we know the answers, and we could store them in the ring, by setting the value of the property IsFinite (30.4.2) to true and the value of the attribute Size (30.4.6) to *n* (the argument of the call to MyZmodnZ). If we do not want to do so then GAP could only try to find out the number of elements of the ring via forming the closure of the generators under addition and multiplication, but up to now, GAP does not know how to add or multiply two elements of our ring.

So we must install some methods for arithmetic and other operations if the elements are to behave as we want.

We start with a method for showing elements nicely on the screen. There are different operations for this purpose. One of them is PrintObj (6.3.5), which is called for each argument in an explicit call to Print (6.3.4). Another one is ViewObj (6.3.5), which is called in the read-eval-print loop for each object. ViewObj (6.3.5) shall produce short and human readable information about the object in question, whereas PrintObj (6.3.5) shall produce information that may be longer and is (if reasonable) readable by GAP. We cannot satisfy the latter requirement for a PrintObj (6.3.5) method because there is no way to make a family GAP readable. So we decide to display the expression (k mod n) for an object that is given by the residue k and the modulus n, which would be fine as a ViewObj (6.3.5) method. Since the default for ViewObj (6.3.5) is to call PrintObj (6.3.5), and since no other ViewObj (6.3.5) method is applicable to our elements, we need only a PrintObj (6.3.5) method.

Example

```
gap> InstallMethod( PrintObj,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObj and IsMyModulusRep ],
>    function( x )
>    Print( "( ", x![1], " mod ", FamilyObj(x)!.modulus, " )" );
>    end );
```

So we installed a method for the operation PrintObj (6.3.5) (first argument), and we gave it a suitable information message (second argument), see 7.2.1 and 7.3 for applications of this information string. The third argument tells GAP that the method is applicable for objects that lie in the category IsMyZmodnZObj and in the representation IsMyModulusRep. and the fourth argument is the method itself. More details about InstallMethod (78.2.1) can be found in 78.2.

Note that the requirement IsMyModulusRep for the argument x allows us to access the residue as x![1]. Since the family of x has the component modulus bound if it is constructed by MyZmodnZ, we may access this component. We check whether the method installation has some effect.

Example

```
gap> gens;
[ ( 1 mod 4 ) ]
```

Next we install methods for the comparison operations. Note that we can assume that the residues in the representation chosen are normalized.

Example

```
gap> InstallMethod( \=,
>    "for two elements in Z/nZ (ModulusRep)",
>    IsIdenticalObj,
>    [IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep],
>    function( x, y ) return x![1] = y![1]; end );
gap>
gap> InstallMethod( \<,
>    "for two elements in Z/nZ (ModulusRep)",
>    IsIdenticalObj,
>    [IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep],
>    function( x, y ) return x![1] < y![1]; end );
```

The third argument used in these installations specifies the required relation between the families of the arguments (see 13.1). This argument of a method installation, if present, is a function that shall be applied to the families of the arguments. `IsIdenticalObj` (12.5.1) means that the methods are applicable only if both arguments lie in the same family. (In installations for unary methods, obviously no relation is required, so this argument is left out there.)

Up to now, we see no advantage of the new approach over the one in Section 81.1. For a residue class represented as `[ k, n ]`, the way it is printed on the screen is sufficient, and equality and comparison of lists are good enough to define equality and comparison of residue classes if needed. But this is not the case in other situations. For example, if we would have decided that the residue k need not be normalized then we would have needed functions in Section 81.1 that compute whether two residue classes are equal, and which of two residue classes is regarded as larger than another. Note that we are free to define what "larger" means for objects that are newly introduced.

Next we install methods for the arithmetic operations, first for the additive structure.

Example

```
gap> InstallMethod( \+,
>    "for two elements in Z/nZ (ModulusRep)",
>    IsIdenticalObj,
>    [IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep],
>    function( x, y )
>    return MyZmodnZObj( FamilyObj( x ), x![1] + y![1] );
>    end );
gap>
gap> InstallMethod( ZeroOp,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObj ],
>    x -> MyZmodnZObj( FamilyObj( x ), 0 ) );
gap>
gap> InstallMethod( AdditiveInverseOp,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObj and IsMyModulusRep ],
>    x -> MyZmodnZObj( FamilyObj( x ), AdditiveInverse( x![1] ) ) );
```

Here the new approach starts to pay off. The method for the operation \+ (31.12.1) allows us to use the infix operator + for residue classes. The method for ZeroOp (31.10.3) is used when we call this operation or the attribute Zero (31.10.3) explicitly, and ZeroOp (31.10.3) it is also used when we ask for `0 * rescl`, where *rescl* is a residue class.

(Note that Zero (31.10.3) and ZeroOp (31.10.3) are distinguished because `0 * obj` is guaranteed to return a *mutable* result whenever a mutable version of this result exists in GAP –for example if *obj* is a matrix– whereas Zero (31.10.3) is an attribute and therefore returns *immutable* results; for our example there is no difference since the residue classes are always immutable, nevertheless we have to install the method for ZeroOp (31.10.3). The same holds for AdditiveInverse (31.10.9), One (31.10.2), and Inverse (31.10.8).)

Similarly, AdditiveInverseOp (31.10.9) can be either called directly or via the unary - operator; so we can compute the additive inverse of the residue class *rescl* as `-rescl`.

It is not necessary to install methods for subtraction, since this is handled via addition of the additive inverse of the second argument if no other method is installed.

Let us try what we can do with the methods that are available now.

Example

```
gap> x:= gens[1];  y:= x + x;
( 1 mod 4 )
( 2 mod 4 )
gap> 0 * x;  -x;
( 0 mod 4 )
( 3 mod 4 )
gap> y = -y;  x = y;  x < y;  -x < y;
true
false
true
false
```

We might want to admit the addition of integers and elements in rings $\mathbb{Z}/n\mathbb{Z}$, where an integer is implicitly identified with its residue modulo n. To achieve this, we install methods to add an integer to an object in IsMyZmodnZObj from the left and from the right.

Example

```
gap> InstallMethod( \+,
>    "for element in Z/nZ (ModulusRep) and integer",
>    [ IsMyZmodnZObj and IsMyModulusRep, IsInt ],
>    function( x, y )
>    return MyZmodnZObj( FamilyObj( x ), x![1] + y );
>    end );
gap>
gap> InstallMethod( \+,
>    "for integer and element in Z/nZ (ModulusRep)",
>    [ IsInt, IsMyZmodnZObj and IsMyModulusRep ],
>    function( x, y )
>    return MyZmodnZObj( FamilyObj( y ), x + y![1] );
>    end );
```

Now we can do also the following.

Example

```
gap> 2 + x;  7 - x;  y - 2;
( 3 mod 4 )
```

```
( 2 mod 4 )
( 0 mod 4 )
```

Similarly we install the methods dealing with the multiplicative structure. We need methods to multiply two of our objects, and to compute identity and inverse. The operation `OneOp` (31.10.2) is called when we ask for *`rescl`*`^0`, and `InverseOp` (31.10.8) is called when we ask for *`rescl`*`^-1`. Note that the method for `InverseOp` (31.10.8) returns `fail` if the argument is not invertible.

Example

```
gap> InstallMethod( \*,
>    "for two elements in Z/nZ (ModulusRep)",
>    IsIdenticalObj,
>    [IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep],
>    function( x, y )
>    return MyZmodnZObj( FamilyObj( x ), x![1] * y![1] );
>    end );
gap>
gap> InstallMethod( OneOp,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObj ],
>    elm -> MyZmodnZObj( FamilyObj( elm ), 1 ) );
gap>
gap> InstallMethod( InverseOp,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObj and IsMyModulusRep ],
>    function( elm )
>    local residue;
>    residue:= QuotientMod( 1, elm![1], FamilyObj( elm )!.modulus );
>    if residue <> fail then
>      residue:= MyZmodnZObj( FamilyObj( elm ), residue );
>    fi;
>    return residue;
>    end );
```

To be able to multiply our objects with integers, we need not (but we may, and we should if we are going for efficiency) install special methods. This is because in general, GAP interprets the multiplication of an integer and an additive object as abbreviation of successive additions, and there is one generic method for such a multiplication that uses only additions and –in the case of a negative integer– taking the additive inverse. Analogously, there is a generic method for powering by integers that uses only multiplications and taking the multiplicative inverse.

Note that we could also interpret the multiplication with an integer as a shorthand for the multiplication with the corresponding residue class. We are lucky that this interpretation is compatible with the one that is already available. If this would not be the case then of course we would get into trouble by installing a concurrent multiplication that computes something different from the multiplication that is already defined, since GAP does not guarantee which of the applicable methods is actually chosen (see 78.3).

Now we have implemented methods for the arithmetic operations for our elements, and the following calculations work.

Example

```
gap> y:= 2 * x;  z:= (-5) * x;
( 2 mod 4 )
```

```
( 3 mod 4 )
gap> y * z;  y * y;
( 2 mod 4 )
( 0 mod 4 )
gap> y^-1;  y^0;
fail
( 1 mod 4 )
gap> z^-1;
( 3 mod 4 )
```

There are some other operations in GAP that we may want to accept our elements as arguments. An example is the operation `Int` (14.2.3) that returns, e.g., the integral part of a rational number or the integer corresponding to an element in a finite prime field. For our objects, we may define that `Int` (14.2.3) returns the normalized residue.

Note that we *define* this behaviour for elements but we *implement* it for objects in the representation `IsMyModulusRep`. This means that if someone implements another representation of residue classes then this person must be careful to implement `Int` (14.2.3) methods for objects in this new representation compatibly with our definition, i.e., such that the result is independent of the representation.

Example

```
gap> InstallMethod( Int,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObj and IsMyModulusRep ],
>    z -> z![1] );
```

Another example of an operation for which we might want to install a method is `\mod` (31.12.1). We make the ring print itself as `Integers` (14) mod the modulus, and then it is reasonable to allow a construction this way, which makes the `PrintObj` (6.3.5) output of the ring GAP readable.

Example

```
gap> InstallMethod( PrintObj,
>    "for full collection Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
>    function( R )
>    Print( "(Integers mod ",
>           ElementsFamily( FamilyObj(R) )!.modulus, ")" );
>    end );
gap>
gap> InstallMethod( \mod,
>    "for ‘Integers’, and a positive integer",
>    [ IsIntegers, IsPosRat and IsInt ],
>    function( Integers, n ) return MyZmodnZ( n ); end );
```

Let us try this.

Example

```
gap> Int( y );
2
gap> Integers mod 1789;
(Integers mod 1789)
```

Probably it is not necessary to emphasize that with the approach of Section 81.1, installing methods for existing operations is usually not possible or at least not recommended. For example, installing the function `resclass_sum` defined in Section 81.1 as a `\+` (31.12.1) method for adding two lists of length two (with integer entries) would not be compatible with the general definition of the addition of two lists of same length. Installing a method for the operation `Int` (14.2.3) that takes a list `[ k, n ]` and returns k would in principle be possible, since there is no `Int` (14.2.3) method for lists yet, but it is not sensible to do so because one can think of other interpretations of such a list where different `Int` (14.2.3) methods could be installed with the same right.

As mentioned in Section 81.2, one advantage of the new approach is that with the implementation we have up to now, automatically also matrices of residue classes can be treated.

Example

```
gap> r:= Integers mod 16;
(Integers mod 16)
gap> x:= One( r );
( 1 mod 16 )
gap> mat:= IdentityMat( 2 ) * x;
[ [ ( 1 mod 16 ), ( 0 mod 16 ) ], [ ( 0 mod 16 ), ( 1 mod 16 ) ] ]
gap> mat[1][2]:= x;;
gap> mat;
[ [ ( 1 mod 16 ), ( 1 mod 16 ) ], [ ( 0 mod 16 ), ( 1 mod 16 ) ] ]
gap> Order( mat );
16
gap> mat + mat;
[ [ ( 2 mod 16 ), ( 2 mod 16 ) ], [ ( 0 mod 16 ), ( 2 mod 16 ) ] ]
gap> last^4;
[ [ ( 0 mod 16 ), ( 0 mod 16 ) ], [ ( 0 mod 16 ), ( 0 mod 16 ) ] ]
```

Such matrices, if they are invertible, are valid as group elements. One technical problem is that the default algorithm for inverting matrices may give up since Gaussian elimination need not be successful over rings containing zero divisors. Therefore we install a simpleminded inversion method that inverts an integer matrix.

Example

```
gap> InstallMethod( InverseOp,
>    "for an ordinary matrix over a ring Z/nZ",
>    [ IsMatrix and IsOrdinaryMatrix
>      and CategoryCollections( CategoryCollections( IsMyZmodnZObj ) ) ],
>    function( mat )
>    local one, modulus;
>
>    one:= One( mat[1][1] );
>    modulus:= FamilyObj( one )!.modulus;
>    mat:= InverseOp( List( mat, row -> List( row, Int ) ) );
>    if mat <> fail then
>      mat:= ( mat mod modulus ) * one;
>    fi;
>    if not IsMatrix( mat ) then
>      mat:= fail;
>    fi;
>    return mat;
>    end );
```

Additionally we install a method for finding a domain that contains the matrix entries; this is used by some GAP library functions.

Example

```
gap> InstallMethod( DefaultFieldOfMatrixGroup,
>     "for a matrix group over a ring Z/nZ",
>     [ IsMatrixGroup and CategoryCollections( CategoryCollections(
>           CategoryCollections( IsMyZmodnZObj ) ) ) ],
>     G -> RingWithOneByGenerators([ One( Representative( G )[1][1] ) ]));
```

Now we can deal with matrix groups over residue class rings.

Example

```
gap> mat2:= IdentityMat( 2 ) * x;;
gap> mat2[2][1]:= x;;
gap> g:= Group( mat, mat2 );;
gap> Size( g );
3072
gap> Factors( last );
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3 ]
gap> syl3:= SylowSubgroup( g, 3 );;
gap> gens:= GeneratorsOfGroup( syl3 );
[ [ [ ( 1 mod 16 ), ( 7 mod 16 ) ], [ ( 11 mod 16 ), ( 14 mod 16 ) ] ] ]
gap> Order( gens[1] );
3
```

It should be noted that this way more involved methods for matrix groups may not be available. For example, many questions about a finite matrix group can be delegated to an isomorphic permutation group via a so-called “nice monomorphism”; this can be controlled by the filter `IsHandledByNiceMonomorphism` (40.5.1).

By the way, also groups of (invertible) residue classes can be formed, but this may be of minor interest.

Example

```
gap> g:= Group( x );;  Size( g );
#I  default ‘IsGeneratorsOfMagmaWithInverses’ method returns ‘true’ for
[ ( 1 mod 16 ) ]
1
gap> g:= Group( 3*x );;  Size( g );
#I  default ‘IsGeneratorsOfMagmaWithInverses’ method returns ‘true’ for
[ ( 3 mod 16 ) ]
4
```

(The messages above tell that GAP does not know a method for deciding whether the given elements are valid group elements. We could add an appropriate `IsGeneratorsOfMagmaWithInverses` method if we would want.)

Having done enough for the elements, we may install some more methods for the rings if we want to use them as arguments. These rings are finite, and there are many generic methods that will work if they are able to compute the list of elements of the ring, so we install a method for this.

Example

```
gap> InstallMethod( Enumerator,
>    "for full collection Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
```

```
>    function( R )
>    local F;
>    F:= ElementsFamily( FamilyObj(R) );
>    return List( [ 0 .. Size( R ) - 1 ], x -> MyZmodnZObj( F, x ) );
>    end );
```

Note that this method is applicable only to full rings $\mathbb{Z}/n\mathbb{Z}$, for proper subrings it would return a wrong result. Furthermore, it is not required that the argument is a ring; in fact this method is applicable also to the additive group formed by all elements in the family, provided that it knows to contain the whole family.

Analogously, we install methods to compute the size, a random element, and the units of full rings $\mathbb{Z}/n\mathbb{Z}$.

Example

```
gap> InstallMethod( Random,
>    "for full collection Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
>    R -> MyZmodnZObj( ElementsFamily( FamilyObj(R) ),
>                    Random( [ 0 .. Size( R ) - 1 ] ) ) );
gap>
gap> InstallMethod( Size,
>    "for full ring Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
>    R -> ElementsFamily( FamilyObj(R) )!.modulus );
gap>
gap> InstallMethod( Units,
>    "for full ring Z/nZ",
>    [     CategoryCollections( IsMyZmodnZObj )
>      and IsWholeFamily and IsRing ],
>    function( R )
>    local F;
>    F:= ElementsFamily( FamilyObj( R ) );
>    return List( PrimeResidues( Size(R) ), x -> MyZmodnZObj( F, x ) );
>    end );
```

The `Units` (56.5.2) method has the disadvantage that the result is returned as a list (in fact this list is also strictly sorted). We could improve the implementation by returning the units as a group; if we do not want to take the full list of elements as generators, we can use the function `GeneratorsPrimeResidues` (15.2.4).

Example

```
gap> InstallMethod( Units,
>    "for full ring Z/nZ",
>    [     CategoryCollections( IsMyZmodnZObj )
>      and IsWholeFamily and IsRing ],
>    function( R )
>    local G, gens;
>
>    gens:= GeneratorsPrimeResidues( Size( R ) ).generators;
>    if not IsEmpty( gens ) and gens[ 1 ] = 1 then
>      gens:= gens{ [ 2 .. Length( gens ) ] };
>    fi;
```

```
>    gens:= Flat( gens ) * One( R );
>    return GroupByGenerators( gens, One( R ) );
>    end );
```

Each ring $\mathbb{Z}/n\mathbb{Z}$ is finite, and we could install a method that returns `true` when `IsFinite` (30.4.2) is called with $\mathbb{Z}/n\mathbb{Z}$ as argument. But we can do this more elegantly via installing a *logical implication*.

Example

```
gap> InstallTrueMethod( IsFinite,
>    CategoryCollections( IsMyZmodnZObj ) and IsDomain );
```

In effect, every domain that consists of elements in `IsMyZmodnZObj` will automatically store that it is finite, even if `IsFinite` (30.4.2) is not called for it.

81.4 Compatibility of Residue Class Rings with Prime Fields

The above implementation of residue classes and residue class rings has at least two disadvantages. First, if p is a prime then the ring $\mathbb{Z}/p\mathbb{Z}$ is in fact a field, but the return values of `MyZmodnZ` are never regarded as fields because they are not in the category `IsMagmaWithInversesIfNonzero` (35.1.3). Second, and this makes the example really interesting, there are already elements of finite prime fields implemented in GAP, and we may want to identify them with elements in $\mathbb{Z}/p\mathbb{Z}$.

To be more precise, elements of finite fields in GAP lie in the category `IsFFE` (59.1.1), and there is already a representation, `IsInternalRep`, of these elements via discrete logarithms. The aim of this section is to make `IsMyModulusRep` an alternative representation of elements in finite prime fields.

Note that this is only one step towards the desired compatibility. Namely, after having a second representation of elements in finite prime fields, we may wish that the function `GF` (59.3.2) (which is the usual function to create finite fields in GAP) is able to return `MyZmodnZ( p )` when `GF( p )` is called for a prime p. Moreover, then we have to decide about a default representation of elements in `GF( p )` for primes p for which both representations are available. Of course we can force the new representation by explicitly calling `MyZmodnZ` and `MyZmodnZObj` whenever we want, but it is not a priori clear in which situation which representation is preferable.

The same questions will occur when we want to implement a new representation for non-prime fields. The steps of this implementation will be the same as described in this chapter, and we will have to achieve compatibility with both the internal representation of elements in small finite fields and the representation `IsMyModulusRep` of elements in arbitrary prime fields.

But let us now turn back to the task of this section. We first adjust the setup of the declaration part of the previous section, and then repeat the installations with suitable modifications.

(We should start a new GAP session for that, otherwise GAP will complain that the objects to be declared are already bound; additionally, the methods installed above may be not compatible with the ones we want.)

Example

```
gap> DeclareCategory( "IsMyZmodnZObj", IsScalar );
gap>
gap> DeclareCategory( "IsMyZmodnZObjNonprime", IsMyZmodnZObj );
gap>
gap> DeclareSynonym( "IsMyZmodpZObj", IsMyZmodnZObj and IsFFE );
gap>
```

```
gap> DeclareRepresentation( "IsMyModulusRep", IsPositionalObjectRep, [ 1 ] );
gap>
gap> DeclareGlobalFunction( "MyZmodnZObj" );
gap>
gap> DeclareGlobalFunction( "MyZmodnZ" );
```

As in the previous section, all (newly introduced) elements of rings $\mathbb{Z}/n\mathbb{Z}$ lie in the category `IsMyZmodnZObj`. But now we introduce two subcategories, namely `IsMyZmodnZObjNonprime` for all elements in rings $\mathbb{Z}/n\mathbb{Z}$ where n is not a prime, and `IsMyZmodpZObj` for elements in finite prime fields. All objects in the latter are automatically known to lie in the category `IsFFE` (59.1.1) of finite field elements.

It would be reasonable if also those internally represented elements in the category `IsFFE` (59.1.1) that do in fact lie in a prime field would also lie in the category `IsMyZmodnZObj` (and thus in fact in `IsMyZmodpZObj`). But this cannot be achieved because internally represented finite field elements do in general not store whether they lie in a prime field.

As for the implementation part, again let us start with the definitions of `MyZmodnZObj` and `MyZmodnZ`.

Example

```
gap> InstallGlobalFunction( MyZmodnZObj, function( Fam, residue )
>    if IsFFEFamily( Fam ) then
>      return Objectify( NewType( Fam, IsMyZmodpZObj
>                                   and IsMyModulusRep ),
>                    [ residue mod Characteristic( Fam ) ] );
>    else
>      return Objectify( NewType( Fam, IsMyZmodnZObjNonprime
>                                   and IsMyModulusRep ),
>                    [ residue mod Fam!.modulus ] );
>    fi;
> end );

gap> InstallGlobalFunction( MyZmodnZ, function( n )
>    local F, R;
>
>    if not ( IsInt( n ) and IsPosRat( n ) ) then
>      Error( "<n> must be a positive integer" );
>    elif IsPrimeInt( n ) then
>      # Construct the family of element objects of our field.
>      F:= FFEFamily( n );
>      # Make the domain.
>      R:= FieldOverItselfByGenerators( [ MyZmodnZObj( F, 1 ) ] );
>      SetIsPrimeField( R, true );
>    else
>      # Construct the family of element objects of our ring.
>      F:= NewFamily( Concatenation( "MyZmod", String( n ), "Z" ),
>                     IsMyZmodnZObjNonprime );
>      # Install the data.
>      F!.modulus:= n;
>      # Make the domain.
>      R:= RingWithOneByGenerators( [ MyZmodnZObj( F, 1 ) ] );
>      SetIsWholeFamily( R, true );
>      SetName( R, Concatenation( "(Integers mod ",String(n),")" ) );
```

```
>    fi;
>
>    # Return the ring resp. field.
>    return R;
> end );
```

Note that the result of MyZmodnZ with a prime as argument is a field that does not contain the whole family of its elements, since all finite field elements of a fixed characteristic lie in the same family. Further note that we cannot expect a family of finite field elements to have a component `modulus`, so we use `Characteristic` (31.10.1) to get the modulus. Requiring that `Fam!.modulus` works also if `Fam` is a family of finite field elements would violate the rule that an extension of GAP should not force changes in existing code, in this case code dealing with families of finite field elements.

Example

```
gap> InstallMethod( PrintObj,
>    "for element in Z/nZ (ModulusRep)",
>    [ IsMyZmodnZObjNonprime and IsMyModulusRep ],
>    function( x )
>    Print( "( ", x![1], " mod ", FamilyObj(x)!.modulus, " )" );
>    end );
gap>
gap> InstallMethod( PrintObj,
>    "for element in Z/pZ (ModulusRep)",
>    [ IsMyZmodpZObj and IsMyModulusRep ],
>    function( x )
>    Print( "( ", x![1], " mod ", Characteristic(x), " )" );
>    end );
gap>
gap> InstallMethod( \=,
>    "for two elements in Z/nZ (ModulusRep)",
>    IsIdenticalObj,
>    [ IsMyZmodnZObj and IsMyModulusRep,
>      IsMyZmodnZObj and IsMyModulusRep ],
>    function( x, y ) return x![1] = y![1]; end );
```

The above method to check equality is independent of whether the arguments have a prime or nonprime modulus, so we installed it for arguments in `IsMyZmodnZObj`. Now we install also methods to compare objects in `IsMyZmodpZObj` with the "old" finite field elements.

Example

```
gap> InstallMethod( \=,
>    "for element in Z/pZ (ModulusRep) and internal FFE",
>    IsIdenticalObj,
>    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
>    function( x, y )
>    return DegreeFFE( y ) = 1 and x![1] = IntFFE( y );
>    end );
gap>
gap> InstallMethod( \=,
>    "for internal FFE and element in Z/pZ (ModulusRep)",
>    IsIdenticalObj,
>    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
```

```
>    function( x, y )
>    return DegreeFFE( x ) = 1 and IntFFE( x ) = y![1];
>    end );
```

The situation with the operation < is more difficult. Of course we are free to define the comparison of objects in `IsMyZmodnZObjNonprime`, but for the finite field elements, the comparison must be compatible with the predefined comparison of the "old" finite field elements. The definition of the < comparison of internally represented finite field elements can be found in Chapter 59. In situations where the documentation does not provide the required information, one has to look it up in the GAP code; for example, the comparison in our case can be found in the appropriate source code file of the GAP kernel.

Example

```
gap> InstallMethod( \<,
>    "for two elements in Z/nZ (ModulusRep, nonprime)",
>    IsIdenticalObj,
>    [ IsMyZmodnZObjNonprime and IsMyModulusRep,
>      IsMyZmodnZObjNonprime and IsMyModulusRep ],
>    function( x, y ) return x![1] < y![1]; end );
gap>
gap> InstallMethod( \<,
>    "for two elements in Z/pZ (ModulusRep)",
>    IsIdenticalObj,
>    [ IsMyZmodpZObj and IsMyModulusRep,
>      IsMyZmodpZObj and IsMyModulusRep ],
>    function( x, y )
>    local p, r;      # characteristic and primitive root
>    if x![1] = 0 then
>      return y![1] <> 0;
>    elif y![1] = 0 then
>      return false;
>    else
>      p:= Characteristic( x );
>      r:= PrimitiveRootMod( p );
>      return LogMod( x![1], r, p ) < LogMod( y![1], r, p );
>    fi;
>    end );
gap>
gap> InstallMethod( \<,
>    "for element in Z/pZ (ModulusRep) and internal FFE",
>    IsIdenticalObj,
>    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
>    function( x, y )
>    return x![1] * One( y ) < y;
>    end );
gap>
gap> InstallMethod( \<,
>    "for internal FFE and element in Z/pZ (ModulusRep)",
>    IsIdenticalObj,
>    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
>    function( x, y )
>    return x < y![1] * One( x );
>    end );
```

Now we install the same methods for the arithmetic operations \+ (31.12.1), ZeroOp (31.10.3), AdditiveInverseOp (31.10.9), \-, * (31.12.1), and OneOp (31.10.2) as in the previous section, without listing them below. Also the same Int (14.2.3) method is installed for objects in IsMyZmodnZObj. Note that it is compatible with the definition of Int (14.2.3) for finite field elements. And of course the same method for \mod (31.12.1) is installed.

We have to be careful, however, with the methods for InverseOp (31.10.8), \/ (31.12.1), and \^ (31.12.1). These methods and the missing methods for arithmetic operations with one argument in IsMyModulusRep and the other in IsInternalRep are given below.

Example

```
gap> InstallMethod( \+,
>    "for element in Z/pZ (ModulusRep) and internal FFE",
>    IsIdenticalObj,
>    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
>    function( x, y ) return x![1] + y; end );
gap>
gap> InstallMethod( \+,
>    "for internal FFE and element in Z/pZ (ModulusRep)",
>    IsIdenticalObj,
>    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
>    function( x, y ) return x + y![1]; end );
gap>
gap> InstallMethod( \*,
>    "for element in Z/pZ (ModulusRep) and internal FFE",
>    IsIdenticalObj,
>    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
>    function( x, y ) return x![1] * y; end );
gap>
gap> InstallMethod( \*,
>    "for internal FFE and element in Z/pZ (ModulusRep)",
>    IsIdenticalObj,
>    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
>    function( x, y ) return x * y![1]; end );
gap>
gap> InstallMethod( InverseOp,
>    "for element in Z/nZ (ModulusRep, nonprime)",
>    [ IsMyZmodnZObjNonprime and IsMyModulusRep ],
>    function( x )
>    local residue;
>    residue:= QuotientMod( 1, x![1], FamilyObj(x)!.modulus );
>    if residue <> fail then
>      residue:= MyZmodnZObj( FamilyObj(x), residue );
>    fi;
>    return residue;
>    end );
gap>
gap> InstallMethod( InverseOp,
>    "for element in Z/pZ (ModulusRep)",
>    [ IsMyZmodpZObj and IsMyModulusRep ],
>    function( x )
>    local residue;
>    residue:= QuotientMod( 1, x![1], Characteristic( FamilyObj(x) ) );
>    if residue <> fail then
```

```
>      residue:= MyZmodnZObj( FamilyObj(x), residue );
>    fi;
>    return residue;
>    end );
```

The operation DegreeFFE (59.2.1) is defined for finite field elements, we need a method for objects in IsMyZmodpZObj. Note that we need not require IsMyModulusRep since no access to representation dependent data occurs.

Example

```
gap> InstallMethod( DegreeFFE,
>    "for element in Z/pZ",
>    [ IsMyZmodpZObj ],
>    z -> 1 );
```

The methods for Enumerator (30.3.2), Random (30.7.1), Size (30.4.6), and Units (56.5.2), that we had installed in the previous section had all assumed that their argument contains the whole family of its elements. So these methods make sense only for the nonprime case. For the prime case, there are already methods for these operations with argument a field.

Example

```
gap> InstallMethod( Enumerator,
>    "for full ring Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObjNonprime ) and IsWholeFamily ],
>    function( R )
>    local F;
>    F:= ElementsFamily( FamilyObj( R ) );
>    return List( [ 0 .. Size( R ) - 1 ], x -> MyZmodnZObj( F, x ) );
>    end );
gap>
gap> InstallMethod( Random,
>    "for full ring Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObjNonprime ) and IsWholeFamily ],
>    R -> MyZmodnZObj( ElementsFamily( FamilyObj( R ) ),
>                    Random( [ 0 .. Size( R ) - 1 ] ) ) );
gap>
gap> InstallMethod( Size,
>    "for full ring Z/nZ",
>    [ CategoryCollections( IsMyZmodnZObjNonprime ) and IsWholeFamily ],
>    R -> ElementsFamily( FamilyObj( R ) )!.modulus );
gap>
gap> InstallMethod( Units,
>    "for full ring Z/nZ",
>    [     CategoryCollections( IsMyZmodnZObjNonprime )
>      and IsWholeFamily and IsRing ],
>    function( R )
>    local G, gens;
>
>    gens:= GeneratorsPrimeResidues( Size( R ) ).generators;
>    if not IsEmpty( gens ) and gens[ 1 ] = 1 then
>      gens:= gens{ [ 2 .. Length( gens ) ] };
>    fi;
>    gens:= Flat( gens ) * One( R );
```

```
>    return GroupByGenerators( gens, One( R ) );
>    end );
gap>
gap> InstallTrueMethod( IsFinite,
>    CategoryCollections( IsMyZmodnZObjNonprime ) and IsDomain );
```

81.5 Further Improvements in Implementing Residue Class Rings

There are of course many possibilities to improve the implementation.

With the setup as described above, subsequent calls `MyZmodnZ( n )` with the same *n* yield incompatible rings in the sense that elements of one ring cannot be added to elements of an other one. The solution for this problem is to keep a global list of all results of `MyZmodnZ` in the current GAP session, and to return the stored values whenever possible. Note that this approach would admit `PrintObj` (6.3.5) methods that produce GAP readable output.

One can improve the `Units` (56.5.2) method for the full ring in such a way that a group is returned and not only a list of its elements; then the result of `Units` (56.5.2) can be used, e. g., as input for the operation `SylowSubgroup` (39.13.1).

To make computations more efficient, one can install methods for `\-`, `\/` (31.12.1), and `\^` (31.12.1); one reason for doing so may be that this avoids the unnecessary construction of the additive or multiplicative inverse, or of intermediate powers.

Example

```
InstallMethod( \-, "two elements in Z/nZ (ModulusRep)", ... );
InstallMethod( \-, "Z/nZ-obj. (ModulusRep) and integer", ... );
InstallMethod( \-, "integer and Z/nZ-obj. (ModulusRep)", ... );
InstallMethod( \-, "Z/pZ-obj. (ModulusRep) and internal FFE", ... );
InstallMethod( \-, "internal FFE and Z/pZ-obj. (ModulusRep)", ... );
InstallMethod( \*, "Z/nZ-obj. (ModulusRep) and integer", ... );
InstallMethod( \*, "integer and Z/nZ-obj. (ModulusRep)", ... );
InstallMethod( \/, "two Z/nZ-objs. (ModulusRep, nonprime)", ... );
InstallMethod( \/, "two Z/pZ-objs. (ModulusRep)", ... );
InstallMethod( \/, "Z/nZ-obj. (ModulusRep) and integer", ... );
InstallMethod( \/, "integer and Z/nZ-obj. (ModulusRep)", ... );
InstallMethod( \/, "Z/pZ-obj. (ModulusRep) and internal FFE", ... );
InstallMethod( \/, "internal FFE and Z/pZ-obj. (ModulusRep)", ... );
InstallMethod( \^, "Z/nZ-obj. (ModulusRep, nonprime) & int.", ... );
InstallMethod( \^, "Z/pZ-obj. (ModulusRep), and integer", ... );
```

The call to `NewType` (79.8.1) in `MyZmodnZObj` can be avoided by storing the required type, e.g., in the family. But note that it is *not* admissible to take the type of an existing object as first argument of `Objectify` (79.9.1). For example, suppose two objects in `IsMyZmodnZObj` shall be added. Then we must not use the type of one of the arguments in a call of `Objectify` (79.9.1), because the argument may have knowledge that is not correct for the result of the addition. One may think of the property `IsOne` (31.10.5) that may hold for both arguments but certainly not for their sum.

For comparing two objects in `IsMyZmodpZObj` via "<", we had to install a quite expensive method because of the compatibility with the comparison of finite field elements that did already exist. In fact GAP supports finite fields with elements represented via discrete logarithms only up to a given size. So in principle we have the freedom to define a cheaper comparison via "<" for objects in `IsMyZmodpZObj` if the modulus is large enough. This is possible by introducing two categories

`IsMyZmodpZObjSmall` and `IsMyZmodpZObjLarge`, which are subcategories of `IsMyZmodpZObj`, and to install different \< (31.11.1) methods for pairs of objects in these categories.

Chapter 82

An Example – Designing Arithmetic Operations

In this chapter, we give a –hopefully typical– example of extending GAP by new objects with prescribed arithmetic operations (for a simple approach that may be useful to get started though does not permit to exploit all potential features, see also `ArithmeticElementCreator` (80.9.1)).

82.1 New Arithmetic Operations vs. New Objects

A usual procedure in mathematics is the definition of new operations for given objects; here are a few typical examples. The Lie bracket defines an interesting new multiplicative structure on a given (associative) algebra. Forming a group ring can be viewed as defining a new addition for the elements of the given group, and extending the multiplication to sums of group elements in a natural way. Forming the exterior algebra of a given vector space can be viewed as defining a new multiplication for the vectors in a natural way.

GAP does *not* support such a procedure. The main reason for this is that in GAP, the multiplication in a group, a ring etc. is always written as $*$, and the addition in a vector space, a ring etc. is always written as $+$. Therefore it is not possible to define the Lie bracket as a "second multiplication" for the elements of a given algebra; in fact, the multiplication in Lie algebras in GAP is denoted by $*$. Analogously, constructing the group ring as sketched above is impossible if an addition is already defined for the elements; note the difference between the usual addition of matrices and the addition in the group ring of a matrix group! (See Chapter 65 for an example.) Similarly, there is already a multiplication defined for row vectors (yielding the standard scalar product), hence these vectors cannot be regarded as elements of the exterior algebra of the space.

In situations such as the ones mentioned above, GAP's way to deal with the structures in question is the following. Instead of defining *new* operations for the *given* objects, *new* objects are created to which the *given* arithmetic operations $*$ and $+$ are then made applicable.

With this construction, matrix Lie algebras consist of matrices that are different from the matrices with associative multiplication; technically, the type of a matrix determines how it is multiplied with other matrices (see `IsMatrix` (24.2.1)). A matrix with the Lie bracket as its multiplication can be created with the function `LieObject` (64.1.1) from a matrix with the usual associative multiplication.

Group rings (more general: magma rings, see Chapter 65) can be constructed with `FreeMagmaRing` (65.1.1) from a coefficient ring and a group. The elements of the group are not

contained in such a group ring, one has to use an embedding map for creating a group ring element that corresponds to a given group element.

It should be noted that the GAP approach to the construction of Lie algebras from associative algebras is generic in the sense that all objects in the filter `IsLieObject` (64.1.2) use the same methods for their addition, multiplication etc., by delegating to the "underlying" objects of the associative algebra, no matter what these objects actually are. Analogously, also the construction of group rings is generic.

82.2 Designing new Multiplicative Objects

The goal of this section is to implement objects with a prescribed multiplication. Let us assume that we are given a field F, and that we want to define a new multiplication $*$ on F that is given by $a*b = ab - a - b + 2$; here ab denotes the ordinary product in F.

By the discussion in Section 82.1, we know that we cannot define a new multiplication on F itself but have to create new objects.

We want to distinguish these new objects from all other GAP objects, in order to describe for example the situation that two of our objects shall be multiplied. This distinction is made via the *type* of the objects. More precisely, we declare a new *filter*, a function that will return `true` for our new objects, and `false` for all other GAP objects. This can be done by calling `DeclareFilter` (79.18.5), but since our objects will know about the value already when they are constructed, the filter can be created with `DeclareCategory` (79.18.1) or `NewCategory` (79.1.1).

Example

```
DeclareCategory( "IsMyObject", IsObject );
```

The idea is that the new multiplication will be installed only for objects that "lie in the category `IsMyObject`".

The next question is what internal data our new objects store, and how they are accessed. The easiest solution is to store the "underlying" object from the field F. GAP provides two general possibilities how to store this, namely record-like and list-like structures (for examples, see 79.10 and 79.11). We decide to store the data in a list-like structure, at position 1. This *representation* is declared as follows.

Example

```
DeclareRepresentation( "IsMyObjectListRep", IsPositionalObjectRep, [ 1 ] );
```

Of course we can argue that this declaration is superfluous because *all* objects in the category `IsMyObject` will be represented this way; it is possible to proceed like that, but often (in more complicated situations) it turns out to be useful that several representations are available for "the same element".

For creating the type of our objects, we need to specify to which *family* (see 13.1) the objects shall belong. For the moment, we need not say anything about relations to other GAP objects, thus the only requirement is that all new objects lie in the *same* family; therefore we create a *new* family. Also we are not interested in properties that some of our objects have and others do not have, thus we need only one type, and store it in a global variable.

Example

```
MyType:= NewType( NewFamily( "MyFamily" ),
                  IsMyObject and IsMyObjectListRep );
```

The next step is to write a function that creates a new object. It may look as follows.

Example

```
MyObject:= val -> Objectify( MyType, [ Immutable( val ) ] );
```

Note that we store an *immutable copy* of the argument in the returned object; without doing so, for example if the argument would be a mutable matrix then the corresponding new object would be changed whenever the matrix is changed (see 12.6 for more details about mutability).

Having entered the above GAP code, we can create some of our objects.

Example

```
gap> a:= MyObject( 3 );  b:= MyObject( 5 );
<object>
<object>
gap> a![1];  b![1];
3
5
```

But clearly a lot is missing. Besides the fact that the desired multiplication is not yet installed, we see that also the way how the objects are printed is not satisfactory.

Let us improve the latter first. There are two GAP functions `View` (6.3.3) and `Print` (6.3.4) for showing objects on the screen. `View` (6.3.3) is thought to show a short and human readable form of the object, and `Print` (6.3.4) is thought to show a not necessarily short form that is GAP readable whenever this makes sense. We decide to show `a` as *3* by `View` (6.3.3), and to show the construction `MyObject( 3 )` by `Print` (6.3.4); the methods are installed for the underlying operations `ViewObj` (6.3.5) and `PrintObj` (6.3.5).

Example

```
InstallMethod( ViewObj,
    "for object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    function( obj )
    Print( "<", obj![1], ">" );
    end );

InstallMethod( PrintObj,
    "for object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    function( obj )
    Print( "MyObject( ", obj![1], " )" );
    end );
```

This is the result of the above installations.

Example

```
gap> a; Print( a, "\n" );
<3>
MyObject( 3 )
```

And now we try to install the multiplication.

Example

```
InstallMethod( \*,
    "for two objects in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep,
```

```
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return MyObject( a![1] * b![1] - a![1] - b![1] + 2 );
    end );
```

When we enter the above code, GAP runs into an error. This is due to the fact that the operation * (31.12.1) is declared for two arguments that lie in the category IsMultiplicativeElement (31.14.10). One could circumvent the check whether the method matches the declaration of the operation, by calling InstallOtherMethod (78.2.2) instead of InstallMethod (78.2.1). But it would make sense if our objects would lie in IsMultiplicativeElement (31.14.10), for example because some generic methods for objects with multiplication would be available then, such as powering by positive integers via repeated squaring. So we want that IsMyObject implies IsMultiplicativeElement (31.14.10). The easiest way to achieve such implications is to use the implied filter as second argument of the DeclareCategory (79.18.1) call; but since we do not want to start anew, we can also install the implication afterwards.

Example

```
InstallTrueMethod( IsMultiplicativeElement, IsMyObject );
```

Afterwards, installing the multiplication works without problems. Note that MyType and therefore also a and b are *not* affected by this implication, so we construct them anew.

Example

```
gap> MyType:= NewType( NewFamily( "MyFamily" ),
>                      IsMyObject and IsMyObjectListRep );;
gap> a:= MyObject( 3 );;  b:= MyObject( 5 );;
gap> a*b;  a^27;
<9>
<134217729>
```

Powering the new objects by negative integers is not possible yet, because GAP does not know how to compute the inverse of an element a, say, which is defined as the unique element a' such that both aa' and $a'a$ are "the unique multiplicative neutral element that belongs to a".

And also this neutral element, if it exists, cannot be computed by GAP in our current situation. It does, however, make sense to ask for the multiplicative neutral element of a given magma, and for inverses of elements in the magma.

But before we can form domains of our objects, we must define when two objects are regarded as equal; note that this is necessary in order to decide about the uniqueness of neutral and inverse elements. In our situation, equality is defined in the obvious way. For being able to form sets of our objects, also an ordering via \< (31.11.1) is defined for them.

Example

```
InstallMethod( \=,
    "for two objects in 'IsMyObject'",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return a![1] = b![1];
    end );

InstallMethod( \<,
```

```
    "for two objects in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return a![1] < b![1];
    end );
```

Let us look at an example. We start with finite field elements because then the domains are finite, hence the generic methods for such domains will have a chance to succeed.

Example

```
gap> a:= MyObject( Z(7) );
<Z(7)>
gap> m:= Magma( a );
<magma with 1 generators>
gap> e:= MultiplicativeNeutralElement( m );
<Z(7)^2>
gap> elms:= AsList( m );
[ <Z(7)>, <Z(7)^2>, <Z(7)^5> ]
gap> ForAll( elms, x -> ForAny( elms, y -> x*y = e and y*x = e ) );
true
gap> List( elms, x -> First( elms, y -> x*y = e and y*x = e ) );
[ <Z(7)^5>, <Z(7)^2>, <Z(7)> ]
```

So a multiplicative neutral element exists, in fact all elements in the magma `m` are invertible. But what about the following.

Example

```
gap> b:= MyObject( Z(7)^0 );  m:= Magma( a, b );
<Z(7)^0>
<magma with 2 generators>
gap> elms:= AsList( m );
[ <Z(7)^0>, <Z(7)>, <Z(7)^2>, <Z(7)^5> ]
gap> e:= MultiplicativeNeutralElement( m );
<Z(7)^2>
gap> ForAll( elms, x -> ForAny( elms, y -> x*y = e and y*x = e ) );
false
gap> List( elms, x -> b * x );
[ <Z(7)^0>, <Z(7)^0>, <Z(7)^0>, <Z(7)^0> ]
```

Here we found a multiplicative neutral element, but the element `b` does not have an inverse. If an addition would be defined for our elements then we would say that `b` behaves like a zero element.

When we started to implement the new objects, we said that we wanted to define the new multiplication for elements of a given field F. In principle, the current implementation would admit also something like `MyObject( 2 ) * MyObject( Z(7) )`. But if we decide that our initial assumption holds, we may define the identity and the inverse of the object `<a>` as `<2*e>` and `<a/(a-e)>`, respectively, where `e` is the identity element in F and `/` denotes the division in F; note that the element `<e>` is not invertible, and that the above definitions are determined by the multiplication defined for our objects. Further note that after the installations shown below, also `One( MyObject( 1 ) )` is defined.

(For technical reasons, we do not install the intended methods for the attributes `One` (31.10.2) and `Inverse` (31.10.8) but for the operations `OneOp` (31.10.2) and `InverseOp` (31.10.8). This is because

for certain kinds of objects –mainly matrices– one wants to support a method to compute a *mutable* identity or inverse, and the attribute needs only a method that takes this object, makes it immutable, and then returns this object. As stated above, we only want to deal with immutable objects, so this distinction is not really interesting for us.)

A more interesting point to note is that we should mark our objects as likely to be invertible, since we add the possibility to invert them. Again, this could have been part of the declaration of `IsMyObject`, but we may also formulate an implication for the existing category.

Example

```
InstallTrueMethod( IsMultiplicativeElementWithInverse, IsMyObject );

InstallMethod( OneOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( 2 * One( a![1] ) ) );

InstallMethod( InverseOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( a![1] / ( a![1] - One( a![1] ) ) ) );
```

Now we can form groups of our (nonzero) elements.

Example

```
gap> MyType:= NewType( NewFamily( "MyFamily" ),
>                   IsMyObject and IsMyObjectListRep );;
gap>
gap> a:= MyObject( Z(7) );
<Z(7)>
gap> b:= MyObject( 0*Z(7) );  g:= Group( a, b );
<0*Z(7)>
<group with 2 generators>
gap> Size( g );
6
```

We are completely free to define an *addition* for our elements, a natural one is given by `<a> + <b> = <a+b-1>`. As we did for the multiplication, we first change `IsMyObject` such that the additive structure is also known.

Example

```
InstallTrueMethod( IsAdditiveElementWithInverse, IsMyObject );
```

Next we install the methods for the addition, and those to compute the additive neutral element and the additive inverse.

Example

```
InstallMethod( \+,
    "for two objects in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return MyObject( a![1] + b![1] - 1 );
    end );
```

```
InstallMethod( ZeroOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( One( a![1] ) ) );

InstallMethod( AdditiveInverseOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( a![1] / ( a![1] - One( a![1] ) ) ) );
```

Let us try whether the addition works.

Example

```
gap> MyType:= NewType( NewFamily( "MyFamily" ),
>                  IsMyObject and IsMyObjectListRep );;
gap> a:= MyObject( Z(7) );;  b:= MyObject( 0*Z(7) );;
gap> m:= AdditiveMagma( a, b );
<additive magma with 2 generators>
gap> Size( m );
7
```

Similar as installing a multiplication automatically makes powering by integers available, multiplication with integers becomes available with the addition.

Example

```
gap> 2 * a;
<Z(7)^5>
gap> a+a;
<Z(7)^5>
gap> MyObject( 2*Z(7)^0 ) * a;
<Z(7)>
```

In particular we see that this multiplication does *not* coincide with the multiplication of two of our objects, that is, an integer *cannot* be used as a shorthand for one of the new objects in a multiplication.

(It should be possible to create a *field* with the new multiplication and addition. Currently this fails, due to missing methods for computing several kinds of generators from field generators, for computing the characteristic in the case that the family does not know this in advance, for checking with `AsField` (58.1.9) whether a domain is in fact a field, for computing the closure as a field.)

It should be emphasized that the mechanism described above may be not suitable for the situation that one wants to consider many different multiplications “on the same set of objects”, since the installation of a new multiplication requires the declaration of at least one new filter and the installation of several methods. But the design of GAP is not suitable for such dynamic method installations.

Turning this argument the other way round, the implementation of the new arithmetics defined by the above multiplication and addition is available for any field F, one need not repeat it for each field one is interested in.

Similar to the above situation, the construction of a magma ring RM from a coefficient ring R and a magma M is implemented only once, since the definition of the arithmetic operations depends only on the given multiplication of M and not on M itself. So the addition is not implemented for the elements in M or –more precisely– for an isomorphic copy. In some sense, the addition is installed “for the multiplication”, and as mentioned in Section 82.1, there is only one multiplication `\*` (31.12.1) in GAP.

Chapter 83

Library Files

This chapter describes some of the conventions used in the GAP library files. These conventions are intended as a help on how to read library files and how to find information in them. So everybody is recommended to follow these conventions, although they do not prescribe a compulsory programming style –GAP itself will not bother with the formatting of files.

Filenames have traditionally GAP adhered to the 8+3 convention (to make it possible to use the same filenames even on a MS-DOS file system) and been in lower case (systems that do not recognize lower case in file names will convert them automatically to upper case). It is no longer so important to adhere to these conventions, but at the very least filenames should adhere to a 16+5 convention, and be distinct even after identifying upper and lower case. Directory names of packages, however, *must* be in lower case (the `LoadPackage` (76.2.1) command assumes this).

83.1 File Types

The GAP library consists of the following types of files, distinguished by their suffixes:

`.g` Files which contain parts of the "inner workings" of GAP. These files usually do not contain mathematical functionality, except for providing links to kernel functions.

`.gd` Declaration files. These files contain declarations of all categories, attributes, operations, and global functions. These files also contain the operation definitions in comments.

`.gi` Implementation files. These files contain all installations of methods and global functions. Usually declarations of representations are also considered to be part of the implementation and are therefore found in the `.gi` files.

As a rule of thumb, all `.gd` files are read in before the `.gi` files are read. Therefore a `.gi` file usually may use any operation or global function (it has been declared before), and no care has to be taken towards the order in which the `.gi` files are read.

83.2 Finding Implementations in the Library

For a concretely given function, you can use `FilenameFunc` (5.1.4) and `StartlineFunc` (5.1.5) for finding the file where this function is defined, and the line in this file where the definition of this

function starts. This does not work for arbitrary functions, see Section FilenameFunc (5.1.4) for the restrictions.

If you are interested in getting the function which implements a method for specific arguments, you can use 7.2.1. If FilenameFunc (5.1.4) does not work for this method then setting the print level of 7.2.1 higher will give you the installation string for this method, which can be used for searching in library files.

To find the occurrence of functions, methods, function names, and installation strings in the library, one can use the grep tool under UNIX. To find a function, search for the function name in the .gd files; as global variables are usually declared only once, only few files will show up. The function installation is likely to occur in the corresponding .gi file.

To find a method from the known operation name and the installation string, search for the string "Method(" (this catches both InstallMethod (78.2.1) and InstallOtherMethod (78.2.2)) and the installation string or the operation name.

The following tools from the GAP package Browse can be used for accessing the code of functions.

- BrowseGapMethods (**Browse: BrowseGapMethods see BrowseGapData**) shows an overview of GAP's operations and methods, and allows one to navigate through the files that contain the implementations of the methods, using a pager.
- BrowseProfile (**Browse: BrowseProfile**) shows profiling results (similar to DisplayProfile (7.7.8)) and allows one to navigate through the files that contain the implementations of the functions that were actually used, using a pager.

83.3 Undocumented Variables

For several global variables in GAP, no information is available via the help system (see Section (**Tutorial: Help**), for a quick overview of the help system, or Chapter 2, for details). There are various reasons for "hiding" a variable from the user; namely, the variable may be regarded as of minor importance (for example, it may be a function called by documented GAP functions that first compute many input parameters for the undocumented function), or it belongs to a part of GAP that is still experimental in the sense that the meaning of the variable has not yet been fixed or even that it is not clear whether the variable will vanish in a more developed version.

As a consequence, it is dangerous to use undocumented variables because they are not guaranteed to exist or to behave the same in future versions of GAP.

Conversely, for *documented* variables, the definitions in the GAP manual can be relied on for future GAP versions (unless they turn out to be erroneous); if the GAP developers find that some piece of minor, but documented functionality is an insurmountable obstacle to important developments, they may make the smallest possible incompatible change to the functionality at the time of a major release. However, in any such case it will be announced clearly in the GAP Forum what has been changed and why.

So on the one hand, the developers of GAP want to keep the freedom of changing undocumented GAP code. On the other hand, users may be interested in using undocumented variables.

In this case, whenever you write GAP code involving undocumented variables, and want to make sure that this code will work in future versions of GAP, you may ask at support@gap-system.org for documentation about the variables in question. The GAP developers then decide whether these variables shall be documented or not, and if yes, what the definitions shall be.

In the former case, the new documentation is added to the GAP manual, this means that from then on, this definition is protected against changes.

In the latter case (which may occur for example if the variables in question are still experimental), you may add the current values of these variables to your private code if you want to be sure that nothing will be broken later due to changes in GAP.

Chapter 84

Interface to the GAP Help System

In this chapter we describe which information the help system needs about a manual book and how to tell it this information. The code which implements this interface can be found in `lib/helpbase.gi`.

If you are intending to use a documentation format that is already used by some other help book you probably don't need to know anything from this chapter. However, if you want to create a new format and make it available to GAP then hopefully you will find the necessary information here.

The basic idea of the help system is as follows: One tells GAP a directory which contains a file `manual.six`, see 84.1. When the GAP help is asked something about this book it reads in some basic information from the file `manual.six`: strings like section headers, function names, and index entries to be searched by the online help; information about the available formats of this book like text, html, dvi, and pdf; the actual files containing the documentation, corresponding section numbers, and page numbers: and so on. See 84.2 for a description of the format of the `manual.six` file.

It turns out that there is almost no restriction on the format of the `manual.six` file, except that it must provide a string, say `"myownformat"` which identifies the format of the help book. Then the basic actions on a help book are delegated by the help system to handler functions stored in a record `HELP_BOOK_HANDLER.myownformat`. See 84.3 for information which functions must be provided by the handler and what they are supposed to do. The main work to teach GAP to use a new document format is to write these handler functions and to produce an appropriate `manual.six` file.

84.1 Installing and Removing a Help Book

84.1.1 HELP_ADD_BOOK

▷ `HELP_ADD_BOOK(`*`short, long, dir`*`)` (function)

This command tells GAP that in directory *`dir`* (given as either a string describing the path relative to the GAP root directory `GAPInfo.RootPaths[1]` or as directory object) contains the basic information about a help book. The string *`short`* is used as an identifying name for that book by the online help. The string *`long`* should be a short explanation of the content of the book. Both strings together should easily fit on a line, since they are displayed with `?books`.

It is possible to reinstall a book with different strings *`short`*, *`long`*; (for example, documentation of a not-loaded GAP package indicates this in the string *`short`* and if you later load the package, GAP quietly changes the string *`short`* as it reinstalls its documentation).

The only condition necessary to make the installation of a book *valid* is that the directory *dir* must contain a file manual.six. The next section explains how this file must look.

84.1.2 HELP_REMOVE_BOOK

▷ HELP_REMOVE_BOOK(*short*) (function)

This command tells GAP not to use the help book with identifying name *short* any more. The book can be re-installed using HELP_ADD_BOOK (84.1.1).

84.2 The manual.six File

If a manual.six file for a help book is not in the format of the gapmacro.tex macros, explained in the document "The gapmacro.tex Manual Format" (see the file gap4r5/doc/gapmacrodoc.pdf from the tools archive etc/tools.tar.gz which should be unpacked using the script etc/install-tools.sh), the first non-empty line of manual.six must be of the form

#SIXFORMAT *myownformat*

where *myownformat* is an identifying string for this format. The reading of the (remainder of the) file is then delegated to the function HELP_BOOK_HANDLER.*myownformat*.ReadSix which must exist. Thus there are no further regulations for the format of the manual.six file, other that what you yourself impose. If such a line is missing then it is assumed that the manual.six file complies with the gapmacro.tex documentation format which is the default format.

Section 84.3 explains how the return value of HELP_BOOK_HANDLER.*myownformat*.ReadSix should look like and which further function should be contained in HELP_BOOK_HANDLER.*myownformat*.

84.3 The Help Book Handler

For each document format *myownformat* there must be a record HELP_BOOK_HANDLER.*myownformat* of functions with the following names and functionality.

An implementation example of such a set of handler functions is the default format, which is the format name used for the gapmacro.tex documentation format, and this is contained in the file lib/helpdef.gi.

The package GAPDoc (see Chapter (**GAPDoc: Introduction and Example**)) also defines a format (as it should) which is called: GapDocGAP (the case *is* significant).

As you can see by the above two examples, the name for a document format can be anything, but it should be in some way meaningful.

ReadSix(*stream*)

For an input text stream *stream* to a manual.six file, this must return a record *info* which has at least the following two components: bookname which is the short identifying name of the help book, and entries. Here *info*.entries must be a list with one entry per search string (which can be a section header, function name, index entry, or whatever seems sensible to be searched for matching a help query). A *match* for the GAP help is a pair (*info*, *i*) where *i* refers to an index for the list *info*.entries and this corresponds to a certain position in the document. There is one further regulation for the format of the entries of *info*.entries.

They must be lists and the first element of such a list must be a string which is printed by GAP for example when several matches are found for a query (so it should essentially be the string which is searched for the match, except that it may contain upper and lower case letters or some markup). There may be other components in *info* which are needed by the functions below, but their names and formats are not prescribed. The *stream* argument is typically generated using InputTextFile (10.5.1), e.g.

Example

```
gap> dirs := DirectoriesLibrary( "doc/ref" );;
gap> file := Filename( dirs, "manual.six" );;
gap> stream := InputTextFile( file );;
```

ShowChapters(*info*)

This must return a text string or list of text lines which contains the chapter headers of the book *info*.bookname.

ShowSection(*info*)

This must return a text string or list of text lines which contains the section (and chapter) headers of the book *info*.bookname.

SearchMatches(*info*, *topic*, *frombegin*)

This function must return a list of indices of *info*.entries for entries which match the search string *topic*. If *frombegin* is true then those parts of *topic* which are separated by spaces should be considered as the beginnings of words to decide the matching. It *frombegin* is false, a substring search should be performed. The string *topic* can be assumed to be already normalized (transformed to lower case, and whitespace normalized). The function must return a list with two entries [exact, match] where exact is the list of indices for exact matches and match a list of indices of the remaining matches.

MatchPrevChap(*info*, *i*)

This should return the match [*info*, j] which points to the beginning of the chapter containing match [*info*, *i*], respectively to the beginning of the previous chapter if [*info*, *i*] is already the beginning of a chapter. (Corresponds to ?<<.)

MatchNextChap(*info*, *i*)

Like the previous function except that it should return the match for the beginning of the next chapter. (Corresponds to ?>>.)

MatchPrev(*info*, *i*)

This should return the previous section (or appropriate portion of the document). (Corresponds to ?<.)

MatchNext(*info*, *i*)

Like the previous function except that it should return the next section (or appropriate portion of the document). (Corresponds to ?>.)

HelpData(*info*, *i*, *type*)

This returns for match [*info*, *i*] some data whose format depends on the string *type*, or fail if these data are not available. The values of *type* which currently must be handled and the corresponding result format are described in the list below.

The HELP_BOOK_HANDLER.*myownformat*.HelpData function must recognize the following values of the *type* argument.

"text"
: This must return a corresponding text string in a format which can be fed into the Pager, see Pager (2.4.1).

"url"
: If the help book is available in HTML format this must return an URL as a string (Probably a file:// URL containing a label for the exact start position in that file). Otherwise it returns fail.

"dvi"
: If the help book is available in dvi-format this must return a record of form rec(file := *filename*, page := *pagenumber*). Otherwise it returns fail.

"pdf"
: Same as case "dvi", but for the corresponding pdf-file.

"secnr"
: This must return a pair like [[3,3,1], "3.3.1"] which gives the section number as chapter number, section number, subsection number triple and a corresponding string (a chapter itself is encoded like [[4,0,0], "4."]). Useful for cross-referencing between help books.

84.4 Introducing new Viewer for the Online Help

To introduce a new viewer for the online help, one should extend the global record HELP_VIEWER_INFO (84.4.1), the structure of which is explained below.

84.4.1 HELP_VIEWER_INFO

▷ HELP_VIEWER_INFO (global variable)

The record HELP_VIEWER_INFO (84.4.1) contains one component for each help viewer. Each such component is a record with two components: .type and .show.

The component .type refers to one of the types recognized by the HelpData handler function explained in the previous section (currently one of "text", "url", "dvi", or "pdf").

The component .show is a function which gets as input the result of a corresponding HelpData handler call, if it was not fail. This function has to perform the actual display of the data. (E.g., by calling a function like Pager (2.4.1) or by starting up an external viewer program.)

Chapter 85

Function-Operation-Attribute Triples

GAP is eager to maintain information that it has gathered about an object, possibly by lengthy calculations. The most important mechanism for information maintenance is the automatic storage and look-up that takes place for *attributes*; and this was already mentioned in section (**Tutorial: Attributes**). In this chapter we will describe further mechanisms that allow storage of results that are not values of attributes.

The idea which is common to all sections is that certain operations, which are not themselves attributes, have an attribute associated with them. To automatically delegate tasks to the attribute, GAP knows, in addition to the *operation* and the *attributes* also a *function*, which is "wrapped around" the other two. This "wrapper function" is called by the user and decides whether to call the operation or the attribute or possibly both. The whole *f*unction-*o*peration-*a*ttribute triple (or *FOA triple*) is set up by a single GAP command which writes the wrapper function and already installs some methods, e.g., for the attribute to fall back on the operation. The idea is then that subsequent methods, which perform the actual computation, are installed only for the operation, whereas the wrapper function remains unaltered, and in general no additional methods for the attribute are required either.

85.1 Key Dependent Operations

85.1.1 KeyDependentOperation

▷ `KeyDependentOperation(`*`name, dom-req, key-req, key-test`*`)` (function)

There are several functions that require as first argument a domain, e.g., a group, and as second argument something much simpler, e.g., a prime. `SylowSubgroup` (39.13.1) is an example. Since its value depends on two arguments, it cannot be an attribute, yet one would like to store the Sylow subgroups once they have been computed.

The idea is to provide an attribute of the group, called `ComputedSylowSubgroups`, and to store the groups in this list. The name implies that the value of this attribute may change in the course of a GAP session, whenever a newly-computed Sylow subgroup is put into the list. Therefore, this is a *mutable attribute* (see 79.3). The list contains primes in each bound odd position and a corresponding Sylow subgroup in the following even position. More precisely, if `p = ComputedSylowSubgroups( G )[ even - 1 ]` then `ComputedSylowSubgroups( G )[ even ]` holds the value of `SylowSubgroup( G, p )`. The pairs are sorted in increasing order of `p`, in particular at most one Sylow `p` subgroup of `G` is stored for each prime `p`. This attribute value is maintained by the function `SylowSubgroup`

(39.13.1), which calls the operation SylowSubgroupOp(*G*, *p*) to do the real work, if the prime *p* cannot be found in the list. So methods that do the real work should be installed for SylowSubgroupOp and not for SylowSubgroup (39.13.1).

The same mechanism works for other functions as well, e.g., for PCore (39.11.3), but also for HallSubgroup (39.13.3), where the second argument is not a prime but a set of primes.

KeyDependentOperation declares the two operations and the attribute as described above, with names *name*, *name*Op, and Computed*name*s. *dom-req* and *key-req* specify the required filters for the first and second argument of the operation *name*Op, which are needed to create this operation with DeclareOperation (79.18.6). *dom-req* is also the required filter for the corresponding attribute Computed*name*s. The fourth argument *key-test* is in general a function to which the second argument *info* of *name*(*D*, *info*) will be passed. This function can perform tests on *info*, and raise an error if appropriate.

For example, to set up the three objects SylowSubgroup (39.13.1), SylowSubgroupOp, ComputedSylowSubgroups together, the declaration file lib/grp.gd contains the following line of code.

Example

```
KeyDependentOperation( "SylowSubgroup", IsGroup, IsPosInt, "prime" );
```

In this example, *key-test* has the value "prime", which is silently replaced by a function that tests whether its argument is a prime.

Example

```
gap> s4 := Group((1,2,3,4),(1,2));;
gap> SylowSubgroup( s4, 5 );;  ComputedSylowSubgroups( s4 );
[ 5, Group(()) ]
gap> SylowSubgroup( s4, 2 );;  ComputedSylowSubgroups( s4 );
[ 2, Group([ (3,4), (1,4)(2,3), (1,3)(2,4) ]), 5, Group(()) ]
```

Example

```
gap> SylowSubgroup( s4, 6 );
Error, SylowSubgroup: <p> must be a prime called from
<compiled or corrupted call value>  called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Thus the prime test need not be repeated in the methods for the operation SylowSubgroupOp (which are installed to do the real work). Note that no methods need be installed for SylowSubgroup (39.13.1) and ComputedSylowSubgroups. If a method is installed with InstallMethod (78.2.1) for a wrapper operation such as SylowSubgroup (39.13.1) then a warning is signalled provided the InfoWarning (7.4.7) level is at least 1. (Use InstallMethod (78.2.1) in order to suppress the warning.)

85.2 In Parent Attributes

85.2.1 InParentFOA

▷ InParentFOA(*name*, *super*, *sub*, *AorP*) (function)

This section describes how you can add new "in parent attributes" (see 31.8 and 31.7). As an example, we describe how Index (39.3.2) and its related functions are implemented.

There are two operations Index (39.3.2) and IndexOp, and an attribute IndexInParent. They are created together as shown below, and after they have been created, methods need be installed only for IndexOp. In the creation process, IndexInParent already gets one default method installed (in addition to the usual system getter of each attribute, see 13.5), namely D -> IndexOp(Parent(D), D).

The operation Index (39.3.2) proceeds as follows.

- If it is called with the two arguments *super* and *sub*, and if HasParent(*sub*) and IsIdenticalObj(*super*, Parent(*sub*)) are true, IndexInParent is called with argument *sub*, and the result is returned.
- Otherwise, IndexOp is called with the same arguments that Index (39.3.2) was called with, and the result is returned.

(Note that it is in principle possible to install even Index (39.3.2) and IndexOp methods for a number of arguments different from two, with InstallOtherMethod (78.2.2), see 79.3).

The call of InParentFOA declares the operations and the attribute as described above, with names *name*, *name*Op, and *name*InParent. *super-req* and *sub-req* specify the required filters for the first and second argument of the operation *name*Op, which are needed to create this operation with DeclareOperation (79.18.6). *sub-req* is also the required filter for the corresponding attribute *name*InParent; note that HasParent (31.7.1) is *not* required for the argument *U* of *name*InParent, because even without a parent stored, Parent(*U*) is legal, meaning *U* itself (see 31.7). The fourth argument must be DeclareProperty (79.18.4) if *name*InParent takes only boolean values (for example in the case IsNormalInParent), and DeclareAttribute (79.18.3) otherwise.

For example, to set up the three objects Index (39.3.2), IndexOp, and IndexInParent together, the declaration file lib/domain.gd contains the following line of code.

Example

```
InParentFOA( "Index", IsGroup, IsGroup, DeclareAttribute );
```

Note that no methods need be installed for Index (39.3.2) and IndexInParent.

85.3 Operation Functions

Chapter 41 and, in particular, the Section 41.1 explain that certain operations such as 41.4), besides their usual usage with arguments *G*, *D*, and *opr*, can also be applied to an external set (*G*-set), in which case they can be interpreted as attributes. Moreover, they can also be interpreted as attributes for permutation groups, meaning the natural action on the set of its moved points.

The definition of 41.4 says that a method should be a function with arguments *G*, *D*, *gens*, *oprs*, and *opr*, as in the case of the operation ExternalSet (41.12.2) when specified via *gens* and *oprs* (see 41.12). All other syntax variants allowed for 41.4 (e.g., leaving out *gens* and *oprs*) are handled by default methods.

The default methods for 41.4 support the following behaviour.

1. If the only argument is an external set *xset* and the attribute tester HasOrbits(*xset*) returns true, the stored value of that attribute is returned.

2. If the only argument is an external set *xset* and the attribute value is not known, the default arguments are obtained from the data of *xset*.

3. If *gens* and *oprs* are not specified, *gens* is set to Pcgs(*G*) if CanEasilyComputePcgs(*G*) is true, and to GeneratorsOfGroup(*G*) otherwise; *oprs* is set to *gens*.

4. The default value of *opr* is OnPoints (41.2.1).

5. In the case of an operation of a permutation group *G* on MovedPoints(*G*) via OnPoints (41.2.1), if the attribute tester HasOrbits(*G*) returns true, the stored attribute value is returned.

6. The operation is called as *result* := Orbits(*G*, *D*, *gens*, *oprs*, *opr*).

7. In the case of an external set *xset* or a permutation group *G* in its natural action, the attribute setter is called to store *result*.

8. *result* is returned.

The declaration of operations that match the above pattern is done as follows.

85.3.1 OrbitsishOperation

▷ OrbitsishOperation(*name, reqs, usetype, AorP*) (function)

declares an attribute op, with name *name*. The second argument *reqs* specifies the list of required filters for the usual (five-argument) methods that do the real work.

If the third argument *usetype* is true, the function call op(xset) will –if the value of op for xset is not yet known– delegate to the five-argument call of op with second argument xset rather than with D. This allows certain methods for op to make use of the type of xset, in which the types of the external subsets of xset and of the external orbits in xset are stored. (This is used to avoid repeated calls of NewType (79.8.1) in functions like ExternalOrbits(xset), which call ExternalOrbit(xset, pnt) for several values of pnt.)

For property testing functions such as IsTransitive (41.10.1), the fourth argument *AorP* must be NewProperty (79.3.2), otherwise it must be NewAttribute (79.3.1); in the former case, a property is returned, in the latter case an attribute that is not a property.

For example, to set up the operation Orbits (41.4), the declaration file lib/oprt.gd contains the following line of code:

Example

```
OrbitsishOperation( "Orbits", OrbitsishReq, false, NewAttribute );
```

The global variable OrbitsishReq contains the standard requirements

Example

```
OrbitsishReq := [ IsGroup, IsList,
                  IsList,
                  IsList,
                  IsFunction ];
```

which are usually entered in calls to OrbitsishOperation.

The new operation, e.g., Orbits (41.4), can be called either as Orbits(*xset*) for an external set *xset*, or as Orbits(*G*) for a permutation group *G*, meaning the orbits on the moved points of *G* via OnPoints (41.2.1), or as

Orbits(*G*, *Omega*[, *gens*, *acts*][, *act*]),

with a group *G*, a domain or list *Omega*, generators *gens* of *G*, and corresponding elements *acts* that act on *Omega* via the function *act*; the default of *gens* and *acts* is a list of group generators of *G*, the default of *act* is OnPoints (41.2.1).

Only methods for the five-argument version need to be installed for doing the real work. (And of course methods for one argument in case one wants to define a new meaning of the attribute.)

85.3.2 OrbitishFO

▷ OrbitishFO(*name, reqs, famrel, usetype, realenum*) (function)

is used to create operations like Orbit (41.4.1). This function is analogous to OrbitsishOperation (85.3.1), but for operations *orbish* like Orbit(*G*, *Omega*, *pnt*). Since the return values of these operations depend on the additional argument *pnt*, there is no associated attribute.

The call of OrbitishFO declares a wrapper function and its operation, with names *name* and *name*Op.

The second argument *reqs* specifies the list of required filters for the operation *name*Op.

The third argument *famrel* is used to test the family relation between the second and third argument of *name*(*G*, *D*, *pnt*). For example, *famrel* is IsCollsElms in the case of Orbit (41.4.1) because *pnt* must be an element of *D*. Similarly, in the call Blocks(*G*, *D*, *seed*), *seed* must be a subset of *D*, and the family relation must be IsIdenticalObj (12.5.1).

The fourth argument *usetype* serves the same purpose as in the case of OrbitsishOperation (85.3.1). *usetype* can also be an attribute, such as BlocksAttr or MaximalBlocksAttr. In this case, if only one of the two arguments *Omega* and *pnt* is given, blocks with no seed are computed, they are stored as attribute values according to the rules of OrbitsishOperation (85.3.1).

If the 5th argument is set to true, the action for an external set should use the enumerator, otherwise it uses the HomeEnumerator (41.12.5) value. This will make a difference for external orbits as part of a larger domain.

85.3.3 Example: Orbit and OrbitOp

For example, to setup the function Orbit (41.4.1) and its operation OrbitOp, the declaration file lib/oprt.gd contains the following line of code:

Example

```
OrbitishFO( "Orbit", OrbitishReq, IsCollsElms, false, false );
```

The variable OrbitishReq contains the standard requirements

Example

```
OrbitishReq  := [ IsGroup, IsList, IsObject,
                  IsList,
                  IsList,
                  IsFunction ];
```

which are usually entered in calls to OrbitishFO (85.3.2).

The relation test via *famrel* is used to provide a uniform construction of the wrapper functions created by OrbitishFO (85.3.2), in spite of the different syntax of the specific functions. For example, Orbit (41.4.1) admits the calls Orbit(*G*, *D*, *pnt*, *opr*) and Orbit(*G*, *pnt*, *opr*), i.e., the second argument *D* may be omitted; Blocks (41.11.1) admits the calls Blocks(*G*, *D*, *seed*, *opr*) and Blocks(*G*, *D*, *opr*), i.e., the third argument may be omitted. The translation to the appropriate call of OrbitOp or BlocksOp, for either operation with five or six arguments, is handled via *famrel*.

As a consequence, there must not only be methods for OrbitOp with the six arguments corresponding to OrbitishReq, but also methods for only five arguments (i.e., without *D*). Plenty of examples are contained in the implementation file lib/oprt.gi.

In order to handle a few special cases (currently Blocks (41.11.1) and MaximalBlocks (41.11.2)), also the following form of OrbitishFO (85.3.2) is supported.

OrbitishFO(*name*, *reqs*, *famrel*, *attr*)

The functions in question depend upon an argument *seed*, so they cannot be regarded as attributes. However, they are most often called without giving *seed*, meaning "choose any minimal resp. maximal block system". In this case, the result can be stored as the value of the attribute *attr* that was entered as fourth argument of OrbitishFO (85.3.2). This attribute is considered by a call Blocks(*G*, *D*, *opr*) (i.e., without *seed*) in the same way as Orbits (41.4) considers OrbitsAttr.

To set this up, the declaration file lib/oprt.gd contains the following lines:

Example

```
DeclareAttribute( "BlocksAttr", IsExternalSet );
OrbitishFO( "Blocks",
    [ IsGroup, IsList, IsList,
      IsList,
      IsList,
      IsFunction ], IsIdenticalObj, BlocksAttr, true );
```

And this extraordinary FOA triple works as follows:

Example

```
gap> s4 := Group((1,2,3,4),(1,2));;
gap> Blocks( s4, MovedPoints(s4), [1,2] );
[ [ 1, 2, 3, 4 ] ]
gap> Tester( BlocksAttr )( s4 );
false
gap> Blocks( s4, MovedPoints(s4) );
[ [ 1, 2, 3, 4 ] ]
gap> Tester( BlocksAttr )( s4 );  BlocksAttr( s4 );
true
[ [ 1, 2, 3, 4 ] ]
```

Chapter 86

Weak Pointers

This chapter describes the use of the kernel feature of *weak pointers*. This feature is primarily intended for use only in GAP internals, and should be used extremely carefully otherwise.

The GASMAN garbage collector is the part of the kernel that manages memory in the users workspace. It will normally only reclaim the storage used by an object when the object cannot be reached as a subobject of any GAP variable, or from any reference in the kernel. We say that any link to object *a* from object *b* "keeps object *a* alive", as long as *b* is alive. It is occasionally convenient, however, to have a link to an object which *does not keep it alive*, and this is a weak pointer. The most common use is in caches, and similar structures, where it is only necessary to remember how to solve problem *x* as long as some other link to *x* exists.

The following section 86.1 describes the semantics of the objects that contain weak pointers. Following sections describe the functions available to manipulate them.

86.1 Weak Pointer Objects

A *weak pointer object* is similar to a mutable plain list, except that it does not keep its subobjects alive during a garbage collection. From the GAP viewpoint this means that its entries may become unbound, apparently spontaneously, at any time. Considerable care is therefore needed in programming with such an object.

86.1.1 WeakPointerObj

▷ `WeakPointerObj(`*`list`*`)` (function)

`WeakPointerObj` returns a weak pointer object which contains the same subobjects as the list *`list`*, that is it returns a *shallow* weak copy of *`list`*.

Example

```
gap> w := WeakPointerObj( [ 1, , [2,3], fail, rec( a := 1) ] );
WeakPointerObj( [ 1, , [ 2, 3 ], fail, rec( a := 1 ) ] )
```

After some computations involving garbage collections (but not necessarily in the *first* garbage collection after the above assignment), GAP will notice that the list and the record stored in `w` are not referenced by other objects than `w`, and that therefore these entries may disappear.

Example

```
gap> GASMAN("collect");

... (perhaps more computations and garbage collections) ...

gap> GASMAN("collect");
gap> w;
WeakPointerObj( [ 1, , , fail ] )
```

Note that w has failed to keep its list and record subobjects alive during the garbage collections. Certain subobjects, such as small integers and elements of small finite fields, are not stored in the workspace, and so are not subject to garbage collection, while certain other objects, such as the Boolean values, are always reachable from global variables or the kernel and so are never garbage collected.

Subobjects reachable without going through a weak pointer object do not evaporate, as in:

Example

```
gap> w := WeakPointerObj( [ 1, , , fail ] );
WeakPointerObj( [ 1, , , fail ] )
gap> l := [1,2,3];;
gap> w[1] := l;;
gap> w;
WeakPointerObj( [ [ 1, 2, 3 ], , , fail ] )
gap> GASMAN("collect");
gap> w;
WeakPointerObj( [ [ 1, 2, 3 ], , , fail ] )
```

Note also that the global variables last, last2 and last3 will keep things alive –this can be confusing when debugging.

86.2 Low Level Access Functions for Weak Pointer Objects

86.2.1 SetElmWPObj

▷ SetElmWPObj(*wp, pos, val*) (function)
▷ UnbindElmWPObj(*wp, pos*) (function)
▷ ElmWPObj(*wp, pos*) (function)
▷ IsBoundElmWPObj(*wp, pos*) (function)
▷ LengthWPObj(*wp*) (function)

The functions SetElmWPObj and UnbindElmWPObj set and unbind entries in a weak pointer object.

The function ElmWPObj returns the element at position *pos* of the weak pointer object *wp*, if there is one, and fail otherwise. A return value of fail can thus arise either because (a) the value fail is stored at position *pos*, or (b) no value is stored at position *pos*. Since fail cannot vanish in a garbage collection, these two cases can safely be distinguished by a *subsequent* call to IsBoundElmWPObj, which returns true if there is currently a value bound at position *pos* of *wp* and false otherwise.

Note that it is *not* safe to write:

```
if IsBoundElmWPObj(w,i) then x:= ElmWPObj(w,i); fi;
```

and treat x as reliably containing a value taken from w, as a badly timed garbage collection could leave x containing fail. Instead use

```
x := ElmWPObj(w,i); if x <> fail or IsBoundElmWPObj(w,i) then .  .  ..
```

Here is an example.

Example

```
gap> w := WeakPointerObj( [ 1, , [2,3], fail, rec() ] );
WeakPointerObj( [ 1, , [ 2, 3 ], fail, rec(  ) ] )
gap> SetElmWPObj(w,5,[]);
gap> w;
WeakPointerObj( [ 1, , [ 2, 3 ], fail, [  ] ] )
gap> UnbindElmWPObj(w,1);
gap> w;
WeakPointerObj( [ , , [ 2, 3 ], fail, [  ] ] )
gap> ElmWPObj(w,3);
[ 2, 3 ]
gap> ElmWPObj(w,1);
fail
```

Now after some computations and garbage collections ...

Example

```
gap> 2;;3;;4;;GASMAN("collect"); # clear last, last2, last3
```

... we get the following.

Example

```
gap> ElmWPObj(w,3);
fail
gap> w;
WeakPointerObj( [ , , , fail ] )
gap> ElmWPObj(w,4);
fail
gap> IsBoundElmWPObj(w,3);
false
gap> IsBoundElmWPObj(w,4);
true
```

86.3 Accessing Weak Pointer Objects as Lists

Weak pointer objects are members of ListsFamily and the categories IsList (21.1.1) and IsMutable (12.6.2). Methods based on the low-level functions in the previous section, are installed for the list access operations, enabling them to be used as lists. However, it is *not recommended* that these be used in programming. They are supplied mainly as a convenience for interactive working, and may not be safe, since functions and methods for lists may assume that after IsBound(w[i]) returns true, access to w[i] is safe.

86.4 Copying Weak Pointer Objects

A ShallowCopy (12.7.1) method is installed, which makes a new weak pointer object containing the same objects as the original.

It is possible to apply `StructuralCopy` (12.7.2) to a weak pointer object, obtaining a new weak pointer object containing copies of the objects in the original. This *may not be safe* if a badly timed garbage collection occurs during copying.

Applying `Immutable` (12.6.3) to a weak pointer object produces an immutable plain list containing immutable copies of the objects contained in the weak pointer object. An immutable weak pointer object is a contradiction in terms.

86.5 The GASMAN Interface for Weak Pointer Objects

The key support for weak pointers is in the files `src/gasman.c` and `src/gasman.h`. This document assumes familiarity with the rest of the operation of GASMAN. A kernel type (tnum) of bags which are intended to act as weak pointers to their subobjects must meet three conditions. Firstly, the marking function installed for that tnum must use `MarkBagWeakly` for those subbags, rather than `MARK_BAG`. Secondly, before any access to such a subbag, it must be checked with `IS_WEAK_DEAD_BAG`. If that returns `true`, then the subbag has evaporated in a recent garbage collection and must not be accessed. Typically the reference to it should be removed. Thirdly, a *sweeping function* must be installed for that tnum which copies the bag, removing all references to dead weakly held subbags.

The files `src/weakptr.c` and `src/weakptr.h` use this interface to support weak pointer objects. Other objects with weak behaviour could be implemented in a similar way.

Chapter 87

More about Stabilizer Chains

This chapter contains some rather technical complements to the material handled in the chapters 42 and 43.

87.1 Generalized Conjugation Technique

The command `ConjugateGroup( G, p )` (see `ConjugateGroup` (39.2.6)) for a permutation group `G` with stabilizer chain equips its result also with a stabilizer chain, namely with the chain of `G` conjugate by `p`. Conjugating a stabilizer chain by a permutation `p` means replacing all the points which appear in the `orbit` components by their images under `p` and replacing every permutation `g` which appears in a `labels` or `transversal` component by its conjugate g^p. The conjugate g^p acts on the mapped points exactly as `g` did on the original points, i.e., $(pnt.p).g^p = (pnt.g).p$. Since the entries in the `translabels` components are integers pointing to positions of the `labels` list, the `translabels` lists just have to be permuted by `p` for the conjugated stabilizer. Then `generators` is reconstructed as `labels{ genlabels }` and `transversal{ orbit }` as `labels{ translabels{ orbit } }`.

This conjugation technique can be generalized. Instead of mapping points and permutations under the same permutation `p`, it is sometimes desirable (e.g., in the context of permutation group homomorphisms) to map the points with an arbitrary mapping *map* and the permutations with a homomorphism *hom* such that the compatibility of the actions is still valid: $map(pnt).hom(g) = map(pnt.g)$. (Of course the ordinary conjugation is a special case of this, with $map(pnt) = pnt.p$ and $hom(g) = g^p$.)

In the generalized case, the "conjugated" chain need not be a stabilizer chain for the image of *hom*, since the "preimage" of the stabilizer of $map(b)$ (where b is a base point) need not fix b, but only fixes the preimage $map^{-1}(map(b))$ setwise. Therefore the method can be applied only to one level and the next stabilizer must be computed explicitly. But if *map* is injective, we have $map(b).hom(g) = map(b)$ if and only if $b.g = b$, and if this holds, then $g = w(g_1,\ldots,g_n)$ is a word in the generators $g_1,\ldots,g_n$ of the stabilizer of b and $hom(g) =^* w(hom(g_1),\ldots,hom(g_n))$ is in the "conjugated" stabilizer. If, more generally, *hom* is a right inverse to a homomorphism φ (i.e., $\varphi(hom(g)) = g$ for all g), equality $*$ holds modulo the kernel of φ; in this case the "conjugated" chain can be made into a real stabilizer chain by extending each level with the generators of the kernel and appending a proper stabilizer chain of the kernel at the end. These special cases will occur in the algorithms for permutation group homomorphisms (see 40).

To "conjugate" the points (i.e., `orbit`) and permutations (i.e., `labels`) of the Schreier tree, a loop is set up over the `orbit` list constructed during the orbit algorithm, and for each vertex b with unique edge $a(l)b$ ending at b, the label l is mapped with *hom* and b with *map*. We assume that the `orbit`

list was built w.r.t. a certain ordering $<$ of the labels, where $l' < l$ means that every point in the orbit was mapped with l' before it was mapped with l. This shape of the `orbit` list is guaranteed if the Schreier tree is extended only by `AddGeneratorsExtendSchreierTree` (43.11.10), and it is then also guaranteed for the "conjugated" Schreier tree. (The ordering of the labels cannot be read from the Schreier tree, however.)

In the generalized case, it can happen that the edge $a(l)b$ bears a label l whose image is "old", i.e., equal to the image of an earlier label $l' < l$. Because of the compatibility of the actions we then have $map(b) = map(a).hom(l)^{-1} = map(a).hom(l')^{-1} = map(al'^{-1})$, so $map(b)$ is already equal to the image of the vertex al'^{-1}. This vertex must have been encountered before $b = al^{-1}$ because $l' < l$. We conclude that the image of a label can be "old" only if the vertex at the end of the corresponding edge has an "old" image, too, but then it need not be "conjugated" at all. A similar remark applies to labels which map under *hom* to the identity.

87.2 The General Backtrack Algorithm with Ordered Partitions

Section 43.12 describes the basic functions for a backtrack search. The purpose of this section is to document how the general backtrack algorithm is implemented in GAP and which parts you have to modify if you want to write your own backtrack routines.

87.2.1 Internal representation of ordered partitions

GAP represents an ordered partition as a record with the following components.

`points`
: a list of all points contained in the partition, such that the points of each cell from lie consecutively,

`cellno`
: a list whose *i*th entry is the number of the cell which contains the point *i*,

`firsts`
: a list such that `points[firsts[`*j*`]]` is the first point in `points` which is in cell *j*,

`lengths`
: a list of the cell lengths.

Some of the information is redundant, e.g., the `lengths` could also be read off the `firsts` list, but since this need not be increasing, it would require some searching. Similar for `cellno`, which could be replaced by a systematic search of `points`, keeping track of what cell is currently being traversed. With the above components, the *m*th cell of a partition *P* is expressed as *P*`.points{ [ `*P*`.firsts[`*m*`] .. `*P*`.firsts[`*m*`] + `*P*`.lengths[`*m*`] - 1 ] }`. The most important operations, however, to be performed upon *P* are the splitting of a cell and the reuniting of the two parts. Following the strategy of J. Leon, this is done as follows:

(1) The points which make up the cell that is to be split are sorted so that the ones that remain inside occupy positions `[ `*P*`.firsts[`*m*`] .. `*last*` ]` in the list *P*`.points` (for a suitable value of *last*).

(2) The points at positions `[ last + 1 .. P.firsts[m] + P.lengths[m] - 1 ]` will form the additional cell. For this new cell requires additional entries are added to the lists `P.firsts` (namely, `last+1`) and `P.lengths` (namely, `P.firsts[m] + P.lengths[m] - last - 1`).

(3) The entries of the sublist `P.cellno{ [ last+1 .. P.firsts[m] + P.lengths[m]-1 ] }` must be set to the number of the new cell.

(4) The entry `P.lengths[m]` must be reduced to `last - P.firsts[m] + 1`.

Then reuniting the two cells requires only the reversal of steps 2 to 4 above. The list `P.points` need not be rearranged.

87.2.2 Functions for setting up an R-base

This subsection explains some GAP functions which are local to the library file `lib/stbcbckt.gi` which contains the code for backtracking in permutation groups. They are mentioned here because you might find them helpful when you want to implement you own backtracking function based on the partition concept. An important argument to most of the functions is the R-base R, which you should regard as a black box. We will tell you how to set it up, how to maintain it and where to pass it as argument, but it is not necessary for you to know its internal representation. However, if you insist to learn the whole story: Here are the record components from which an R-base is made up:

`domain`
: the set Ω on which the group G operates

`base`
: the sequence $(a_1,\ldots,a_r)$ of base points

`partition`
: an ordered partition, initially Π_0, this will be refined to $\Pi_1,\ldots,\Pi_r$ during the backtrack algorithm

`where`
: a list such that a_i lies in cell number `where`$[i]$ of Π_i

`rfm` a list whose ith entry is a list of refinements which take Σ_i to Σ_{i+1}; the structure of a refinement is described below

`chain`
: a (copy of a) stabilizer chain for G (not if G is a symmetric group)

`fix` only if G is a symmetric group: a list whose i entry contains `Fixcells(` Π_i `)`

`level`
: initially equal to `chain`, this will be changed to chains for the stabilizers $G_{a_1\ldots a_i}$ for $i=1,\ldots,r$ during the backtrack algorithm; if G is a symmetric group, only the number of moved points is stored for each stabilizer

`lev` a list whose ith entry remembers the `level` entry for $G_{a_1\ldots a_{i-1}}$

`level2`, `lev2`
: a similar construction for a second group (used in intersection calculations), `false` otherwise. This second group H activated if the R-base is constructed as `EmptyRBase(` $[G,H],\Omega,\Pi_0$ `)` (if $G = H$, GAP sets `level2 = true` instead).

`nextLevel`
: this is described below

As our guiding example, we present code for the function `Centralizer` (35.4.4) which calculates the centralizer of an element g in the group G. (The real code is more general and has a few more subtleties.)

```
Pi_0 := TrivialPartition( omega );
R := EmptyRBase( G, omega, Pi_0 );
R.nextLevel := function( Pi, rbase )
local  fix, p, q, where;
NextRBasePoint( Pi, rbase );
fix := Fixcells( Pi );
for p  in fix  do
  q := p ^ g;
  where := IsolatePoint( Pi, q );
  if where <> false  then
    Add( fix, q );
    ProcessFixpoint( R, q );
    AddRefinement( R, "Centralizer", [ Pi.cellno[ p ], q, where ] );
    if Pi.lengths[ where ] = 1  then
      p := FixpointCellNo( Pi, where );
      ProcessFixpoint( R, p );
      AddRefinement( R, "ProcessFixpoint", [ p, where ] );
    fi;
  fi;
od;
end;

return PartitionBacktrack(
  G,
  c -> g ^ c = g,
  false,
  R,
  [ Pi_0, g ],
  L, R );
```

The list numbers below refer to the line numbers of the code above.

1. `omega` is the set on which `G` acts and `Pi_0` is the first member of the decreasing sequence of partitions mentioned in 43.12. We set `Pi_0 = omega`, which is constructed as `TrivialPartition( omega )`, but we could have started with a finer partition, e.g., into unions of g-cycles of the same length.

2. This statement sets up the R-base in the variable `R`.

3.-21.

These lines define a function `R.nextLevel` which is called whenever an additional member in the sequence `Pi_0` $\geq \Pi_1 \geq \ldots$ of partitions is needed. If Π_i does not yet contain enough base points in one-point cells, GAP will call `R.nextLevel(` Π_i, `R )`, and this function will choose a new base point a_{i+1}, refine Π_i to Π_{i+1} (thereby *changing* the first argument) and store all necessary information in `R`.

5. This statement selects a new base point a_{i+1}, which is not yet in a one-point cell of Π and still moved by the stabilizer $G_{a_1 \ldots a_i}$ of the earlier base points. If certain points of `omega` should be preferred as base point (e.g., because they belong to long cycles of g), a list of points starting with the most wanted ones, can be given as an optional third argument to `NextRBasePoint` (actually, this is done in the real code for `Centralizer` (35.4.4)).

6. `Fixcells(` Π `)` returns the list of points in one-point cells of Π (ordered as the cells are ordered in Π).

7. For every point $p \in fix$, if we know the image $p\hat{}g$ under $c \in C_G(e)$, we also know $(p\hat{}g)\hat{}c = (p\hat{}c)\hat{}g$. We therefore want to isolate these extra points in Π.

9. This statement puts point q in a cell of its own, returning in `where` the number of the cell of Π from which q was taken. If q was already the only point in its cell, `where = false` instead.

12. This command does the necessary bookkeeping for the extra base point q: It prescribes q as next base in the stabilizer chain for G (needed, e.g., in line 5) and returns `false` if q was already fixed the stabilizer of the earlier base points (and `true` otherwise; this is not used here). Another call to `ProcessFixpoint` like this was implicitly made by the function `NextRBasePoint` to register the chosen base point. By contrast, the point q was not chosen this way, so `ProcessFixpoint` must be called explicitly for q.

13. This statement registers the function which will be used during the backtrack search to perform the corresponding refinements on the "image partition" Σ_i (to yield the refined Σ_{i+1}). After choosing an image b_{i+1} for the base point a_{i+1}, GAP will compute $\Sigma_i \wedge (\{b_{i+1}\}, \Omega \setminus \{b_{i+1}\})$ and store this partition in I`.partition`, where I is a black box similar to R, but corresponding to the current "image partition" (hence it is an "R-image" in analogy to the R-base). Then GAP will call the function `Refinements.Centralizer( R, I, Pi.cellno[ p ], p, where )`, with the then current values of R and I, but where Π`.cellno`$[p]$, p, `where` still have the values they have at the time of this `AddRefinement` command. This function call will further refine I`.partition` to yield Σ_{i+1} as it is programmed in the function `Refinements.Centralizer`, which is described below. (The global variable `Refinements` is a record which contains all refinement functions for all backtracking procedures.)

14.-19.

If the cell from which q was taken out had only two points, we now have an additional one-point cell. This condition is checked in line 13 and if it is true, this extra fixpoint p is taken (line 15), processed like q before (line 16) and is then (line 17) passed to another refinement function `Refinements.ProcessFixpoint( R, I, p, where )`, which is also described below.

23.-29.

This command starts the backtrack search. Its result will be the centralizer as a subgroup of G. Its arguments are

24. the group we want to run through,

25. the property we want to test, as a GAP function,

26. `false` if we are looking for a subgroup, `true` in the case of a representative search (when the result would be one representative),

27. the R-base,

28. a list of data, to be stored in I`.data`, which has in position 1 the first member Σ_0 of the decreasing sequence of "image partitions" mentioned in 43.12. In the centralizer example, position 2 contains the element that is to be centralized. In the case of a representative search, i.e., a conjugacy test `g^c ?= h`, we would have h instead of g here, and possibly a Σ_0 different from Π_0 (e.g., a partition into unions of h-cycles of same length).

29. two subgroups $L \leq C_G(g)$ and $R \leq C_G(h)$ known in advance (we have $L = R$ in the centralizer case).

87.2.3 Refinement functions for the backtrack search

The last subsection showed how the refinement process leading from Π_i to Π_{i+1} is coded in the function R`.nextLevel`, this has to be executed once the base point a_{i+1}. The analogous refinement step from Σ_i to Σ_{i+1} must be performed for each choice of an image b_{i+1} for a_{i+1}, and it will depend on the corresponding value of $\Sigma_i \wedge (\{b_{i+1}\}, \Omega \setminus \{b_{i+1}\})$. But before we can continue our centralizer example, we must, for the interested reader, document the record components of the other black box I, as we did above for the R-base black box R. Most of the components change as GAP walks up and down the levels of the search tree.

`data`
: this will be mentioned below

`depth`
: the level i in the search tree of the current node Σ_i

`bimg`
: a list of images of the points in R`.base`

`partition`
: the partition Σ_i of the current node

`level`
: the stabilizer chain R`.lev`$[i]$ at the current level

`perm`
: a permutation mapping `Fixcells`(Π_i) to `Fixcells`(Σ_i); this implies mapping $(a_1, \ldots, a_i)$ to $(b_1, \ldots, b_i)$

`level2`**,** `perm2`
: a similar construction for the second stabilizer chain, `false` otherwise (and `true` if R`.level2 = true`)

As declared in the above code for `Centralizer` (35.4.4), the refinement is performed by the function `Refinement.Centralizer`$(R, I, \Pi\,.\,\texttt{cellno}[p], p, where)$. The functions in the record `Refinement` always take two additional arguments before the ones specified in the `AddRefinement` call (in line 13 above), namely the R-base R and the current value I of the "R-image". In our example, p is a fixpoint of $\Pi = \Pi_i \wedge (\{a_{i+1}\}, \Omega \setminus \{a_{i+1}\})$ such that $where = \Pi\,.\,\texttt{cellno}[p^g]$. The `Refinement` functions must return `false` if the refinement is unsuccessful (e.g., because it leads to Σ_{i+1} having different cell sizes from Π_{i+1}) and `true` otherwise. Our particular function looks like this.

```
Refinements.Centralizer := function( R, I, cellno, p, where )
local  Sigma, q;
Sigma := I.partition;
q := FixpointCellNo( Sigma, cellno ) ^ I.data[ 2 ];
return IsolatePoint( Sigma, q ) = where and ProcessFixpoint( I, p, q );
end;
```

The list numbers below refer to the line numbers of the code immediately above.

3. The current value of $\Sigma_i \wedge (\{b_{i+1}\}, \Omega \setminus \{b_{i+1}\})$ is always found in $I\,.\,\texttt{partition}$.

4. The image of the only point in cell number $cellno = \Pi_i\,.\,\texttt{cellno}[p]$ in Σ under $g = I\,.\,\texttt{data}[2]$ is calculated.

5. The function returns `true` only if the image q has the same cell number in Σ as p had in Π (i.e., *where*) and if q can be prescribed as an image for p under the coset of the stabilizer $G_{a_1 \ldots a_{i+1}}.c$ where $c \in G$ is an (already constructed) element mapping the earlier base points $a_1, \ldots, a_{i+1}$ to the already chosen images $b_1, \ldots, b_{i+1}$. This latter condition is tested by `ProcessFixpoint`(I, p, q) which, if successful, also does the necessary bookkeeping in I. In analogy to the remark about line 12 in the program above, the chosen image b_{i+1} for the base point a_{i+1} has already been processed implicitly by the function `PartitionBacktrack`, and this processing includes the construction of an element $c \in G$ which maps `Fixcells`(Π_i) to `Fixcells`(Σ_i) and a_{i+1} to b_{i+1}. By contrast, the extra fixpoints p and q in Π_{i+1} and Σ_{i+1} were not chosen automatically, so they require an explicit call of `ProcessFixpoint`, which replaces the element c by some $c'.c$ (with $c' \in G_{a_1 \ldots a_{i+1}}$) which in addition maps p to q, or returns `false` if this is impossible.

You should now be able to guess what `Refinements.ProcessFixpoint`$(R, I, p, where)$ does: it simply returns `ProcessFixpoint`$(I, p,$`FixpointCellNo`$(I\,.\,\texttt{partition}, where))$.

Summary.

When you write your own backtrack functions using the partition technique, you have to supply an R-base, including a component `nextLevel`, and the functions in the `Refinements` record which you need. Then you can start the backtrack by passing the R-base and the additional data (for the `data` component of the "R-image") to `PartitionBacktrack`.

87.2.4 Functions for meeting ordered partitions

A kind of refinement that occurs in particular in the normalizer calculation involves computing the meet of Π (cf. lines 6ff. above) with an arbitrary other partition Λ, not just with one point. To do this efficiently, GAP uses the following two functions.

`StratMeetPartition( `R, Π, Λ `[,` g `] )`
`MeetPartitionStrat( `R, I`{, }`Λ'`}[, {`g'`}]`, *strat* `)`

Such a `StratMeetPartition` command would typically appear in the function call R`.nextLevel(`Π, R`)` (during the refinement of Π_i to Π_{i+1}). This command replaces Π by $\Pi \wedge \Lambda$ (thereby *changing* the second argument) and returns a "meet strategy" *strat*. This is (for us) a black box which serves two purposes: First, it allows GAP to calculate faster the corresponding meet $\Sigma \wedge \Lambda'$, which must then appear in a `Refinements` function (during the refinement of Σ_i to Σ_{i+1}). It is faster to compute $\Sigma \wedge \Lambda'$ with the "meet strategy" of $\Pi \wedge \Lambda$ because if the refinement of Σ is successful at all, the intersection of a cell from the left hand side of the $\wedge$ sign with a cell from the right hand side must have the same size in both cases (and *strat* records these sizes, so that only non-empty intersections must be calculated for $\Sigma \wedge \Lambda'$). Second, if there is a discrepancy between the behaviour prescribed by *strat* and the behaviour observed when refining Σ, the refinement can immediately be abandoned.

On the other hand, if you only want to meet a partition Π with Λ for a one-time use, without recording a strategy, you can simply type `StratMeetPartition`(Π, Λ) as in the following example, which also demonstrates some other partition-related commands.

Example

```
gap> P := Partition( [[1,2],[3,4,5],[6]] );;  Cells( P );
[ [ 1, 2 ], [ 3, 4, 5 ], [ 6 ] ]
gap> Q := Partition( OnTuplesTuples( last, (1,3,6) ) );;  Cells( Q );
[ [ 3, 2 ], [ 6, 4, 5 ], [ 1 ] ]
gap> StratMeetPartition( P, Q );
[  ]
gap> # The ‘‘meet strategy’’ was not recorded, ignore this result.
gap> Cells( P );
[ [ 1 ], [ 5, 4 ], [ 6 ], [ 2 ], [ 3 ] ]
```

You can even say `StratMeetPartition`(Π, Δ) where Δ is simply a subset of Ω, it will then be interpreted as the partition $(\Delta, \Omega \setminus \Delta)$.

GAP makes use of the advantages of a "meet strategy" if the refinement function in `Refinements` contains a `MeetPartitionStrat` command where *strat* is the "meet strategy" calculated by `StratMeetPartition` before. Such a command replaces I`.partition` by its meet with Λ', again changing the argument I. The necessary reversal of these changes when backtracking from a node (and prescribing the next possible image for a base point) is automatically done by the function `PartitionBacktrack`.

In all cases, an additional argument g means that the meet is to be taken not with Λ, but instead with $\Lambda.g^{-1}$, where operation on ordered partitions is meant cellwise (and setwise on each cell). (Analogously for the primed arguments.)

Example

```
gap> P := Partition( [[1,2],[3,4,5],[6]] );;
gap> StratMeetPartition( P, P, (1,6,3) );;  Cells( P );
[ [ 1 ], [ 5, 4 ], [ 6 ], [ 2 ], [ 3 ] ]
```

Note that $P.(1,3,6) = Q$.

87.2.5 Avoiding multiplication of permutations

In the description of the last subsections, the backtrack algorithm constructs an element $c \in G$ mapping the base points to the prescribed images and finally tests the property in question for that element.

During the construction, c is obtained as a product of transversal elements from the stabilizer chain for G, and so multiplications of permutations are required for every c submitted to the test, even if the test fails (i.e., in our centralizer example, if `g^c<>g`). Even if the construction of c stops before images for all base points have been chosen, because a refinement was unsuccessful, several multiplications will already have been performed by (explicit or implicit) calls of `ProcessFixpoint`, and, actually, the general backtrack procedure implemented in GAP avoids this.

For this purpose, GAP does not actually multiply the permutations but rather stores all the factors of the product in a list. Specifically, instead of carrying out the multiplication in $c \mapsto c'.c$ mentioned in the comment to line 5 of the above program — where $c' \in G_{a_1 \ldots a_{i+1}}$ is a product of factorized inverse transversal elements, see 43.9 — GAP appends the list of these factorized inverse transversal elements (giving c') to the list of factors already collected for c. Here c' is multiplied from the left and is itself a product of *inverses* of strong generators of G, but GAP simply spares itself all the work of inverting permutations and stores only a "list of inverses", whose product is then $(c'.c)^{-1}$ (which is the new value of c^{-1}). The "list of inverses" is extended this way whenever `ProcessFixpoint` is called to improve c.

The product has to be multiplied out only when the property is finally tested for the element c. But it is often possible to delay the multiplication even further, namely until after the test, so that no multiplication is required in the case of an unsuccessful test. Then the test itself must be carried out with the factorized version of the element c. For this purpose, `PartitionBacktrack` can be passed its second argument (the property in question) in a different way, not as a single GAP function, but as a list like in lines 2–4 of the following alternative excerpt from the code for `Centralizer` (35.4.4).

```
return PartitionBacktrack( G,
  [ g, g,
  OnPoints,
  c -> c!.lftObj = c!.rgtObj ],
  false, R, [ Pi_0, g ], L, R );
```

The test for c to have the property in question is of the form $opr(left, c) = right$ where opr is an operation function as explained in 41.12. In other words, c passes the test if and only if it maps a "left object" to a "right object" under a certain operation. In the centralizer example, we have opr = `OnPoints` and $left = right = g$, but in a conjugacy test, we would have $right = h$.

2. Two first two entries (here g and g) are the values of $left$ and $right$.

3. The third entry (here `OnPoints` (41.2.1)) is the operation opr.

4. The fourth entry is the test to be performed upon the mapped left object $left$ and preimage of the right object $opr(right, c\text{\^{}-1})$. Here GAP operates with the inverse of c because this is the product of the permutations stored in the "list of inverses". The preimage of $right$ under c is then calculated by mapping $right$ with the factors of c^{-1} one by one, without the need to multiply these factors. This mapping of $right$ is automatically done by the `ProcessFixpoint` function whenever c is extended, the current value of $right$ is always stored in `c!.rgtObj`. When the test given by the fourth entry is finally performed, the element c has two components `c!.lftObj`$=$ $left$ and `c!.rgtObj`$= opr(right, c\text{\^{}-1})$, which must be used to express the desired relation as a function of c. In our centralizer example, we simply have to test whether they are equal.

87.3 Stabilizer Chains for Automorphisms Acting on Enumerators

This section describes a way of representing the automorphism group of a group as permutation group, following [Sim97]. The code however is not yet included in the GAP library.

In this section we present an example in which objects we already know (namely, automorphisms of solvable groups) are equipped with the permutation-like operations `^` and `/` for action on positive integers. To achieve this, we must define a new type of objects which behave like permutations but are represented as automorphisms acting on an enumerator. Our goal is to generalize the Schreier-Sims algorithm for construction of a stabilizer chain to groups of such new automorphisms.

87.3.1 An operation domain for automorphisms

The idea we describe here is due to C. Sims. We consider a group A of automorphisms of a group G, given by generators, and we would like to know its order. Of course we could follow the strategy of the Schreier-Sims algorithm (described in 43.6) for A acting on G. This would involve a call of `StabChainStrong( EmptyStabChain( [], One( A ) ), GroupGenerators( A ) )` where `StabChainStrong` is a function as the one described in the pseudo-code below:

```
StabChainStrong := function( S, newgens )
  Extend the Schreier tree of S with newgens.
  for sch  in  Schreier generators  do
    if not sch in S.stabilizer  then
      StabChainStrong( S.stabilizer, [ sch ] );
    fi;
  od;
end;
```

The membership test $sch \notin S$`.stabilizer` can be performed because the stabilizer chain of S`.stabilizer` is already correct at that moment. We even know a base in advance, namely any generating set for G. Fix such a generating set $(g_1,\ldots,g_d)$ and observe that this base is generally very short compared to the degree $|G|$ of the operation. The problem with the Schreier-Sims algorithm, however, is then that the length of the first basic orbit $g_1.A$ would already have the magnitude of $|G|$, and the basic orbits at deeper levels would not be much shorter. For the advantage of a short base we pay the high price of long basic orbits, since the product of the (few) basic orbit lengths must equal $|A|$. Such long orbits make the Schreier-Sims algorithm infeasible, so we have to look for a longer base with shorter basic orbits.

Assume that G is solvable and choose a characteristic series with elementary abelian factors. For the sake of simplicity we assume that $N < G$ is an elementary abelian characteristic subgroup with elementary abelian factor group G/N. Since N is characteristic, A also acts as a group of automorphisms on the factor group G/N, but of course not necessarily faithfully. To retain a faithful action, we let A act on the disjoint union G/N with G, and choose as base $(g_1N,\ldots,g_dN,g_1,\ldots,g_d)$. Now the first d basic orbits lie inside G/N and can have length at most $[G:N]$. Since the base points $g_1N,\ldots,g_dN$ form a generating set for G/N, their iterated stabilizer $A^{(d+1)}$ acts trivially on the factor group G/N, i.e., it leaves the cosets g_iN invariant. Accordingly, the next d basic orbits lie inside g_iN (for $i = 1,\ldots,d$) and can have length at most $|N|$.

Generalizing this method to a characteristic series $G = N_0 > N_1 > \ldots > N_l = \{1\}$ of length $l > 2$, we can always find a base of length $l.d$ such that each basic orbit is contained in a coset of a characteristic factor, i.e. in a set of the form g_iN_{j-1}/N_j (where g_i is one of the generators of G and

$1 \leq j \leq l$). In particular, the length of the basic orbits is bounded by the size of the corresponding characteristic factors. To implement a Schreier-Sims algorithm for such a base, we must be able to let automorphisms act on cosets of characteristic factors $g_i N_{j-1}/N_j$, for varying i and j. We would like to translate each such action into an action on $\{1,\ldots,[N_{j-1}:N_j]\}$, because then we need not enumerate the operation domain, which is the disjoint union of G/N_1, $G/N_2 \ldots G/N_l$, as a whole. Enumerating it as a whole would result in basic orbits like `orbit`$\subseteq \{1001,\ldots,1100\}$ with a `transversal` list whose first 1000 entries would be unbound, but still require 4 bytes of memory each (see 43.9).

Identifying each coset $g_i N_{j-1}/N_j$ into $\{1,\ldots,[N_{j-1}:N_j]\}$ of course means that we have to change the action of the automorphisms on every level of the stabilizer chain. Such flexibility is not possible with permutations because their effect on positive integers is "hardwired" into them, but we can install new operations for automorphisms.

87.3.2 Enumerators for cosets of characteristic factors

So far we have not used the fact that the characteristic factors are elementary abelian, but we will do so from here on. Our first task is to implement an enumerator (see `AsList` (30.3.8) and 21.23) for a coset of a characteristic factor in a solvable group G. We assume that such a coset gN/M is given by

(1) a pcgs for the group G (see `Pcgs` (45.2.1)), let $n =$`Length( `*pcgs*` )`;

(2) a range $range = [start..stop]$ indicating that $N = \langle pcgs\{[start..n]\}\rangle$ and $M = \langle pcgs\{[stop+1..n]\}\rangle$, i.e., the cosets of $pcgs\{range\}$ form a base for the vector space N/M;

(3) the representative g.

We first define a new representation for such enumerators and then construct them by simply putting these three pieces of data into a record object. The enumerator should behave as a list of group elements (representing cosets modulo M), consequently, its family will be the family of the *pcgs* itself.

Example

```
DeclareRepresentation( "IsCosetSolvableFactorEnumeratorRep", IsEnumerator,
    [ "pcgs", "range", "representative" ] );

EnumeratorCosetSolvableFactor := function( pcgs, range, g )
    return Objectify( NewType( FamilyObj( pcgs ),
                    IsCosetSolvableFactorEnumeratorRep ),
                    rec( pcgs := pcgs,
                        range := range,
                representative := g ) );
end;
```

The definition of the operations `Length` (21.17.5), `\[\]` (21.2.1) and `Position` (21.16.1) is now straightforward. The code has sometimes been abbreviated and is meant "cum grano salis", e.g., the declaration of the local variables has been left out.

Example

```
InstallMethod( Length, [ IsCosetSolvableFactorEnumeratorRep ],
    enum -> Product( RelativeOrdersPcgs( enum!.pcgs ){ enum!.range } ) );

InstallMethod( \[\], [ IsCosetSolvableFactorEnumeratorRep,
        IsPosRat and IsInt ],
    function( enum, pos )
```

```
    elm := ();
    pos := pos - 1;
    for i  in Reversed( enum!.range )  do
        p := RelativeOrderOfPcElement( enum!.pcgs, i );
        elm := enum!.pcgs[ i ] ^ ( pos mod p ) * elm;
        pos := QuoInt( pos, p );
    od;
    return enum!.representative * elm;
end );

InstallMethod( Position, [ IsCosetSolvableFactorEnumeratorRep,
        IsObject, IsZeroCyc ],
    function( enum, elm, zero )
    exp := ExponentsOfPcElement( enum!.pcgs,
                   LeftQuotient( enum!.representative, elm ) );
    pos := 0;
    for i  in enum!.range  do
        pos := pos * RelativeOrderOfPcElement( pcgs, i ) + exp[ i ];
    od;
    return pos + 1;
end );
```

87.3.3 Making automorphisms act on such enumerators

Our next task is to make automorphisms of the solvable group *pcgs*`!.group` act on $[1..\texttt{Length}(\mathit{enum})]$ for such an enumerator *enum*. We achieve this by introducing a new representation of automorphisms on enumerators and by putting the enumerator together with the automorphism into an object which behaves like a permutation. Turning an ordinary automorphism into such a special automorphism requires then the construction of a new object which has the new type. We provide an operation `PermOnEnumerator( model, aut )` which constructs such a new object having the same type as *model*, but representing the automorphism *aut*. So *aut* can be either an ordinary automorphism or one which already has an enumerator in its type, but perhaps different from the one we want (i.e. from the one in *model*).

Example

```
DeclareCategory( "IsPermOnEnumerator",
    IsMultiplicativeElementWithInverse and IsPerm );

DeclareRepresentation( "IsPermOnEnumeratorDefaultRep",
    IsPermOnEnumerator and IsAttributeStoringRep,
    [ "perm" ] );

DeclareOperation( "PermOnEnumerator",
    [ IsEnumerator, IsObject ] );

InstallMethod( PermOnEnumerator,
    [ IsEnumerator, IsObject ],
    function( enum, a )
    SetFilterObj( a, IsMultiplicativeElementWithInverse );
    a := Objectify( NewKind( PermutationsOnEnumeratorsFamily,
                 IsPermOnEnumeratorDefaultRep ),
                 rec( perm := a ) );
```

```
    SetEnumerator( a, enum );
    return a;
end );

InstallMethod( PermOnEnumerator,
    [ IsEnumerator, IsPermOnEnumeratorDefaultRep ],
    function( enum, a )
    a := Objectify( TypeObj( a ), rec( perm := a!.perm ) );
    SetEnumerator( a, enum );
    return a;
end );
```

Next we have to install new methods for the operations which calculate the product of two automorphisms, because this product must again have the right type. We also have to write a function which uses the enumerators to apply such an automorphism to positive integers.

Example

```
InstallMethod( \*, IsIdenticalObj,
    [ IsPermOnEnumeratorDefaultRep, IsPermOnEnumeratorDefaultRep ],
    function( a, b )
    perm := a!.perm * b!.perm;
    SetIsBijective( perm, true );
    return PermOnEnumerator( Enumerator( a ), perm );
end );

InstallMethod( \^,
    [ IsPosRat and IsInt, IsPermOnEnumeratorDefaultRep ],
    function( p, a )
    return PositionCanonical( Enumerator( a ),
                   Enumerator( a )[ p ] ^ a!.perm );
end );
```

How the corresponding methods for `p / aut` and `aut ^ n` look like is obvious.

Now we can formulate the recursive procedure `StabChainStrong` which extends the stabilizer chain by adding in new generators *newgens*. We content ourselves again with pseudo-code, emphasizing only the lines which set the `EnumeratorDomainPermutation`. We assume that initially S is a stabilizer chain for the trivial subgroup with a level for each pair $(range, g)$ characterizing an enumerator (as described above). We also assume that the `identity` element at each level already has the type corresponding to that level.

```
StabChainStrong := function( S, newgens )
  for i  in [ 1 .. Length( newgens ) ]  do
    newgens[ i ] := AutomorphismOnEnumerator( S.identity, newgens[ i ] );
  od;
  Extend the Schreier tree of S with newgens.
  for sch  in  Schreier generators  do
    if not sch in S.stabilizer  then
      StabChainStrong( S.stabilizer, [ sch ] );
    fi;
  od;
end;
```

References

[ACM98] ACM. *ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, New York, NY, USA, 1998. ACM Press. Chairman: Volker Weispfenning and Barry Trager. 1310, 1313

[AMW82] D. G. Arrell, S. Manrai, and M. F. Worboys. A procedure for obtaining simplified defining relations for a subgroup. In Campbell and Robertson [CR82], page 155–159. 706

[AR84] D. G. Arrell and E. F. Robertson. A modified Todd-Coxeter algorithm. In Atkinson [Atk84], page 27–32. 706

[Art73] E. Artin. *Galoissche Theorie*. Verlag Harri Deutsch, Zurich, 1973. Übersetzung nach der zweiten englischen Auflage besorgt von Viktor Ziegler, Mit einem Anhang von N. A. Milgram, Zweite, unveränderte Auflage, Deutsch-Taschenbücher, No. 21. 875

[Atk84] M. D. Atkinson, editor. *Computational group theory*, London, 1984. Academic Press Inc. [Harcourt Brace Jovanovich Publishers]. 1309, 1313, 1315, 1316

[Bak84] A. Baker. *A concise introduction to the theory of numbers*. Cambridge University Press, Cambridge, 1984. 194

[BC76] M. J. Beetham and C. M. Campbell. A note on the Todd-Coxeter coset enumeration algorithm. *Proc. Edinburgh Math. Soc. (2)*, 20(1):73–79, 1976. 690

[BC89] R. P. Brent and G. L. Cohen. A new lower bound for odd perfect numbers. *Math. Comp.*, 53(187):431–437, S7–S24, 1989. 196

[BC94] U. Baum and M. Clausen. Computing irreducible representations of supersolvable groups. *Math. Comp.*, 63(207):351–359, 1994. 1097, 1098

[BCFS91] L. Babai, G. Cooperman, L. Finkelstein, and Á. Seress. Nearly linear time algorithms for permutation groups with a small base. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC'91), Bonn 1991*, page 200–209. ACM Press, 1991. 600, 601

[BE99a] H. U. Besche and B. Eick. Construction of finite groups. *J. Symbolic Comput.*, 27(4):387–404, 1999. 656, 734

[BE99b] H. U. Besche and B. Eick. The groups of order at most 1000 except 512 and 768. *J. Symbolic Comput.*, 27(4):405–413, 1999. 734

[BE01] H. U. Besche and B. Eick. The groups of order $q^n \cdot p$. *Comm. Algebra*, 29(4):1759–1772, 2001. 734

[BEO01] H. U. Besche, B. Eick, and E. A. O'Brien. The groups of order at most 2000. *Electron. Res. Announc. Amer. Math. Soc.*, 7:1–4 (electronic), 2001. 734

[BEO02] H. U. Besche, B. Eick, and E. A. O'Brien. A millennium project: constructing small groups. *Internat. J. Algebra Comput.*, 12(5):623–644, 2002. 734

[Ber76] T. R. Berger. Characters and derived length in groups of odd order. *J. Algebra*, 39(1):199–207, 1976. 1189

[Bes92] H. U. Besche. Die Berechnung von Charaktergraden und Charakteren endlicher auflösbarer Gruppen im Computeralgebrasystem GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992. 1064

[BFS79] F. R. Beyl, U. Felgner, and P. Schmid. On groups occurring as center factor groups. *J. Algebra*, 61(1):161–177, 1979. 545

[BJR87] R. Brown, D. L. Johnson, and E. F. Robertson. Some computations of nonabelian tensor products of groups. *J. Algebra*, 111(1):177–202, 1987. 544, 545

[BL98] T. Breuer and S. Linton. The GAP 4 type system. organizing algebraic algorithms. In *ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation* [ACM98], page 38–45. Chairman: Volker Weispfenning and Barry Trager. 159

[BLS75] J. Brillhart, D. Lehmer, and J. Selfridge. New primality criteria and factorizations of $2^m \pm 1$. *Mathematics of Computation*, 29:620–647, 1975. 180

[BM83] G. Butler and J. McKay. The transitive groups of degree up to eleven. *Comm. Algebra*, 11(8):863–911, 1983. 732

[Bou70] N. Bourbaki. *Éléments de mathématique. Algèbre. Chapitres 1 à 3*. Hermann, Paris, 1970. 452

[BP98] T. Breuer and G. Pfeiffer. Finding possible permutation characters. *J. Symbolic Comput.*, 26(3):343–354, 1998. 1151, 1152, 1155

[Bre91] T. Breuer. Potenzabbildungen, Untergruppenfusionen, Tafel-Automorphismen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1991. 1064, 1158

[Bre97] T. Breuer. Integral bases for subfields of cyclotomic fields. *Appl. Algebra Engrg. Comm. Comput.*, 8(4):279–289, 1997. 891

[Bre99] T. Breuer. Computing possible class fusions from character tables. *Comm. Algebra*, 27(6):2733–2748, 1999. 1158

[BTW93] B. Beauzamy, V. Trevisan, and P. S. Wang. Polynomial factorization: sharp bounds, efficient algorithms. *J. Symbolic Comput.*, 15(4):393–413, 1993. 1002

[Bur55] W. Burnside. *Theory of groups of finite order*. Dover Publications Inc., New York, 1955. Unabridged republication of the second edition, published in 1911. 1039

[But93] G. Butler. The transitive groups of degree fourteen and fifteen. *J. Symbolic Comput.*, 16(5):413–422, 1993. 732

[Can73] J. J. Cannon. Construction of defining relators for finite groups. *Discrete Math.*, 5:105–129, 1973. 673, 685

[Car72] R. W. Carter. *Simple groups of Lie type*. John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28. 723

[CCN+85] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson. *Atlas of finite groups*. Oxford University Press, Eynsham, 1985. Maximal subgroups and ordinary characters for simple groups, With computational assistance from J. G. Thackray. 224, 227, 617, 742, 1079, 1088, 1093, 1094, 1107

[CHM98] J. H. Conway, A. Hulpke, and J. McKay. On transitive permutation groups. *LMS J. Comput. Math.*, 1:1–8 (electronic), 1998. 732

[CLO97] D. Cox, J. Little, and D. O'Shea. *Ideals, varieties, and algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition, 1997. An introduction to computational algebraic geometry and commutative algebra. 1008, 1011, 1012

[Coh93] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1993. 192, 341, 342

[Con90a] S. B. Conlon. Calculating characters of *p*-groups. *J. Symbolic Comput.*, 9(5-6):535–550, 1990. Computational group theory, Part 1. 1097

[Con90b] S. B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Comput.*, 9(5-6):551–570, 1990. Computational group theory, Part 1. 1074, 1097

[CR82] C. M. Campbell and E. F. Robertson, editors. *Groups–St. Andrews 1981*, volume 71 of *London Mathematical Society Lecture Note Series*, Cambridge, 1982. Cambridge University Press. 1309, 1315

[DE05] H. Dietrich and B. Eick. On the groups of cube-free order. *J. Algebra*, 292(1):122–137, 2005. 734

[Dix67] J. D. Dixon. High speed computation of group characters. *Numer. Math.*, 10:446–450, 1967. 1100

[Dix93] J. D. Dixon. Constructing representations of finite groups. In L. Finkelstein and W. M. Kantor, editors, *Groups and computation (New Brunswick, NJ, 1991)*, volume 11 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, page 105–112. Amer. Math. Soc., Providence, RI, 1993. 1098, 1099

[DM88] J. D. Dixon and B. Mortimer. The primitive permutation groups of degree less than 1000. *Math. Proc. Cambridge Philos. Soc.*, 103(2):213–238, 1988. 743, 744

[Dre69] A. Dress. A characterisation of solvable groups. *Math. Z.*, 110:213–217, 1969. 1049, 1053

[EH01] B. Eick and A. Hulpke. Computing the maximal subgroups of a permutation group I. In W. M. Kantor and Á. Seress, editors, *Proceedings of the 3rd International Conference held at The Ohio State University, Columbus, OH, June 15–19, 1999*, Ohio State University Mathematical Research Institute Publications, 8, page 155–168, Berlin, 2001. Walter de Gruyter & Co. 599

[EH03] B. Eick and B. Höfling. The solvable primitive permutation groups of degree at most 6560. *LMS J. Comput. Math.*, 6:29–39 (electronic), 2003. 743

[Eic97] B. Eick. Special presentations for finite soluble groups and computing (pre-)Frattini subgroups. In L. Finkelstein and W. M. Kantor, editors, *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, page 101–112. Amer. Math. Soc., Providence, RI, 1997. 639

[Ell98] G. Ellis. On the capability of groups. *Proc. Edinburgh Math. Soc. (2)*, 41(3):487–495, 1998. 545

[EO99a] B. Eick and E. A. O'Brien. Enumerating p-groups. *J. Austral. Math. Soc. Ser. A*, 67(2):191–205, 1999. Group theory. 734

[EO99b] B. Eick and E. A. O'Brien. The groups of order 512. In B. H. Matzat, G.-M. Greuel, and G. Hiss, editors, *Algorithmic algebra and number theory (Heidelberg, 1997)*, page 379–380. Springer, Berlin, 1999. Proceedings of Abschlusstagung des DFG Schwerpunktes Algorithmische Algebra und Zahlentheorie in Heidelberg. 734

[FJNT95] V. Felsch, D. L. Johnson, J. Neubüser, and S. V. Tsaranov. The structure of certain Coxeter groups. In *Groups '93 Galway/St Andrews, Vol. 1 (Galway, 1993)*, volume 211 of *London Math. Soc. Lecture Note Ser.*, page 177–190. Cambridge Univ. Press, Cambridge, 1995. 681

[FN79] V. Felsch and J. Neubüser. An algorithm for the computation of conjugacy classes and centralizers in p-groups. In E. W. Ng, editor, *Symbolic and algebraic computation (EUROSAM '79, Internat. Sympos., Marseille, 1979)*, volume 72 of *Lecture Notes in Comput. Sci.*, page 452–465. Springer, Berlin, 1979. EUROSAM '79, an International Symposium held in Marseille, June 1979. 643

[Fra82] J. S. Frame. Recursive computation of tensor power components. *Bayreuth. Math. Schr.*, 10:153–159, 1982. 1145

[Gir03] B. Girnat. Klassifikation der Gruppen bis zur Ordnung p^5. Staatsexamensarbeit, TU Braunschweig, Braunschweig, Germany, 2003. 734

[Hal36] P. Hall. The Eulerian functions of a group. *Quarterly J. Of Mathematics*, os-7(1):134–151, 1936. 524

[Hav69] G. Havas. Symbolic and algebraic calculation. Basser Computing Dept., Technical Report 89, Basser Department of Computer Science, University of Sydney, Sydney, Australia, 1969. 695

[Hav74] G. Havas. A Reidemeister-Schreier program. In M. F. Newman, editor, *Proceedings of the Second International Conference on the Theory of Groups (Australian Nat. Univ.,*

Canberra, 1973), volume 372 of *Lecture Notes in Math.*, pages 347–356. Lecture Notes in Math., Vol. 372, Berlin, 1974. Springer. Held at the Australian National University, Canberra, August 13–24, 1973, With an introduction by B. H. Neumann, Lecture Notes in Mathematics, Vol. 372. 688

[HB82] B. Huppert and N. Blackburn. *Finite groups. II*, volume 242 of *Grundlehren Math. Wiss.* Springer-Verlag, Berlin, 1982. 528

[HIÖ89] T. Hawkes, I. M. Isaacs, and M. Özaydin. On the Möbius function of a finite group. *Rocky Mountain J. Math.*, 19(4):1003–1034, 1989. 1050

[HJ59] M. Hall Jr. *The theory of groups*. The Macmillan Co., New York, N.Y., 1959. 600

[HJLP] G. Hiss, C. Jansen, K. Lux, and R. A. Parker. Computational Modular Character Theory. `http://www.math.rwth-aachen.de/~MOC/CoMoChaT/`. 1065

[HKRR84] G. Havas, P. E. Kenne, J. S. Richardson, and E. F. Robertson. A Tietze transformation program. In Atkinson [Atk84], page 69–73. 695

[How76] J. M. Howie. *An introduction to semigroup theory*. Academic Press [Harcourt Brace Jovanovich Publishers], London, 1976. L.M.S. Monographs, No. 7. 771

[HP89] D. F. Holt and W. Plesken. *Perfect groups*. Oxford Mathematical Monographs. The Clarendon Press Oxford University Press, New York, 1989. With an appendix by W. Hanrath, Oxford Science Publications. 737, 738, 740, 741

[HR94] D. F. Holt and S. Rees. Testing modules for irreducibility. *J. Austral. Math. Soc. Ser. A*, 57(1):1–16, 1994. 1035

[Hul93] A. Hulpke. Zur Berechnung von Charaktertafeln. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, 1993. 1064, 1100

[Hul96] A. Hulpke. *Konstruktion transitiver Permutationsgruppen*. Dissertation, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1996. 563, 732

[Hul98] A. Hulpke. Computing normal subgroups. In *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation (Rostock)* [ACM98], page 194–198 (electronic). Chairman: Volker Weispfenning and Barry Trager. 532

[Hul99] A. Hulpke. Computing subgroups invariant under a set of automorphisms. *J. Symbolic Comput.*, 27(4):415–427, 1999. 537

[Hul00] A. Hulpke. Conjugacy classes in finite permutation groups via homomorphic images. *Math. Comp.*, 69(232):1633–1651, 2000. 506

[Hul01] A. Hulpke. Representing subgroups of finitely presented groups by quotient subgroups. *Experiment. Math.*, 10(3):369–381, 2001. 676

[Hul05] A. Hulpke. Constructing transitive permutation groups. *J. Symbolic Comput.*, 39(1):1–30, 2005. 732

[Hum72] J. E. Humphreys. *Introduction to Lie algebras and representation theory*. Springer-Verlag, New York, 1972. Graduate Texts in Mathematics, Vol. 9. 975

[Hum78] J. E. Humphreys. *Introduction to Lie algebras and representation theory*, volume 9 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1978. Second printing, revised. 975

[Hup67] B. Huppert. *Endliche Gruppen. I.* Die Grundlehren der Mathematischen Wissenschaften, Band 134. Springer-Verlag, Berlin, 1967. 723

[IE94] H. Ishibashi and A. G. Earnest. Two-element generation of orthogonal groups over finite fields. *J. Algebra*, 165(1):164–171, 1994. 723

[Isa76] I. M. Isaacs. *Character theory of finite groups*. Academic Press [Harcourt Brace Jovanovich Publishers], New York, 1976. Pure and Applied Mathematics, No. 69. 1085, 1148, 1246

[JLPW95] C. Jansen, K. Lux, R. Parker, and R. Wilson. *An atlas of Brauer characters*, volume 11 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 1995. Appendix 2 by T. Breuer and S. Norton, Oxford Science Publications. 1156, 1157

[Joh97] D. L. Johnson. *Presentations of groups*, volume 15 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, second edition, 1997. 683

[Kau92] A. Kaup. Gitterbasen und Charaktere endlicher Gruppen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992. 1064

[KL90] P. Kleidman and M. Liebeck. *The subgroup structure of the finite classical groups*, volume 129 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1990. 723, 726

[Kli66] A. U. Klimyk. Decomposition of the direct product of irreducible representations of semisimple Lie algebras into irreducible representations. *Ukrain. Mat. Ž.*, 18(5):19–27, 1966. 974

[Kli68] A. U. Klimyk. Decomposition of a direct product of irreducible representations of a semisimple Lie algebra into irreducible representations. In *American Mathematical Society Translations. Series 2*, volume 76, page 63–73. American Mathematical Society, Providence, R.I., 1968. 974

[KLM01] G. Kemper, F. Lübeck, and K. Magaard. Matrix generators for the Ree groups ${}^2G_2(q)$. *Comm. Algebra*, 29(1):407–413, 2001. 723

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1998. 189

[Leo91] J. S. Leon. Permutation group algorithms based on partitions. I. Theory and algorithms. *J. Symbolic Comput.*, 12(4-5):533–583, 1991. Computational group theory, Part 2. 610

[LLJL82] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982. 341, 342

[LNS84] R. Laue, J. Neubüser, and U. Schoenwaelder. Algorithms for finite soluble groups and the SOGOS system. In Atkinson [Atk84], page 105–135. 574, 629, 631

[LP91] K. Lux and H. Pahlings. Computational aspects of representation theory of finite groups. In G. O. Michler and C. M. Ringel, editors, *Representation theory of finite groups and finite-dimensional algebras (Bielefeld, 1991)*, volume 95 of *Progr. Math.*, page 37–64, Basel, 1991. Birkhäuser. 1062

[LRW97] E. M. Luks, F. Rákóczi, and C. R. B. Wright. Some algorithms for nilpotent permutation groups. *J. Symbolic Comput.*, 23(4):335–354, 1997. 1240

[Lüb03] F. Lübeck. Conway polynomials for finite fields. `http://www.math.rwth-aachen.de:8001/~Frank.Luebeck/data/ConwayPol`, 2003. 883

[Maa10] L. Maas. On a construction of the basic spin representations of symmetric groups. *Communications in Algebra*, 38:4545–4552, 2010. 545

[Mac81] I. G. Macdonald. Numbers of conjugacy classes in some finite classical groups. *Bull. Austral. Math. Soc.*, 23(1):23–48, 1981. 730

[MN89] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral. Math. Soc.*, 40(2):281–292, 1989. 641, 643

[Mur58] F. D. Murnaghan. The orthogonal and symplectic groups. *Comm. Dublin Inst. Adv. Studies. Ser. A, no.*, 13:146, 1958. 1144, 1145

[MV97] M. Mahajan and V. Vinay. Determinant: combinatorics, algorithms, and complexity. *Chicago J. Theoret. Comput. Sci.*, pages Article 5, 26 pp. (electronic), 1997. 314

[MY79] J. McKay and K. C. Young. The nonabelian simple groups G, $|G| < 10^6$–minimal generating pairs. *Math. Comp.*, 33(146):812–814, 1979. 540

[Neb95] G. Nebe. *Endliche rationale Matrixgruppen vom Grad 24*. Dissertation, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1995. 747

[Neb96] G. Nebe. Finite subgroups of $GL_n(Q)$ for $25 \leq n \leq 31$. *Comm. Algebra*, 24(7):2341–2397, 1996. 747

[Neu82] J. Neubüser. An elementary introduction to coset table methods in computational group theory. In Campbell and Robertson [CR82], page 1–45. 664, 673, 685, 690

[Neu92] J. Neukirch. *Algebraische Zahlentheorie*. Springer, Berlin, Heidelberg and New York, 1992. 1024

[New77] M. F. Newman. Determination of groups of prime-power order. In R. A. Bryce, J. Cossey, and M. F. Newman, editors, *Group theory (Proc. Miniconf., Australian Nat. Univ., Canberra, 1975)*, volume 573 of *Lecture Notes in Math.*, pages 73–84. Lecture Notes in Math., Vol. 573, Berlin, 1977. Springer. Lecture Notes in Mathematics, Vol. 573. 734

[New90] M. F. Newman. Proving a group infinite. *Arch. Math. (Basel)*, 54(3):209–211, 1990. 683

[NOVL04] M. F. Newman, E. A. O'Brien, and M. R. Vaughan-Lee. Groups and nilpotent Lie rings whose order is the sixth power of a prime. *J. Algebra*, 278(1):383–401, 2004. 734

[NP95a] G. Nebe and W. Plesken. *Finite rational matrix groups*. Number 556 in Mem. Amer. Math. Soc. AMS, 1995. vol. 116. 1316

[NP95b] G. Nebe and W. Plesken. *Finite rational matrix groups of degree 16*, page 74–144. In *Mem. Amer. Math. Soc.* [NP95a], 1995. vol. 116. 747

[NPP84] J. Neubüser, H. Pahlings, and W. Plesken. CAS; design and use of a system for the handling of characters of finite groups. In Atkinson [Atk84], page 195–247. 1062, 1064, 1091, 1146, 1153

[O'B90] E. A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9(5-6):677–698, 1990. Computational group theory, Part 1. 734

[O'B91] E. A. O'Brien. The groups of order 256. *J. Algebra*, 143(1):219–235, 1991. 734

[OVL05] E. A. O'Brien and M. R. Vaughan-Lee. The groups with order p^7 for odd prime p. *J. Algebra*, 292(1):243–258, 2005. 734

[Pah93] H. Pahlings. On the Möbius function of a finite group. *Arch. Math. (Basel)*, 60(1):7–14, 1993. 1050

[Par84] R. A. Parker. The computer calculation of modular characters (the meat-axe). In Atkinson [Atk84], page 267–274. 1027

[Pfe91] G. Pfeiffer. Von Permutationscharakteren und Markentafeln. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1991. 1064

[Pfe97] G. Pfeiffer. The subgroups of M_{24}, or how to compute the table of marks of a finite group. *Experiment. Math.*, 6(3):247–270, 1997. 1040, 1049

[Ple85] W. Plesken. Finite unimodular groups of prime degree and circulants. *J. Algebra*, 97(1):286–312, 1985. 747

[Ple95] W. Plesken. Solving $XX^{\mathrm{tr}} = A$ over the integers. *Linear Algebra Appl.*, 226/228:331–344, 1995. 343

[PN95] W. Plesken and G. Nebe. *Finite rational matrix groups*, page 1–73. In *Mem. Amer. Math. Soc.* [NP95a], 1995. vol. 116. 747, 748

[Poh87] M. Pohst. A modification of the LLL reduction algorithm. *J. Symbolic Comput.*, 4(1):123–127, 1987. 341, 342

[PP77] W. Plesken and M. Pohst. On maximal finite irreducible subgroups of GL(n,Z). I. the five and seven dimensional cases, II. the six dimensional case. *Math. Comp.*, 31:536–576, 1977. 747

[PP80] W. Plesken and M. Pohst. On maximal finite irreducible subgroups of GL(n,Z). III. the nine dimensional case, IV. remarks on even dimensions with application to n = 8, V. the eight dimensional case and a complete description of dimensions less than ten. *Math. Comp.*, 34:245–301, 1980. 747

[RD05] C. M. Roney-Dougal. The primitive permutation groups of degree less than 2500. *J. Algebra*, 292(1):154–183, 2005. 743

[RDU03] C. M. Roney-Dougal and W. R. Unger. The affine primitive permutation groups of degree less than 1000. *J. Symbolic Comput.*, 35(4):421–439, 2003. 743, 745

[Rin93] M. Ringe. *The C MeatAxe, Release 1.5*. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1993. 301

[Rob88] E. F. Robertson. Tietze transformations with weighted substring search. *J. Symbolic Comput.*, 6(1):59–64, 1988. 695

[Roy87] G. F. Royle. The transitive groups of degree twelve. *J. Symbolic Comput.*, 4(2):255–268, 1987. 732

[RT98] L. J. Rylands and D. E. Taylor. Matrix generators for the orthogonal groups. *J. Symbolic Comput.*, 25(3):351–360, 1998. 723

[Sch11] J. Schur. Über die darstellung der symmetrischen und der alternierenden gruppe durch gebrochene lineare substitutionen. *Journal für die reine und angewandte Mathematik*, 139:155–250, 1911. 545

[Sch90] G. J. A. Schneider. Dixon's character table algorithm revisited. *J. Symbolic Comput.*, 9(5-6):601–606, 1990. Computational group theory, Part 1. 1100

[Sch92] M. Scherner. Erweiterung einer Arithmetik von Kreisteilungskörpern auf deren Teilkörper und deren Implementation in GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992. 1064

[Sch94] U. Schiffer. Cliffordmatrizen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1994. 1064

[Sco73] L. L. Scott. Modular permutation representations. *Trans. Amer. Math. Soc.*, 175:101–121, 1973. 1153

[Sho92] M. W. Short. *The primitive soluble permutation groups of degree less than* 256, volume 1519 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1992. 743

[Sim70] C. C. Sims. Computational methods in the study of permutation groups. In J. Leech, editor, *Computational Problems in Abstract Algebra (Proc. Conf., Oxford, 1967)*, volume 29 of *Proceedings of a Conference held at Oxford under the auspices of the Science Research Council, Atlas Computer Laboratory*, page 169–183, Oxford, 1970. Pergamon. 599, 600

[Sim90] C. C. Sims. Computing the order of a solvable permutation group. *J. Symbolic Comput.*, 9(5-6):699–705, 1990. Computational group theory, Part 1. 611

[Sim94] C. C. Sims. *Computation with finitely presented groups*, volume 48 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1994. 448, 648, 668, 669, 786

[Sim97] C. C. Sims. Computing with subgroups of automorphism groups of finite groups. In W. Küchlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*, page 400–403 (electronic), New York, 1997. The Association for Computing Machinery, ACM. Held in Kihei, HI, July 21–23, 1997. 560, 1305

[SM85] L. Soicher and J. McKay. Computing Galois groups over the rationals. *J. Number Theory*, 20(3):273–281, 1985. 1002

[Sou94] B. Souvignier. Irreducible finite integral matrix groups of degree 8 and 10. *Math. Comp.*, 63(207):335–350, 1994. With microfiche supplement. 747

[SPA89] Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany. *SPAS - Subgroup Presentation Algorithms System, version 2.5, User's reference manual*, 1989. 688

[Tay87] D. E. Taylor. Pairs of generators for matrix groups. I. *The Cayley Bulletin*, 3, 1987. 723

[The93] H. Theißen. Methoden zur Bestimmung der rationalen Konjugiertheit in endlichen Gruppen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1993. 643

[The97] H. Theißen. *Eine Methode zur Normalisatorberechnung in Permutationsgruppen mit Anwendungen in der Konstruktion primitiver Gruppen*. Dissertation, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997. 610, 743

[vdW76] R. W. van der Waall. On symplectic primitive modules and monomial groups. *Nederl. Akad. Wetensch. Proc. Ser. A 79, Indag. Math.*, 38(4):362–375, 1976. 1196

[Wag90] S. Wagon. Editor's corner: the Euclidean algorithm strikes again. *Amer. Math. Monthly*, 97(2):125–129, 1990. 198

[Wie69] H. Wielandt. Permutation groups through invariant relations and invariant functions. Lecture notes, Department of Mathematics, The Ohio State University, 1969. 1243

[Zag90] D. Zagier. A one-sentence proof that every prime $p \equiv 1 \pmod 4$ is a sum of two squares. *Amer. Math. Monthly*, 97(2):144, 1990. 198

[Zum89] M. Zumbroich. Grundlagen einer Arithmetik in Kreisteilungskörpern und ihre Implementation in CAS. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1989. 890, 1064

Index

*, 408
    for pcwords, 647
*, 57
    for character tables, 1073
\+, 408
+, 57
-, 57
-A, 33
-B, 34
-C, 38
-D, 34
-E, 34
-K, 35
-L, 35
-M, 36
-O, 36
-P, 38
-R, 37
-T, 37
-U, 38
-W, 38
-X, 37
-a, 33
-b, 34
-e, 34
-f, 35
-g, 35
-g -g, 35
-h, 35
-i, 35
-infinity, 223
-l, 35
-m, 36
-n, 36
-o, 36
-p, 38
-q, 36
-r, 37
-s, 37
-x, 37
-y, 37
-z, 38
\., 378
\.\:\=, 378
\/, 408
    for a free group and a list of elements, 660
    for a free monoid and a list of pairs of elements, 785
    for a free semigroup and a list of pairs of elements, 782
/, 57
    for character tables, 1073
\<, 407
    for associative words, 471
    for nonassociative words, 464
    for pcwords, 647
    for permutations, 590
    for two elements in a f.p. group, 661
    for two strings, 353
\=, 407
    for associative words, 470
    for nonassociative words, 464
    for pcwords, 647
    for permutations, 590
    for two elements in a f.p. group, 661
    for two elements in a f.p. semigroup, 783
    for two strings, 353
\[\], 246
\[\]\:\=, 246
*
    for transformations, 796
\/
    for a transformation and a permutation, 796
\<
    for transformations, 797
\=
    for transformations, 797
\^

for a positive integer and a transformation, 796
for a transformation and a permutation, 796
\', 350
\XYZ, 350
\\, 350
\b, 350
\c, 350
\r, 350
^
for class functions, 1122
\^, 408
for a field and a pair of integers, 905
for a field and an integer, 905
^, 57
\{\}, 248
\{\}\:\=, 250

abelian number field, 888
abelian number fields
CanonicalBasis, 889
Galois group, 891
AbelianGroup, 720
AbelianInvariants, 523
for a character table, 1076
AbelianInvariants
for groups, 523
AbelianInvariantsMultiplier, 544
AbelianInvariantsNormalClosureFpGroup, 681
AbelianInvariantsNormalClosureFpGroupRrs, 681
AbelianInvariantsOfList, 339
AbelianInvariantsSubgroupFpGroup, 680
AbelianInvariantsSubgroupFpGroupMtc, 681
AbelianInvariantsSubgroupFpGroupRrs
for a group and a coset table, 681
for two groups, 681
AbelianNumberField, 887
AbelianSubfactorAction, 578
About GAP manual, 23
AbsInt, 174
absolute value of an integer, 174
AbsoluteDiameter, 236
AbsoluteIrreducibleModules, 1099
AbsolutelyIrreducibleModules, 1099
AbsoluteValue, 221
AbsolutIrreducibleModules, 1099
abstract word, 462
AbstractWordTietzeWord, 691
accessing
list elements, 247
record elements, 373
AClosestVectorCombinationsMatFFEVecFFE, 306
AClosestVectorCombinationsMatFFEVecFFECoords, 306
Acos, 235
Acosh, 235
ActingAlgebra, 943
ActingDomain, 584
Action
for a group, an action domain, etc., 576
for an external set, 576
action
by conjugation, 567
on blocks, 568
on sets, 568
ActionHomomorphism
for a group, an action domain, etc., 575
for an action image, 575
for an external set, 575
actions, 567
ActorOfExternalSet, 587
Add, 250
add
an element to a set, 276
AddCoeffs, 303
AddDictionary, 369
AddGenerator, 694
AddGeneratorsExtendSchreierTree, 610
addition, 57
list and non-list, 261
matrices, 310
matrix and scalar, 311
operation, 408
rational functions, 990
scalar and matrix, 311
scalar and matrix list, 312
vector and scalar, 300
vectors, 300
AdditiveInverse, 407
AdditiveInverseAttr, 406

AdditiveInverseImmutable, 406
AdditiveInverseMutable, 407
AdditiveInverseOp, 407
AdditiveInverseSameMutability, 407
AdditiveInverseSM, 407
AdditiveNeutralElement, 842
AddRelator, 695
AddRowVector, 303
AddRule, 487
AddRuleReduced, 487
AddSet, 276
AdjointAssociativeAlgebra, 967
AdjointBasis, 928
AdjointMatrix, 967
AdjointModule, 945
AffineAction, 642
AffineActionLayer, 642
Agemo, 516
Algebra, 915
AlgebraByStructureConstants, 917
AlgebraGeneralMappingByImages, 934
AlgebraHomomorphismByImages, 935
AlgebraHomomorphismByImagesNC, 935
AlgebraicExtension, 1020
AlgebraWithOne, 916
AlgebraWithOneGeneralMappingByImages, 936
AlgebraWithOneHomomorphismByImages, 936
AlgebraWithOneHomomorphismByImagesNC, 936
AllAutomorphisms, 561
AllBlocks, 583
AllEndomorphisms, 561
AllHomomorphismClasses, 561
AllHomomorphisms, 561
AllIrreducibleSolvableGroups, 746
All*Library*Groups, 731
AllPrimitiveGroups, 731
AllSmallGroups, 735
AllSmallNonabelianSimpleGroups, 521
AllSubgroups, 531
AllTransitiveGroups, 731
Alpha, 1189
AlternatingGroup
  for a degree, 722
  for a domain, 722
and, 242
  for filters, 160, 242
ANFAutomorphism, 893
AntiIsomorphismTransformationSemigroup, 811
antisymmetric relation, 438
AntiSymmetricParts, 1144
Append, 251
AppendTo, 130
  for streams, 139
ApplicableMethod, 101
ApplicableMethodTypes, 101
Apply, 280
ApplySimpleReflection, 963
ApproximateSuborbitsStabilizerPermGroup, 608
ARCH_IS_MAC_OS_X, 42
ARCH_IS_UNIX, 42
ARCH_IS_WINDOWS, 42
arg
  special function argument, 55, 66
Argument, 236
arithmetic operators
  precedence, 58
ArithmeticElementCreator, 1248
Arrangements, 204
arrow notation for functions, 67
AsAlgebra, 928
AsAlgebraWithOne, 928
AsBinaryRelationOnPoints
  for a binary relation, 440
  for a permutation, 440
  for a transformation, 440
AsBlockMatrix, 333
AscendingChain, 528
AsDivisionRing, 870
AsDuplicateFreeList, 279
AsField, 870
AsGroup, 493
AsGroupGeneralMappingByImages, 551
Asin, 235
Asinh, 235
AsInternalFFE, 881
AsLeftIdeal, 850
AsLeftModule, 863
AsList, 384
AsMagma, 455
AsMonoid, 761

AsPartialPerm
for a permutation, 826
for a permutation and a positive integer, 826
for a permutation and a set of positive integers, 826
for a transformation, 827
for a transformation and a positive integer, 827
for a transformation and a set of positive integer, 827
AsPermutation, 594
AsPolynomial, 993
AsRightIdeal, 850
AsSemigroup, 758
Assert, 105
AssertionLevel, 105
AsSet, 385
AssignGeneratorVariables, 470
assignment
to a list, 248
to a record, 374
variable, 59
AssignNiceMonomorphismAutomorphismGroup, 560
AssociatedPartition, 211
AssociatedReesMatrixSemigroupOfDClass, 778
Associates, 854
associativity, 57
AssocWordByLetterRep, 476
AsSortedList, 385
AsSSortedList, 385
As*Struct*, 399
AsSubalgebra, 929
AsSubalgebraWithOne, 929
AsSubgroup, 495
AsSubgroupOfWholeGroupByQuotient, 677
AsSubmagma, 455
AsSubmonoid, 761
AsSubsemigroup, 758
AsSubspace, 895
AsSub*struct*, 402
AsTransformation, 794
AsTwoSidedIdeal, 850
AsVectorSpace, 895
at exit functions, 90
Atan, 235
Atan2, 235
Atanh, 235
AtlasIrrationality, 227
atomic irrationalities, 224
AttributeValueNotSet, 167
AugmentationIdeal, 982
AugmentedCosetTableInWholeGroup, 670
AugmentedCosetTableMtc, 670
AugmentedCosetTableRrs, 670
automatic loading of GAP packages, 1198
automorphism group
of number fields, 891
AutomorphismDomain, 558
AutomorphismGroup, 558
AutomorphismGroup
for groups with pcgs, 643
AutomorphismsOfTable, 1078

b_N (irrational value), 224
backslash character, 350
backspace character, 350
Backtrace
GAP3 name for Where, 87
BANNER, 1209
BaseFixedSpace, 320
BaseIntersectionIntMats, 335
BaseIntMat, 335
BaseMat, 325
BaseMatDestructive, 325
BaseOfGroup, 606
BaseOrthogonalSpaceMat, 325
BaseStabChain, 606
BaseSteinitzVectors, 326
BasicSpinRepresentationOfSymmetricGroup, 546
BasicWreathProductOrdering, 450
Basis, 898
BasisNC, 898
BasisVectors, 899
Bell, 201
Bernoulli, 201
BestQuoInt, 176
BestSplittingMatrix, 1101
BiAlgebraModule, 941
BiAlgebraModuleByGenerators, 940
BibEntry, 1206
BilinearFormMat, 961

binary relation, 437
BinaryRelationByElements, 437
BinaryRelationOnPoints, 440
BinaryRelationOnPointsNC, 440
BindGlobal, 53
Binomial, 200
BisectInterval, 236
blank, 47
BlistList, 294
BlockMatrix, 333
Blocks
for a group, an action domain, etc., 582
for an external set, 582
BlocksInfo, 1086
BlownUpMat, 328
BlownUpVector, 328
BlowupInterval, 236
BlowUpIsomorphism, 615
body, 65
BombieriNorm, 1002
bound, 50
Brauer character, 1128
BrauerCharacterValue, 1156
BrauerTable
for a character table, and a prime integer, 1066
for a group, and a prime integer, 1066
BrauerTableOp, 1066
BravaisGroup, 620
BravaisSubgroups, 620
BravaisSupergroups, 621
Break loop message, 86
break statement, 64
browsing backwards, 28
browsing backwards one chapter, 29
browsing forward, 28
browsing forward one chapter, 29
browsing the next section browsed, 29
browsing the previous section browsed, 29

c_N (irrational value), 224
CallFuncList, 72
CanComputeIndex, 548
CanComputeIsSubset, 548
CanComputeSize, 547
CanComputeSizeAnySubgroup, 548
candidates
for permutation characters, 1148
CanEasilyCompareElements, 408
CanEasilyCompareElementsFamily, 408
CanEasilyComputePcgs, 623
CanEasilyComputeWithIndependentGensAbelianGroup, 547
CanEasilySortElements, 408
CanEasilySortElementsFamily, 408
CanEasilyTestMembership, 547
canonical basis
for matrix spaces, 907
for row spaces, 907
CanonicalBasis, 899
CanonicalGenerators, 961
CanonicalPcElement, 627
CanonicalPcgs, 631
CanonicalPcgsByGeneratorsWithImages, 633
CanonicalRepresentativeDeterminatorOfExternalSet, 587
CanonicalRepresentativeOfExternalSet, 586
CanonicalRightCosetElement, 503
Carmichael's lambda function, 191
carriage return character, 350
CartanMatrix, 961
CartanSubalgebra, 957
Cartesian
for a list, 282
for various objects, 282
CategoriesOfObject, 163
CategoryCollections, 380
CategoryFamily, 1217
Ceil, 236
Center, 459
center, 459
central character, 1132
CentralCharacter, 1132
CentralIdempotentsOfAlgebra, 932
centraliser, 459
Centralizer
for a class of objects in a magma, 459
for a magma and a submagma, 459
for a magma and an element, 459
Centralizer
for groups with pcgs, 643
CentralizerInGLnZ, 620

CentralizerModulo, 530
CentralizerSizeLimitConsiderFunction, 644
CentralNormalSeriesByPcgs, 636
Centre, 459
Centre
for groups with pcgs, 643
centre
of a character, 1131
CentreOfCharacter, 1131
CF
for (subfield and) conductor, 886
for (subfield and) generators, 886
ChangeStabChain, 609
Character
for a character table and a list, 1125
for a group and a list, 1125
character tables, 1065
access to, 1065
calculate, 1065
infix operators, 1073
of groups, 1065
character value
of group element using powering operator, 1122
CharacterDegrees
for a character table, 1073
for a group, 1073
Characteristic, 403
for a class function, 1123
characteristic polynomial
for field elements, 873
CharacteristicPolynomial, 327
CharacterNames, 1079
CharacterParameters, 1079
characters, 1116
permutation, 1148
symmetrizations of, 1143
CharacterTable
for a group, 1065
for a string, 1065
for an ordinary character table, 1065
CharacterTableDirectProduct, 1105
CharacterTableFactorGroup, 1106
CharacterTableIsoclinic, 1107
CharacterTableRegular, 1066
CharacterTableWithSortedCharacters, 1108
CharacterTableWithSortedClasses, 1109
CharacterTableWithStoredGroup, 1071
CharacterTableWreathSymmetric, 1107
CharInt, 359
CharsFamily, 352
CharSInt, 360
CheckDigitISBN, 186
CheckDigitISBN13, 186
CheckDigitPostalMoneyOrder, 186
CheckDigitTestFunction, 187
CheckDigitUPC, 186
CheckFixedPoints, 1174
CheckForHandlingByNiceBasis, 914
CheckPermChar, 1182
ChevalleyBasis, 959
ChiefNormalSeriesByPcgs, 637
ChiefSeries, 524
ChiefSeriesThrough, 525
ChiefSeriesUnderAction, 525
Chinese remainder, 178
ChineseRem, 178
Chomp, 358
Cite, 1207
CIUnivPols, 990
class function, 1118
class function objects, 1118
class functions, 1169
as ring elements, 1121
class multiplication coefficient, 1091, 1092
ClassElementLattice, 533
classes
real, 1081
ClassesSolvableGroup, 643
ClassFunction
for a character table and a list, 1125
for a group and a list, 1125
ClassFunctionSameType, 1126
ClassMultiplicationCoefficient
for character tables, 1091
ClassMultiplicationCoefficient
for character tables, 1091
ClassNames, 1079
ClassNamesTom, 1048
ClassOrbit, 1081
ClassParameters, 1079

ClassPermutation, 1110
ClassPositionsOfAgemo, 1082
ClassPositionsOfCenter
for a character table, 1082
ClassPositionsOfCentre
for a character, 1131
for a character table, 1082
ClassPositionsOfDerivedSubgroup, 1083
ClassPositionsOfDirectProduct-
Decompositions, 1082
ClassPositionsOfElementaryAbelian-
Series, 1083
ClassPositionsOfFittingSubgroup, 1083
ClassPositionsOfKernel, 1130
ClassPositionsOfLowerCentralSeries,
1083
ClassPositionsOfMaximalNormal-
Subgroups, 1082
ClassPositionsOfMinimalNormal-
Subgroups, 1082
ClassPositionsOfNormalClosure, 1084
ClassPositionsOfNormalSubgroup, 1114
ClassPositionsOfNormalSubgroups, 1082
ClassPositionsOfPCore, 1084
ClassPositionsOfSupersolvableResiduum,
1084
ClassPositionsOfUpperCentralSeries,
1084
ClassRoots, 1081
ClassStructureCharTable, 1091
ClassTypesTom, 1047
CleanedTailPcElement, 627
ClearCacheStats, 112
ClearProfile, 109
clone
an object, 156
CloseMutableBasis, 904
CloseStream, 135
ClosureGroup, 497
ClosureGroupAddElm, 497
ClosureGroupCompare, 497
ClosureGroupDefault, 497
ClosureGroupIntest, 497
ClosureLeftModule, 864
ClosureNearAdditiveGroup
for a near-additive group and an element, 843
for two near-additive groups, 843
ClosureRing
for a ring and a ring element, 846
for two rings, 846
Closure*Struct*, 398
ClosureSubgroup, 498
ClosureSubgroupNC, 498
Coboundaries, 973
Cochain, 972
CochainSpace, 972
Cocycles
for Lie algebra module, 973
cocycles, 541
CodegreeOfPartialPerm, 819
CodegreeOfPartialPermCollection, 819
CodegreeOfPartialPermSemigroup, 834
CodePcGroup, 657
CodePcgs, 656
coefficient
binomial, 200
Coefficients, 900
coefficients
for cyclotomics, 221
CoefficientsAndMagmaElements, 983
CoefficientsFamily, 1014
CoefficientsMultiadic, 178
CoefficientsOfLaurentPolynomial, 1003
CoefficientsOfUnivariatePolynomial, 994
CoefficientsOfUnivariateRational-
Function, 993
CoefficientsQadic, 178
CoefficientsRing, 1005
CoeffsCyc, 221
CoeffsMod, 304
cohomology, 541
COHORTS_PRIMITIVE_GROUPS, 744
CoKernelOfAdditiveGeneralMapping, 431
CoKernelOfMultiplicativeGeneral-
Mapping, 430
CollapsedMat, 1177
Collected, 278
CollectionsFamily, 379
ColorPrompt, 43
Columns, 777
Combinations, 203
CombinatorialCollector, 649
Comm, 409
Comm

for words, 471
comments, 47
CommutativeDiagram, 1173
CommutatorFactorGroup, 529
CommutatorLength, 512
for a character table, 1076
CommutatorSubgroup, 511
Compacted, 278
CompanionMat, 329
CompareVersionNumbers, 1203
comparison
fp semigroup elements, 783
operation, 407
rational functions, 991
comparisons
of booleans, 241
of lists, 257
CompatibleConjugacyClasses, 1072
CompatiblePairs, 655
ComplementClassesRepresentatives, 510
ComplementClassesRepresentativesEA, 543
ComplementIntMat, 335
ComplementSystem, 515
ComplexConjugate, 228
for a class function, 1123
ComplexI, 236
ComplexificationQuat
for a matrix, 921
for a vector, 921
ComponentPartialPermInt, 829
ComponentRepsOfPartialPerm, 825
ComponentRepsOfTransformation, 805
ComponentsOfPartialPerm, 824
ComponentsOfTransformation, 804
ComponentTransformationInt, 797
CompositionMapping, 421
CompositionMapping
for Frobenius automorphisms, 882
CompositionMapping2, 421
CompositionMapping2General, 421
CompositionMaps, 1170
CompositionOfStraightLinePrograms, 480
CompositionSeries, 525
CompositionSeries
for groups with pcgs, 643
ComputedBrauerTables, 1066
ComputedClassFusions, 1165
ComputedIndicators, 1090
ComputedIsPSolubleCharacterTables, 1089
ComputedIsPSolvableCharacterTables, 1089
ComputedPowerMaps, 1159
ComputedPrimeBlockss, 1085
Concatenation
for a list of lists, 277
for several lists, 277
concatenation
of lists, 277
Conductor
for a collection of cyclotomics, 220
for a cyclotomic, 220
ConfluentRws, 486
Congruences
for character tables, 1179
ConjugacyClass, 506
ConjugacyClasses
attribute, 506
for character tables, 1070
ConjugacyClasses
for groups with pcgs, 643
for linear groups, 729
ConjugacyClassesByOrbits, 507
ConjugacyClassesByRandomSearch, 507
ConjugacyClassesMaximalSubgroups, 531
ConjugacyClassesPerfectSubgroups, 535
ConjugacyClassesSubgroups, 531
ConjugacyClassSubgroups, 530
conjugate
matrix, 312
of a word, 471
ConjugateDominantWeight, 964
ConjugateDominantWeightWithWord, 964
ConjugateGroup, 493
Conjugates, 874
ConjugateSubgroup, 496
ConjugateSubgroups, 496
conjugation, 567
ConjugatorAutomorphism, 556
ConjugatorAutomorphismNC, 556
ConjugatorIsomorphism, 556
ConjugatorOfConjugatorIsomorphism, 557
ConsiderKernels, 1180
ConsiderSmallerPowerMaps, 1180
ConsiderStructureConstants, 1168

ConsiderTableAutomorphisms, 1183
ConstantTimeAccessList, 272
ConstantTransformation, 794
constituent
of a group character, 1129
ConstituentsCompositionMapping, 421
ConstituentsOfCharacter, 1130
ContainedCharacters, 1178
ContainedDecomposables, 1178
ContainedMaps, 1172
ContainedPossibleCharacters, 1176
ContainedPossibleVirtualCharacters, 1176
ContainedSpecialVectors, 1176
ContainedTom, 1053
ContainingTom, 1053
continue statement, 65
ContinuedFractionApproximationOfRoot, 197
ContinuedFractionExpansionOfRoot, 197
convert
to a string, 351
ConvertToBlistRep, 297
ConvertToCharacterTable, 1067
ConvertToCharacterTableNC, 1067
ConvertToMatrixRep
for a list (and a field), 330
for a list (and a prime power), 330
ConvertToMatrixRepNC
for a list (and a field), 330
for a list (and a prime power), 331
ConvertToRangeRep, 290
ConvertToStringRep, 351
ConvertToTableOfMarks, 1046
ConvertToVectorRep
for a list (and a field), 302
for a list (and a prime power), 302
ConvertToVectorRepNC
for a list (and a field), 302
for a list (and a prime power), 302
ConwayPolynomial, 883
coprime, 58
Copy, 156
copy
an object, 156
CopyListEntries, 251
CopyOptionsDefaults, 608
CopyStabChain, 608
CopyToStringRep, 351
Core, 510
CorrespondingGeneratorsByModuloPcgs, 633
Cos, 235
coset, 502
CosetDecomposition, 503
CosetLeadersMatFFE, 306
CosetTable, 665
CosetTableBySubgroup, 666
CosetTableDefaultLimit, 668
CosetTableDefaultMaxLimit, 667
CosetTableFromGensAndRels, 666
CosetTableInWholeGroup, 669
CosetTableOfFpSemigroup, 788
CosetTableStandard, 669
Cosh, 235
Cot, 235
Coth, 235
CrcFile, 131
CrcString, 362
CrystGroupDefaultAction, 621
Csc, 235
Csch, 235
Cuberoot, 236
Cycle, 579
CycleIndex
for a permutation and an action domain, 580
for a permutation group and an action domain, 580
CycleLength, 579
CycleLengths, 579
Cycles, 579
CyclesOfTransformation, 805
CycleStructureClass, 1132
CycleStructurePerm, 592
CycleTransformationInt, 806
CyclicExtensionsTom
for a list of primes, 1053
for a prime, 1053
CyclicGroup, 720
cyclotomic field elements, 218
cyclotomic fields
CanonicalBasis, 889
CyclotomicField
for (subfield and) conductor, 886

for (subfield and) generators, 886
CyclotomicPolynomial, 1000
Cyclotomics, 218
CyclotomicsFamily, 219

d_N (irrational value), 224
Darstellungsgruppe
see EpimorphismSchurCover, 544
data type
unknown, 1185
DataType, 171
DayDMY, 363
DaysInMonth, 362
DaysInYear, 362
DEC, 339
DeclareAttribute
DeclareAttribute
example, 1247
DeclareAttribute, 1235
DeclareAttribute
example, 1242
DeclareAutoPackage, 1209
DeclareAutoreadableVariables, 1203
DeclareCategory, 1234
DeclareFilter, 1235
DeclareGlobalFunction, 1235
DeclareGlobalVariable, 1236
DeclareHandlingByNiceBasis, 913
DeclareInfoClass, 103
DeclareOperation, 1235
DeclarePackage, 1209
DeclarePackageAutoDocumentation, 1209
DeclarePackageDocumentation, 1209
DeclareProperty, 1235
DeclareRepresentation, 1234
DeclareRepresentation
belongs to implementation part, 1237
example, 1243
DeclareSynonym, 1236
DeclareSynonymAttr, 1236
DeclareUserPreference, 40
DecodeTree, 706
decompose
a group character, 1129
DecomposedFixedPointVector, 1054
DecomposeTensorProduct, 974
Decomposition, 340
decomposition matrix, 339
DecompositionInt, 341
DecompositionMatrix, 1087
Decreased, 1141
DEFAULTDISPLAYSTRING, 354
DefaultField
for a list of generators, 869
for cyclotomics, 223
for finite field elements, 881
for several generators, 869
DefaultFieldByGenerators, 869
DefaultFieldOfMatrix, 313
DefaultFieldOfMatrixGroup, 614
DefaultInfoHandler, 104
DefaultRing
for a collection, 845
for finite field elements, 881
for ring elements, 845
DefaultRingByGenerators, 845
DefaultStabChainOptions, 604
DEFAULTVIEWSTRING, 354
DefiningPolynomial, 871
DefiningQuotientHomomorphism, 677
DegreeFFE
for a FFE, 879
for a matrix of FFEs, 879
for a vector of FFEs, 879
DegreeIndeterminate, 997
DegreeOfBinaryRelation, 439
DegreeOfCharacter, 1128
DegreeOfLaurentPolynomial, 995
DegreeOfPartialPerm, 818
DegreeOfPartialPermCollection, 818
DegreeOfPartialPermSemigroup, 834
DegreeOfTransformation, 798
DegreeOfTransformationCollection, 798
DegreeOfTransformationSemigroup, 809
DegreeOverPrimeField, 871
Delta, 1189
denominator
of a rational, 217
DenominatorCyc, 222
DenominatorOfModuloPcgs, 632
DenominatorOfRationalFunction, 992
DenominatorRat, 217
DenseHashTable, 370
DenseIntKey, 370

deprecated, 1208
DepthOfPcElement, 626
DepthOfUpperTriangularMatrix, 327
Derangements, 207
Derivations, 954
Derivative, 998
DerivedLength, 526
DerivedSeriesOfGroup, 526
DerivedSubgroup, 511
DerivedSubgroupsTom, 1051
DerivedSubgroupsTomPossible, 1052
DerivedSubgroupsTomUnique, 1052
DerivedSubgroupTom, 1051
DescriptionOfRootOfUnity, 222
Determinant, 313
determinant
    integer matrix, 339
determinant character, 1132
DeterminantIntMat, 339
DeterminantMat, 313
DeterminantMatDestructive, 314
DeterminantMatDivFree, 314
DeterminantOfCharacter, 1132
DiagonalizeIntMat, 338
DiagonalizeMat, 323
DiagonalMat, 315
DiagonalOfMat, 326
DictionaryByPosition, 368
DicyclicGroup, 721
Difference, 390
DifferenceBlist, 295
DihedralGroup, 721
Dimension, 866
DimensionOfHighestWeightModule, 975
DimensionOfMatrixGroup, 614
DimensionOfVectors, 906
DimensionsLoewyFactors, 528
DimensionsMat, 313
DirectoriesLibrary, 126
DirectoriesPackageLibrary, 1202
DirectoriesPackagePrograms, 1202
DirectoriesSystemPrograms, 126
Directory, 125
DirectoryContents, 126
DirectoryCurrent, 125
DirectoryDesktop, 126
DirectoryHome, 127
DirectoryTemporary, 125
DirectProduct, 712
DirectProductOp, 712
DirectSumDecomposition
    for Lie algebras, 932
DirectSumOfAlgebraModules
    for a list of Lie algebra modules, 947
    for two Lie algebra modules, 947
DirectSumOfAlgebras
    for a list of algebras, 931
    for two algebras, 931
disable automatic loading, 1198
DisableAttributeValueStoring, 168
Discriminant, 998
Display, 82
    for a character table, 1092
    for a ffe, 884
    for a table of marks, 1043
    for class functions, 1124
DisplayCacheStats, 112
DisplayCompositionSeries, 525
DisplayEggBoxOfDClass, 770
DisplayImfInvariants, 749
DisplayInformationPerfectGroups
    for a pair [order, index], 740
    for group order (and index), 740
DisplayOptions, 1094
DisplayOptionsStack, 122
DisplayPackageLoadingLog, 1199
DisplayProfile, 108
DisplayString, 354
DistancePerms, 591
DistancesDistributionMatFFEVecFFE, 305
DistancesDistributionVecFFEsVecFFE, 305
DistanceVecFFE, 305
division, 57
    operation, 408
division rings, 868
DivisionRingByGenerators, 870
divisors
    of an integer, 184
DivisorsInt, 184
Dixon-Schneider algorithm, 1100
DixonInit, 1101
DixonRecord, 1101
DixonSplit, 1101
DixontinI, 1101

DMYDay, 363
DMYhmsSeconds, 364
DnLattice, 1142
DnLatticeIterative, 1143
do, 63
document formats
  for help books, 1283
document formats (text, dvi, ps, pdf, HTML), 29
Domain, 403
DomainByGenerators, 403
DomainOfPartialPerm, 819
DomainOfPartialPermCollection, 819
DominantCharacter
  for a root system and a highest weight, 974
  for a semisimple Lie algebra and a highest weight, 974
DominantWeights, 974
dot-file, 533
DotFileLatticeSubgroups, 533
DoubleCoset, 504
DoubleCosetRepsAndSizes, 505
DoubleCosets, 505
DoubleCosetsNC, 505
DoubleCoverOfAlternatingGroup, 547
DoubleHashArraySize, 371
doublequote character, 350
doublequotes, 347
DownEnv, 88
duplicate free, 271
DuplicateFreeList, 278
DxIncludeIrreducibles, 1102

E, 218
e_N (irrational value), 224
EANormalSeriesByPcgs, 635
Earns
  for a group, an action domain, etc., 581
  for an external set, 581
EB, 224
EC, 224
ED, 224
Edit, 96
EE, 224
EF, 224
EG, 224
EggBoxOfDClass, 770
EH, 224
EI, 225
Eigenspaces, 321
Eigenvalues, 321
EigenvaluesChar, 1133
Eigenvectors, 321
EJ, 226
EK, 226
EL, 226
element test
  for lists, 255
ElementaryAbelianGroup, 720
ElementaryAbelianSeries
  for a group, 526
  for a list, 526
ElementaryAbelianSeriesLargeSteps, 526
ElementaryDivisorsMat, 321
ElementaryDivisorsMatDestructive, 321
ElementaryDivisorsTransformationsMat, 322
ElementaryDivisorsTransformationsMatDestructive, 322
ElementOfFpGroup, 663
ElementOfFpSemigroup, 784
ElementOfMagmaRing, 983
ElementOrdersPowerMap, 1161
ElementProperty, 610
Elements, 386
elements
  definition, 151
  of a list or collection, 385
ElementsFamily, 380
ElementsStabChain, 607
elif, 60
EliminatedWord, 473
EliminationOrdering, 1011
ElmWPObj, 1293
ElmWPObj, 1293
else, 60
EM, 226
emacs, 96
email addresses, 27
Embedding
  for a domain and a positive integer, 422
  for group products, 718
  for two domains, 422
Embedding
  example for direct products, 713

example for semidirect products, 714
example for wreath products, 716
for Lie algebras, 952
for magma rings, 983
embeddings
find all, 562
EmptyBinaryRelation
for a degree, 438
for a domain, 438
EmptyMatrix, 315
EmptyPartialPerm, 817
EmptyPlist, 256
EmptySCTable, 918
EmptyStabChain, 609
EmptyString, 352
EnableAttributeValueStoring, 168
End, 910
end, 65
EndlineFunc, 71
Enumerator, 381
EnumeratorByBasis, 901
EnumeratorByFunctions
for a domain and a record, 382
for a family and a record, 382
EnumeratorOfCombinations, 203
EnumeratorOfTuples, 206
EnumeratorSorted, 381
environment, 65
Epicentre, 545
EpimorphismFromFreeGroup, 498
EpimorphismNilpotentQuotient, 679
EpimorphismNonabelianExteriorSquare, 545
EpimorphismPGroup, 678
EpimorphismQuotientSystem, 678
epimorphisms
find all, 561
EpimorphismSchurCover, 544
EpimorphismSolvableQuotient, 679
EqFloat, 236
equality
associative words, 470
elements of finitely presented groups, 661
for pcwords, 647
for transformations, 797
nonassociative words, 464
of booleans, 241
of records, 376
operation, 407
equality test, 56
for permutations, 590
equivalence class, 443
equivalence relation, 439, 442
EquivalenceClasses
attribute, 444
EquivalenceClassOfElement, 444
EquivalenceClassOfElementNC, 444
EquivalenceClassRelation, 443
EquivalenceRelationByPairs, 442
EquivalenceRelationByPairsNC, 442
EquivalenceRelationByPartition, 442
EquivalenceRelationByPartitionNC, 442
EquivalenceRelationByProperty, 442
EquivalenceRelationByRelation, 442
EquivalenceRelationPartition, 443
ER, 225
Erf, 236
Error, 89
ErrorCount, 89
ErrorNoTraceBack, 85
errors
syntax, 77
ES, 225
escaped characters, 349
escaping non-special characters, 350
ET, 225
EU, 225
EuclideanDegree, 856
EuclideanQuotient, 857
EuclideanRemainder, 857
Euler's totient function, 191
EulerianFunction, 524
EulerianFunctionByTom, 1054
EV, 225
EvalStraightLineProgElm, 484
EvalString, 361
evaluation, 49
strings, 360
EW, 225
EX, 225
ExactSizeConsiderFunction, 539
Excel, 146
Exec, 150
execution, 58

exit, 90
Exp, 235
Exp10, 235
Exp2, 235
Expanded form of monomials, 1015
Expm1, 236
Exponent, 524
  for a character table, 1076
exponent
  of the prime residue group, 191
exponentiation
  operation, 408
ExponentOfPcElement, 625
ExponentsConjugateLayer, 628
ExponentsOfCommutator, 628
ExponentsOfConjugate, 628
ExponentsOfPcElement, 625
ExponentsOfRelativePower, 628
ExponentSumWord, 472
ExponentSyllable, 474
ExtendedPcgs, 630
ExtendRootDirectories, 1199
ExtendStabChain, 609
Extension, 654
ExtensionNC, 654
ExtensionRepresentatives, 655
Extensions, 654
exterior power, 1144
ExteriorCentre, 545
ExteriorPowerOfAlgebraModule, 978
External representation of polynomials, 1015
ExternalOrbit, 585
ExternalOrbits
  for a group, an action domain, etc., 586
  for an external set, 586
ExternalOrbitsStabilizers
  for a group, an action domain, etc., 586
  for an external set, 586
ExternalSet, 584
ExternalSet
  computing orbits, 1288
ExternalSubset, 585
Extract, 1138
ExtraspecialGroup, 721
ExtRepDenominatorRatFun, 1016
ExtRepNumeratorRatFun, 1016
ExtRepOfObj, 1232
  for a cyclotomic, 222
ExtRepPolynomialRatFun, 1016
EY, 225

f_N (irrational value), 224
FactorCosetAction, 577
  for fp groups, 666
FactorFreeSemigroupByRelations, 782
FactorGroup, 529
FactorGroupFpGroupByRels, 660
FactorGroupNC, 529
FactorGroupNormalSubgroupClasses, 1114
FactorGroupTom, 1055
Factorial, 200
Factorization, 499
factorization, 498
Factors, 855
  for polynomials over abelian number fields, 888
  of polynomial, 1000
FactorsInt, 181
  using Pollard's Rho, 181
FactorsOfDirectProduct, 1106
FactorsSquarefree, 1001
fail, 240
FaithfulModule
  for Lie algebras, 946
FamiliesOfGeneralMappingsAndRanges, 435
FamiliesOfRows, 1113
FamilyForOrdering, 447
FamilyObj, 160
FamilyPcgs, 646
FamilyRange, 435
FamilySource, 435
features
  under UNIX, 33
fi, 60
Fibonacci, 212
Field
  for (a field and) a list of generators, 869
  for several generators, 869
field homomorphisms
  Frobenius, 882
FieldByGenerators, 870
FieldExtension, 872
FieldOfMatrixGroup, 614
FieldOverItselfByGenerators, 871

fields, 868
FileDescriptorOfStream, 135
Filename
for a directory and a string, 127
for a list of directories and a string, 127
FilenameFunc, 71
Filtered, 283
FindSl2, 969
finiteness test
for a list or collection, 386
First, 284
FittingSubgroup, 512
FixedPointsOfPartialPerm
for a partial perm, 820
for a partial perm coll, 820
Flat, 279
FlatKernelOfTransformation, 802
FLOAT
constants, 237
Float, 238
Floor, 236
flush character, 350
FlushCaches, 1237
FOA triples, 1286
for loop, 63
ForAll, 285
ForAny, 285
FpElmComparisonMethod, 662
FpGroupPresentation, 685
FpGrpMonSmgOfFpGrpMonSmgElement, 782
FpLieAlgebraByCartanMatrix, 970
Frac, 236
Frame, 1144
FrattiniSubgroup, 512
FrattiniSubgroup
for groups with pcgs, 643
FreeAbelianGroup, 721
FreeAlgebra
for ring and several names, 916
for ring, rank (and name), 916
FreeAlgebraWithOne
for ring and several names, 916
for ring, rank (and name), 916
FreeAssociativeAlgebra
for ring and several names, 917
for ring, rank (and name), 917
FreeAssociativeAlgebraWithOne
for ring and several names, 917
for ring, rank (and name), 917
FreeGeneratorsOfFpGroup, 662
FreeGeneratorsOfFpSemigroup, 784
FreeGeneratorsOfWholeGroup, 662
FreeGroup
for a list of names, 469
for given rank, 469
for infinitely many generators, 469
for various names, 469
FreeGroupOfFpGroup, 662
FreeLeftModule, 865
FreeLieAlgebra
for ring and several names, 954
for ring, rank (and name), 954
FreeMagma
for a list of names, 466
for given rank, 466
for infinitely many generators, 466
for various names, 466
FreeMagmaRing, 981
FreeMagmaWithOne
for a list of names, 466
for given rank, 466
for infinitely many generators, 466
for various names, 466
FreeMonoid
for a list of names, 762
for given rank, 762
for infinitely many generators, 762
for various names, 762
FreeMonoidOfRewritingSystem, 787
FreeProduct
for a list, 717
for several groups, 717
FreeSemigroup
for a list of names, 759
for given rank, 759
for infinitely many generators, 759
for various names, 759
FreeSemigroupOfFpSemigroup, 784
FreeSemigroupOfRewritingSystem, 787
Frobenius automorphism, 882
FrobeniusAutomorphism, 882
FrobeniusCharacterValue, 1156
FullMatrixAlgebra, 921
FullMatrixAlgebraCentralizer, 931

FullMatrixLieAlgebra, 954
FullMatrixModule, 867
FullMatrixSpace, 905
FullRowModule, 867
FullRowSpace, 905
FullTransformationMonoid, 809
FullTransformationSemigroup, 809
FunctionAction, 584
FunctionField
for an integral ring and a list of indeterminate numbers, 1007
for an integral ring and a list of indeterminates, 1007
for an integral ring and a list of names (and an exclusion list), 1007
for an integral ring and a rank (and an exclusion list), 1007
FunctionOperation, 1208
functions
as in mathematics, 419
as in programming language, 70
definition by arrow notation, 67
definition of, 65
recursive, 65
with a variable number of arguments, 55, 66
FunctionsFamily, 75
FusionCharTableTom, 1058
FusionConjugacyClasses
for a homomorphism, 1164
for two character tables, 1164
for two groups, 1164
FusionConjugacyClassesOp
for a homomorphism, 1164
for two character tables, 1164
fusions, 1163
FusionsAllowedByRestrictions, 1183
FusionsTom, 1048

G-sets, 583
computing orbits, 1288
g_N (irrational value), 224
gac, 1203
GaloisCyc
for a class function, 1123
for a cyclotomic, 228
for a list of cyclotomics, 228
GaloisField
for characteristic and degree, 881
for characteristic and polynomial, 881
for field size, 881
for subfield and degree, 881
for subfield and polynomial, 881
GaloisGroup
for abelian number fields, 892
of field, 873
of rational class of a group, 508
GaloisMat, 229
GaloisStabilizer, 889
GaloisType, 1002
Gamma, 236
GammaL, 727
gap.ini, 38
Gap3CatalogueIdGroup, 736
GAPInfo, 42
GAPInfo.Architecture, 1202
GAPInfo.CommandLineOptions, 33
GAPInfo.Keywords, 48
GAPInfo.ProfileThreshold, 108
GAPInfo.RootPaths, 124
GAPInfo.UserGapRoot, 124
GAPInfo.Version, 112
GapInputPcGroup, 652
GapInputSCTable, 919
GAPKB_REW, 786
GAPTCENUM, 666
GasmanLimits, 119
GasmanMessageStatus, 119
GasmanStatistics, 119
Gaussian algorithm, 318
GaussianIntegers, 893
GaussianRationals, 887
Gcd
for (a ring and) a list of elements, 857
for (a ring and) several elements, 857
Gcdex, 177
GcdInt, 177
GcdOp, 858
GcdRepresentation
for (a ring and) a list of elements, 858
for (a ring and) several elements, 858
GcdRepresentationOp, 858
GeneralisedEigenspaces, 321
GeneralisedEigenvalues, 320
generalized characters, 1116

generalized conjugation technique, 1296
GeneralizedEigenspaces, 321
GeneralizedEigenvalues, 320
GeneralLinearGroup
  for dimension and a ring, 724
  for dimension and field size, 724
GeneralMappingByElements, 420
GeneralMappingsFamily, 435
GeneralOrthogonalGroup, 726
GeneralSemilinearGroup, 727
GeneralUnitaryGroup, 725
generator
  of the prime residue group, 193
GeneratorsOfAdditiveGroup, 842
GeneratorsOfAdditiveMagma, 842
GeneratorsOfAdditiveMagmaWithZero, 842
GeneratorsOfAlgebra, 926
GeneratorsOfAlgebraModule, 941
GeneratorsOfAlgebraWithOne, 927
GeneratorsOfDivisionRing, 870
GeneratorsOfDomain, 402
GeneratorsOfEquivalenceRelationPartition, 443
GeneratorsOfField, 870
GeneratorsOfGroup, 492
GeneratorsOfIdeal, 849
GeneratorsOfInverseMonoid, 764
GeneratorsOfInverseSemigroup, 764
GeneratorsOfLeftIdeal, 849
GeneratorsOfLeftModule, 862
GeneratorsOfLeftOperatorAdditiveGroup, 862
GeneratorsOfLeftVectorSpace, 896
GeneratorsOfMagma, 458
GeneratorsOfMagmaWithInverses, 459
GeneratorsOfMagmaWithOne, 458
GeneratorsOfMonoid, 762
GeneratorsOfNearAdditiveGroup, 842
GeneratorsOfNearAdditiveMagma, 842
GeneratorsOfNearAdditiveMagmaWithZero, 842
GeneratorsOfPresentation, 685
GeneratorsOfRightIdeal, 850
GeneratorsOfRightModule, 863
GeneratorsOfRightOperatorAdditiveGroup, 863
GeneratorsOfRing, 846
GeneratorsOfRingWithOne, 851
GeneratorsOfRws, 487
GeneratorsOfSemigroup, 758
GeneratorsOf*Struct*, 398
GeneratorsOfTwoSidedIdeal, 849
GeneratorsOfVectorSpace, 896
GeneratorsPrimeResidues, 191
GeneratorsSmallest, 539
GeneratorsSubgroupsTom, 1056
GeneratorSyllable, 474
GetCyclotomicsLimit, 231
GetFusionMap, 1165
getting help, 28
GF
  for characteristic and degree, 881
  for characteristic and polynomial, 881
  for field size, 881
  for subfield and degree, 881
  for subfield and polynomial, 881
GL
  for dimension and a ring, 724
  for dimension and field size, 724
GlobalMersenneTwister, 188
GlobalRandomSource, 188
GModuleByMats
  for empty list, the dimension, and a field, 1027
  for generators and a field, 1027
GO, 726
GQuotients, 561
Grading, 933
graphviz, 533
GreensDClasses, 770
GreensDClassOfElement, 770
GreensDRelation, 769
GreensHClasses, 770
GreensHClassOfElement, 770
GreensHRelation, 769
GreensJClasses, 770
GreensJClassOfElement, 770
GreensJRelation, 769
GreensLClasses, 770
GreensLClassOfElement, 770
GreensLRelation, 769
GreensRClasses, 770
GreensRClassOfElement, 770
GreensRRelation, 769

GroebnerBasis
for a list and a monomial ordering, 1012
for an ideal and a monomial ordering, 1012
GroebnerBasisNC, 1012
Group
for a list of generators (and an identity element), 492
for several generators, 492
group actions, 566, 567
operations syntax, 566
group algebra, 980
group characters, 1116
group operations, 567, 1208
group ring, 980
GroupByGenerators, 492
with explicitly specified identity element, 492
GroupByRws, 650
GroupByRwsNC, 650
GroupGeneralMappingByImages, 550
from group to itself, 550
GroupGeneralMappingByImagesNC, 550
from group to itself, 550
GroupHClassOfGreensDClass, 770
GroupHomomorphismByFunction
by function (and inverse function) between two domains, 551
by function and function that computes one preimage, 551
GroupHomomorphismByImages, 549
GroupHomomorphismByImagesNC, 550
GroupOfPcgs, 625
GroupRing, 981
GroupStabChain, 607
GroupWithGenerators, 492
GrowthFunctionOfGroup, 499
with word length limit, 499
GU, 725

h_N (irrational value), 224
HallSubgroup, 515
HallSystem, 516
HallSystem
for groups with pcgs, 643
HasAbelianFactorGroup, 530
HasElementaryAbelianFactorGroup, 530
HasIndeterminateName, 989
HasParent, 401
HasseDiagramBinaryRelation, 441
HeadPcElementByNumber, 627
HELP_ADD_BOOK, 1282
HELP_REMOVE_BOOK, 1283
HELP_VIEWER_INFO, 1285
HenselBound, 1003
Hermite normal form, 1209
HermiteNormalFormIntegerMat, 337
HermiteNormalFormIntegerMatTransform, 337
HeuristicCancelPolynomials, 1019
HexStringInt, 355
HighestWeightModule, 977
HMSMSec, 363
Hom, 910
HomeEnumerator, 585
HomomorphismQuotientSemigroup, 768
homomorphisms
find all, 560
Frobenius, field, 882
Hypothenuse, 236

i_N (irrational value), 225
Ideal, 847
IdealByGenerators, 848
IdealNC, 848
Idempotent, 792
Idempotents, 459
IdempotentsTom, 1049
IdempotentsTomInfo, 1049
IdentificationOfConjugacyClasses, 1071
Identifier
for character tables, 1080
for tables of marks, 1049
Identity, 404
IdentityBinaryRelation
for a degree, 437
for a domain, 437
IdentityFromSCTable, 920
IdentityMapping, 422
IdentityMat, 315
IdentityTransformation, 794
IdFunc, 74
IdGap3SolvableGroup, 736
IdGroup, 735
IdSmallGroup, 735

IdsOfAllSmallGroups, 735
if statement, 60
Image
    set of images of a collection under a mapping, 425
    set of images of the source of a general mapping, 425
    unique image of an element under a mapping, 425
Image
    for Frobenius automorphisms, 883
image
    vector under matrix, 312
ImageElm, 425
ImageListOfPartialPerm, 820
ImageListOfTransformation, 799
ImageOfPartialPermCollection, 819
Images
    set of images of a collection under a mapping, 425
    set of images of an element under a mapping, 425
    set of images of the source of a general mapping, 425
ImagesElm, 424
ImageSetOfPartialPerm, 820
ImageSetOfTransformation, 799
ImagesRepresentative, 424
ImagesSet, 425
ImagesSmallestGenerators, 554
ImagesSource, 424
ImaginaryPart, 228
ImfInvariants, 752
ImfMatrixGroup, 753
ImfNumberQClasses, 749
ImfNumberQQClasses, 749
ImfNumberZClasses, 749
Immutable, 155
ImmutableBasis, 903
ImmutableMatrix, 330
\in
    element test for lists, 255
    for a collection, 390
    for strictly sorted lists, 275
\in
    operation for testing membership, 390
in
    for lists, 255
    operation for, 390
IncreaseInterval, 236
IndependentGeneratorExponents, 540
IndependentGeneratorsOfAbelianGroup, 540
Indeterminate
    for a family and a number, 988
    for a ring (and a name, and an exclusion list), 988
    for a ring (and a number), 988
IndeterminateName, 989
Indeterminateness, 1175
IndeterminateNumberOfLaurentPolynomial, 1004
IndeterminateNumberOfUnivariateRationalFunction, 989
IndeterminateOfUnivariateRationalFunction, 989
IndeterminatesOfFunctionField, 1005
IndeterminatesOfPolynomialRing, 1005
Index
    for a group and its subgroup, 494
    for two character tables, 1089
IndexInWholeGroup, 494
IndexNC
    for a group and its subgroup, 494
IndexPeriodOfPartialPerm, 824
IndexPeriodOfTransformation, 803
Indicator, 1090
IndicatorOp, 1090
IndicesCentralNormalSteps, 635
IndicesChiefNormalSteps, 637
IndicesEANormalSteps, 634
IndicesInvolutaryGenerators, 668
IndicesNormalSteps, 638
IndicesOfAdjointBasis, 928
IndicesPCentralNormalStepsPGroup, 636
IndicesStabChain, 607
Indirected, 1171
InducedAutomorphism, 559
InducedClassFunction
    for a given monomorphism, 1135
    for a supergroup, 1135
    for the character table of a supergroup, 1135
InducedClassFunctions, 1135
InducedClassFunctionsByFusionMap, 1135

InducedCyclic, 1136
InducedPcgs, 629
InducedPcgsByGenerators, 630
InducedPcgsByGeneratorsNC, 630
InducedPcgsByPcSequence, 629
InducedPcgsByPcSequenceAndGenerators, 630
InducedPcgsByPcSequenceNC, 629
InducedPcgsWrtFamilyPcgs, 646
InducedPcgsWrtSpecialPcgs, 641
Inequalities, 1155
inequality
  of booleans, 241
  of records, 376
inequality test, 56
InertiaSubgroup, 1131
Inf, 236
infinity, 223
inflated class functions, 1134
Info, 103
InfoAlgebra, 915
InfoAttributes, 167
InfoBckt, 611
InfoCharacterTable, 1069
InfoCoh, 543
InfoComplement, 511
InfoCoset, 506
InfoFpGroup, 660
InfoGroebner, 1013
InfoGroup, 494
InfoLattice, 536
InfoLevel, 103
InfoMatrix, 309
InfoMonomial, 1189
InfoNumtheor, 190
InfoObsolete, 1209
InfoOptions, 122
InfoPackageLoading, 1199
InfoPcSubgroup, 539
InfoPoly, 997
InfoText, 1080
InfoTom, 1045
InfoWarning, 105
Init, 188
InitFusion, 1182
InitPowerMap, 1179
InjectionZeroMagma, 455
inner product
  of group characters, 1129
InnerAutomorphism, 556
InnerAutomorphismNC, 556
InnerAutomorphismsAutomorphismGroup, 559
InParentFOA, 1287
InputFromUser, 142
InputLogTo
  for a filename, 131
  for streams, 140
  stop logging input, 131
InputOutputLocalProcess, 144
InputTextFile, 141
InputTextNone, 145
InputTextString, 143
InputTextUser, 142
InsertTrivialStabilizer, 609
InstallAtExit, 90
InstallCharReadHookFunc, 146
InstalledPackageVersion, 1201
InstallFactorMaintenance, 412
InstallFlushableValue, 1236
InstallGlobalFunction, 1235
InstallHandlingByNiceBasis, 913
InstallImmediateMethod, 1214
InstallIsomorphismMaintenance, 412
InstallMethod, 1212
InstallOtherMethod, 1212
InstallReadlineMacro, 95
InstallSubsetMaintenance, 411
InstallTrueMethod, 1215
InstallValue, 1236
Int, 173
  for a cyclotomic, 220
  for a FFE, 879
  for strings, 360
IntChar, 359
integer part of a quotient, 176
Integers
  global variable, 172
IntegralizedMat, 341
IntegratedStraightLineProgram, 481
IntermediateGroup, 528
IntermediateResultOfSLP, 482
IntermediateResultOfSLPWithoutOverwrite, 482

IntermediateResultsOfSLPWithoutOverwrite, 482
IntermediateSubgroups, 528
InterpolatedPolynomial, 860
IntersectBlist, 296
Intersection
for a list, 389
for various collections, 389
Intersection
for groups with pcgs, 643
intersection
of collections, 389
of sets, 276
Intersection2, 389
IntersectionBlist
for a list, 295
for various boolean lists, 295
IntersectionsTom, 1054
IntersectSet, 276
IntFFE, 879
IntFFESymm
for a FFE, 880
for a vector of FFEs, 880
IntHexString, 360
IntScalarProducts, 1176
IntVecFFE, 880
InvariantBilinearForm, 617
InvariantElementaryAbelianSeries, 527
InvariantLattice, 619
InvariantQuadraticForm, 617
InvariantSesquilinearForm, 617
InvariantSubgroupsElementaryAbelianGroup, 537
Inverse
group homomorphism, 552
Inverse, 406
for a pcword, 647
for a transformation, 803
inverse
matrix, 312
of class function, 1121
InverseAttr, 406
InverseClasses, 1080
InverseGeneralMapping, 420
InverseImmutable, 406
InverseMap, 1170
InverseMatMod, 332
InverseMonoid, 763
InverseMutable, 406
InverseOfTransformation, 803
InverseOp, 406
InverseRepresentative, 607
InverseSameMutability, 406
InverseSemigroup, 763
InverseSM, 406
InversesOfSemigroupElement, 765
InvocationReadlineMacro, 95
Irr
for a character table, 1074
for a group, 1074
irrationalities, 218
IrrBaumClausen, 1097
IrrConlon, 1097
IrrDixonSchneider, 1097
irreducible character, 1128
irreducible characters
computation, 1100
IrreducibleDifferences, 1137
IrreducibleModules, 1099
IrreducibleModules
for groups with pcgs, 643
IrreducibleRepresentations, 1098
IrreducibleRepresentationsDixon, 1099
IrreducibleSolvableGroup, 747
IrreducibleSolvableGroupMS, 746
Is16BitsFamily, 476
Is32BitsFamily, 476
IsAbelian, 460
for a character table, 1076
IsAbelianNumberField, 888
IsAbelianNumberFieldPolynomialRing, 1006
IsAbelianTom, 1051
IsAdditiveElement, 413
IsAdditiveElementWithInverse, 414
IsAdditiveElementWithZero, 413
IsAdditiveGroup, 839
IsAdditiveGroupGeneralMapping, 431
IsAdditiveGroupHomomorphism, 431
IsAdditivelyCommutative, 842
IsAdditivelyCommutativeElement, 417
IsAdditivelyCommutativeElementCollColl, 417

IsAdditivelyCommutativeElement-
Collection, 417
IsAdditivelyCommutativeElementFamily,
417
IsAdditiveMagma, 839
IsAdditiveMagmaWithInverses, 839
IsAdditiveMagmaWithZero, 839
IsAlgebra, 925
IsAlgebraGeneralMapping, 433
IsAlgebraHomomorphism, 433
IsAlgebraicElement, 1022
IsAlgebraicExtension, 1020
IsAlgebraModuleElement, 942
IsAlgebraModuleElementCollection, 942
IsAlgebraModuleElementFamily, 942
IsAlgebraWithOne, 925
IsAlgebraWithOneGeneralMapping, 433
IsAlgebraWithOneHomomorphism, 433
IsAlmostSimple
for a character table, 1076
IsAlmostSimpleGroup, 518
IsAlphaChar, 353
IsAlternatingGroup, 598
IsAnticommutative, 853
IsAntisymmetricBinaryRelation, 438
IsAssociated, 854
IsAssociative, 460
IsAssociativeElement, 416
IsAssociativeElementCollColl, 416
IsAssociativeElementCollection, 416
IsAssocWord, 469
IsAssocWordWithInverse, 469
IsAssocWordWithOne, 469
IsAttributeStoringRep, 1243
IsAutomorphismGroup, 558
IsBasicWreathLessThanOrEqual, 471
IsBasicWreathProductOrdering, 450
IsBasis, 898
IsBasisByNiceBasis, 913
IsBasisOfAlgebraModuleElementSpace, 943
IsBergerCondition
for a character, 1189
for a group, 1189
IsBijective, 423
IsBinaryRelation, 437
IsBinaryRelation
same as IsEndoGeneralMapping, 437
IsBLetterAssocWordRep, 475
IsBLetterWordsFamily, 475
IsBlist, 293
IsBlistRep, 297
IsBlockMatrixRep, 333
IsBool, 240
IsBound
for a global variable, 51
for a list position, 252
for a record component, 377
IsBound\., 378
IsBound\[\], 246
IsBoundElmWPObj, 1293
IsBoundGlobal, 52
IsBrauerTable, 1068
IsBravaisGroup, 620
IsBuiltFromAdditiveMagmaWithInverses,
488
IsBuiltFromGroup, 488
IsBuiltFromMagma, 488
IsBuiltFromMagmaWithInverses, 488
IsBuiltFromMagmaWithOne, 488
IsBuiltFromSemigroup, 488
IsCanonicalBasis, 901
IsCanonicalBasisFullMatrixModule, 907
IsCanonicalBasisFullRowModule, 907
IsCanonicalNiceMonomorphism, 556
IsCanonicalPcgs, 631
IsCentral, 460
IsCentralFactor, 545
IsChar, 347
IsCharacter, 1128
IsCharacteristicSubgroup, 496
IsCharacterTable, 1068
IsCharacterTableInProgress, 1068
IsCharCollection, 347
IsCheapConwayPolynomial, 884
IsClassFunction, 1118
IsClassFusionOfNormalSubgroup, 1090
IsClosedStream, 133
IsCochain, 972
IsCochainCollection, 972
IsCollection, 379
IsCollectionFamily, 380
IsCommutative, 460
IsCommutativeElement, 417
IsCommutativeElementCollColl, 417

IsCommutativeElementCollection, 417
IsComponentObjectRep, 1243
IsCompositionMappingRep, 421
IsConfluent
  for a rewriting system, 486
  for an algebra with canonical rewriting system, 486
  for pc groups, 650
IsConjugacyClassSubgroupsByStabilizerRep, 531
IsConjugacyClassSubgroupsRep, 531
IsConjugate
  for a group and two elements, 509
  for a group and two groups, 509
IsConjugatorAutomorphism, 557
IsConjugatorIsomorphism, 557
IsConstantRationalFunction, 994
IsConstantTimeAccessGeneralMapping, 433
IsConstantTimeAccessList, 246
IsContainedInSpan, 904
IsCopyable, 154
IsCyc, 219
IsCyclic, 517
  for a character table, 1076
IsCyclicTom, 1051
IsCyclotomic, 219
IsCyclotomicField, 888
IsCyclotomicMatrixGroup, 619
IsDenseList, 245
IsDiagonalMat, 314
IsDictionary, 369
IsDigitChar, 352
IsDirectory, 125
IsDirectoryPath, 129
IsDirectProductElement, 419
IsDisjoint, 236
IsDistributive, 853
IsDivisionRing, 868
IsDomain, 402
IsDoneIterator, 393
IsDoubleCoset
  operation, 505
IsDuplicateFree, 271
IsDuplicateFreeList, 271
IsDxLargeGroup, 1102
IsElementaryAbelian, 517
  for a character table, 1076
IsElementOfFpMonoid, 781
IsElementOfFpSemigroup, 781
IsElementOfFreeMagmaRing, 983
IsElementOfFreeMagmaRingCollection, 983
IsElementOfFreeMagmaRingFamily, 983
IsElementOfMagmaRingModuloRelations, 984
IsElementOfMagmaRingModuloRelationsCollection, 984
IsElementOfMagmaRingModuloRelationsFamily, 984
IsElementOfMagmaRingModuloSpanOfZeroFamily, 985
IsEmpty, 386
IsEmptyString, 351
IsEndOfStream, 138
IsEndoGeneralMapping, 434
IsEndoGeneralMapping
  same as IsBinaryRelation, 437
IsEqualSet, 275
IsEquivalenceClass, 443
IsEquivalenceRelation, 439
IsEuclideanRing, 856
IsEvenInt, 174
IsExecutableFile, 129
IsExistingFile, 128
IsExtAElement, 413
IsExternalOrbit, 585
IsExternalSet, 584
IsExternalSubset, 585
IsExtLElement, 414
IsExtRElement, 414
IsFamilyPcgs, 646
IsFFE, 876
IsFFECollColl, 876
IsFFECollCollColl, 876
IsFFECollection, 876
IsFFEMatrixGroup, 614
IsField, 868
IsFieldControlledByGaloisGroup, 872
IsFieldHomomorphism, 433
IsFinite, 386
  float, 237
  for a character table, 1076
IsFiniteDimensional, 866
  for matrix algebras, 926
IsFiniteFieldPolynomialRing, 1006

IsFinitelyGeneratedGroup, 522
IsFiniteOrderElement, 417
IsFiniteOrderElementCollColl, 417
IsFiniteOrderElementCollection, 417
IsFiniteOrdersPcgs, 624
IsFixedStabilizer, 610
IsFLMLOR, 924
IsFLMLORWithOne, 924
IsFpGroup, 659
IsFpMonoid, 781
IsFpSemigroup, 781
IsFreeGroup, 469
IsFreeLeftModule, 865
IsFreeMagmaRing, 981
IsFreeMagmaRingWithOne, 982
IsFromFpGroupGeneralMappingByImages, 565
IsFromFpGroupHomomorphismByImages, 565
IsFromFpGroupStdGensGeneralMappingByImages, 565
IsFromFpGroupStdGensHomomorphismByImages, 565
IsFullHomModule, 910
IsFullMatrixModule, 867
IsFullRowModule, 867
IsFullSubgroupGLorSLRespectingBilinearForm, 617
IsFullSubgroupGLorSLRespectingQuadraticForm, 618
IsFullSubgroupGLorSLRespectingSesquilinearForm, 617
IsFullTransformationMonoid, 809
IsFullTransformationSemigroup, 809
IsFunction, 75
IsFunctionField, 1007
IsGAPRandomSource, 188
IsGaussianIntegers, 893
IsGaussianRationals, 887
IsGaussianSpace, 905
IsGaussInt, 222
IsGaussRat, 223
IsGeneralizedDomain, 402
IsGeneralizedRowVector, 258
IsGeneralLinearGroup, 616
IsGeneralMapping, 433
IsGeneralMappingFamily, 435
IsGeneratorsOfSemigroup, 759
IsGeneratorsOf*Struct*, 398
IsGL, 616
IsGlobalRandomSource, 188
IsGreensClass, 769
IsGreensDClass, 769
IsGreensDRelation, 769
IsGreensHClass, 769
IsGreensHRelation, 769
IsGreensJClass, 769
IsGreensJRelation, 769
IsGreensLClass, 769
IsGreensLessThanOrEqual, 769
IsGreensLRelation, 769
IsGreensRClass, 769
IsGreensRelation, 769
IsGreensRRelation, 769
IsGroup, 493
IsGroupGeneralMapping, 430
IsGroupGeneralMappingByAsGroupGeneralMappingByImages, 564
IsGroupGeneralMappingByImages, 564
IsGroupGeneralMappingByPcgs, 565
IsGroupHClass, 771
IsGroupHomomorphism, 430
IsGroupOfAutomorphisms, 558
IsGroupRing, 982
IsHandledByNiceBasis, 913
IsHandledByNiceMonomorphism, 555
IsHasseDiagram, 439
IsHomogeneousList, 245
IsIdempotent, 406
IsIdenticalObj, 152
IsIncomparableUnder, 446
IsInducedFromNormalSubgroup, 1192
IsInducedPcgs, 629
IsInducedPcgsWrtSpecialPcgs, 641
IsInfBitsFamily, 476
IsInfiniteAbelianizationGroup, 683
IsInfiniteAbelianizationGroup for groups, 683
IsInfinity, 223
IsInjective, 423
IsInjectiveListTrans, 797
IsInnerAutomorphism, 557
IsInputOutputStream, 144
IsInputStream, 134
IsInputTextNone, 134

IsInputTextStream, 134
IsInt, 173
IsIntegerMatrixGroup, 619
IsIntegers, 173
IsIntegralBasis, 902
IsIntegralCyclotomic, 219
IsIntegralRing, 852
IsInternallyConsistent, 158
    for character tables, 1089
    for tables of marks, 1051
IsInverseMonoid, 766
IsInverseSemigroup, 766
IsInverseSubsemigroup, 764
IsIrreducibleCharacter, 1128
IsIrreducibleRingElement, 855
IsIterator, 393
IsJacobianElement, 417
IsJacobianElementCollColl, 417
IsJacobianElementCollection, 417
IsJacobianRing, 853
IsLaurentPolynomial, 994
IsLaurentPolynomialDefaultRep, 1017
IsLDistributive, 852
IsLeftAlgebraModuleElement, 942
IsLeftAlgebraModuleElementCollection, 942
IsLeftIdeal, 848
IsLeftIdealInParent, 848
IsLeftModule, 862
IsLeftModuleGeneralMapping, 432
IsLeftModuleHomomorphism, 432
IsLeftOperatorAdditiveGroup, 862
IsLeftSemigroupIdeal, 767
IsLeftVectorSpace, 894
IsLessThanOrEqualUnder, 447
IsLessThanUnder, 447
IsLetterAssocWordRep, 475
IsLetterWordsFamily, 475
IsLexicographicallyLess, 280
IsLexOrderedFFE, 878
IsLieAbelian, 958
IsLieAlgebra, 925
IsLieMatrix, 310
IsLieNilpotent, 958
IsLieObject, 951
IsLieObjectCollection, 951
IsLieSolvable, 959
IsLinearMapping, 432
IsLinearMappingsModule, 911
IsList, 244
IsListDefault, 259
IsListOrCollection, 381
IsLogOrderedFFE, 878
IsLookupDictionary, 369
IsLowerAlphaChar, 352
IsLowerTriangularMat, 314
IsMagma, 452
IsMagmaHomomorphism, 429
IsMagmaRingModuloRelations, 985
IsMagmaRingModuloSpanOfZero, 985
IsMagmaRingObjDefaultRep, 982
IsMagmaWithInverses, 453
IsMagmaWithInversesIfNonzero, 452
IsMagmaWithOne, 452
IsMagmaWithZeroAdjoined, 455
IsMapping, 423
IsMatchingSublist, 271
IsMatrix, 309
IsMatrixGroup, 613
IsMatrixModule, 867
IsMatrixSpace, 905
IsMersenneTwister, 188
IsMinimalNonmonomial, 1196
IsModuloPcgs, 631
IsMonoid, 761
IsMonomial
    for a character table, 1076
IsMonomial
    for positive integers, 1193
IsMonomialGroup, 518
IsMonomialMatrix, 314
IsMonomialNumber, 1193
IsMonomialOrdering, 1008
IsMultiplicativeElement, 414
IsMultiplicativeElementWithInverse, 415
IsMultiplicativeElementWithOne, 414
IsMultiplicativeElementWithZero, 414
IsMultiplicativeGeneralizedRowVector, 259
IsMultiplicativeZero, 460
IsMutable, 154
IsMutableBasis, 903
IsNaN, 237
IsNaturalAlternatingGroup, 597

IsNaturalGL, 616
IsNaturalGLnZ, 619
IsNaturalSL, 616
IsNaturalSLnZ, 619
IsNaturalSymmetricGroup, 597
IsNearAdditiveElement, 413
IsNearAdditiveElementWithInverse, 413
IsNearAdditiveElementWithZero, 413
IsNearAdditiveGroup, 838
IsNearAdditiveMagma, 838
IsNearAdditiveMagmaWithInverses, 838
IsNearAdditiveMagmaWithZero, 838
IsNearlyCharacterTable, 1068
IsNearRingElement, 415
IsNearRingElementWithInverse, 416
IsNearRingElementWithOne, 415
IsNegInfinity, 223
IsNegRat, 216
IsNilpotent
    for a character table, 1076
IsNilpotent
    for groups with pcgs, 643
IsNilpotentElement, 968
IsNilpotentGroup, 517
IsNilpotentTom, 1051
IsNInfinity, 237
IsNonassocWord, 463
IsNonassocWordCollection, 464
IsNonassocWordWithOne, 463
IsNonassocWordWithOneCollection, 464
IsNonnegativeIntegers, 173
IsNonSPGeneralMapping, 435
IsNonTrivial, 386
IsNormal, 495
IsNormalBasis, 902
IsNotIdenticalObj, 153
IsNumberField, 888
IsObject, 151
IsOddInt, 174
isomorphic
    pc group, 650, 651
IsomorphicSubgroups, 562
IsomorphismFpAlgebra, 938
IsomorphismFpGroup, 672
IsomorphismFpGroup
    for subgroups of fp groups, 675
IsomorphismFpGroupByGenerators, 673
IsomorphismFpGroupByGeneratorsNC, 673
IsomorphismFpGroupByPcgs, 648
IsomorphismFpSemigroup, 783
IsomorphismGroups, 560
IsomorphismMatrixAlgebra, 938
IsomorphismPartialPermMonoid, 836
IsomorphismPartialPermSemigroup, 836
IsomorphismPcGroup, 651
IsomorphismPermGroup, 596
    for Imf matrix groups, 754
IsomorphismPermGroupImfGroup, 755
IsomorphismReesMatrixSemigroup, 773
IsomorphismRefinedPcGroup, 650
Isomorphism*RepStruct*, 399
isomorphisms
    find all, 560
IsomorphismSCAlgebra
    for an algebra, 939
    w.r.t. a given basis, 939
IsomorphismSimplifiedFpGroup, 676
IsomorphismSpecialPcGroup, 652
IsomorphismTransformationMonoid, 810
IsomorphismTransformationSemigroup, 810
IsomorphismTypeInfoFiniteSimpleGroup
    for a character table, 1076
    for a group, 519
    for a group order, 519
IsOne, 405
IsOperation, 75
IsOrdering, 445
IsOrderingOnFamilyOfAssocWords, 448
IsOrdinaryMatrix, 310
IsOrdinaryTable, 1068
IsOutputStream, 134
IsOutputTextNone, 134
IsOutputTextStream, 134
IsPackageMarkedForLoading, 1203
IsPadicExtensionNumber, 1026
IsPadicExtensionNumberFamily, 1026
IsParentPcgsFamilyPcgs, 646
IsPartialOrderBinaryRelation, 439
IsPartialPerm, 814
IsPartialPermCollection, 814
IsPartialPermMonoid, 833
IsPartialPermSemigroup, 833
IsPcGroup, 647
IsPcGroupGeneralMappingByImages, 565

IsPcGroupHomomorphismByImages, 565
IsPcgs, 623
IsPcgsCentralSeries, 635
IsPcgsChiefSeries, 637
IsPcgsElementaryAbelianSeries, 634
IsPcgsPCentralSeriesPGroup, 636
IsPerfect
for a character table, 1076
IsPerfectGroup, 517
IsPerfectTom, 1051
IsPerm, 590
IsPermCollColl, 590
IsPermCollection, 590
IsPermGroup, 595
IsPermGroupGeneralMapping, 564
IsPermGroupGeneralMappingByImages, 564
IsPermGroupHomomorphism, 564
IsPermGroupHomomorphismByImages, 564
IsPGroup, 522
IsPInfinity, 237
IsPNilpotent, 523
IsPolycyclicGroup, 518
IsPolynomial, 993
IsPolynomialDefaultRep, 1016
IsPolynomialFunction, 992
IsPolynomialFunctionsFamily, 1014
IsPolynomialRing, 1006
IsPosInt, 173
IsPositiveIntegers, 173
IsPosRat, 216
IsPreimagesByAsGroupGeneralMappingBy-
Images, 564
IsPreOrderBinaryRelation, 439
IsPrime, 855
IsPrimeField, 871
IsPrimeInt, 179
IsPrimeOrdersPcgs, 624
IsPrimePowerInt, 180
IsPrimitive
for a group, an action domain, etc., 582
for an external set, 582
IsPrimitiveCharacter, 1191
IsPrimitivePolynomial, 994
IsPrimitiveRootMod, 193
IsProbablyPrimeInt, 179
IsPseudoCanonicalBasisFullHomModule,
911
IsPSolubleCharacterTable, 1089
IsPSolubleCharacterTableOp, 1089
IsPSolvable, 523
IsPSolvableCharacterTable, 1089
IsPSolvableCharacterTableOp, 1089
IsPurePadicNumber, 1024
IsPurePadicNumberFamily, 1024
IsQuasiPrimitive, 1191
IsQuaternion, 926
IsQuaternionCollColl, 926
IsQuaternionCollection, 926
IsQuickPositionList, 292
IsQuotientSemigroup, 768
IsRandomSource, 187
IsRange, 290
IsRat, 216
IsRationalFunction, 992
IsRationalFunctionDefaultRep, 1016
IsRationalFunctionsFamily, 1014
IsRationalMatrixGroup, 619
IsRationals, 215
IsRationalsPolynomialRing, 1006
IsRDistributive, 852
IsReadableFile, 128
IsReadOnlyGlobal, 51
IsRecord, 372
IsRecordCollColl, 372
IsRecordCollection, 372
IsRectangularTable, 246
IsReduced, 486
IsReductionOrdering, 448
IsReesCongruence, 767
IsReesCongruenceSemigroup, 766
IsReesMatrixSemigroup, 775
IsReesMatrixSemigroupElement, 774
IsReesMatrixSubsemigroup, 775
IsReesZeroMatrixSemigroup, 775
IsReesZeroMatrixSemigroupElement, 774
IsReesZeroMatrixSubsemigroup, 775
IsReflexiveBinaryRelation, 438
IsRegular
for a group, an action domain, etc., 581
for an external set, 581
IsRegularDClass, 771
IsRegularSemigroup, 765
IsRegularSemigroupElement, 765
IsRelativelySM

for a character, 1195
for a group, 1195
IsRestrictedJacobianElement, 417
IsRestrictedJacobianElementCollColl, 417
IsRestrictedJacobianElementCollection, 417
IsRestrictedLieAlgebra, 965
IsRestrictedLieObject, 951
IsRestrictedLieObjectCollection, 951
IsRewritingSystem, 485
IsRightAlgebraModuleElement, 942
IsRightAlgebraModuleElementCollection, 942
IsRightCoset, 503
IsRightIdeal, 848
IsRightIdealInParent, 848
IsRightModule, 863
IsRightOperatorAdditiveGroup, 863
IsRightSemigroupIdeal, 767
IsRing, 844
IsRingElement, 415
IsRingElementWithInverse, 416
IsRingElementWithOne, 416
IsRingGeneralMapping, 432
IsRingHomomorphism, 432
IsRingWithOne, 850
IsRingWithOneGeneralMapping, 433
IsRingWithOneHomomorphism, 433
IsRootSystem, 960
IsRootSystemFromLieAlgebra, 960
IsRowModule, 866
IsRowSpace, 904
IsRowVector, 299
IsScalar, 416
IsSemiEchelonized, 906
IsSemigroup, 757
IsSemigroupCongruence, 767
IsSemigroupIdeal, 767
IsSemiRegular
for a group, an action domain, etc., 581
for an external set, 581
IsSet, 271
IsShortLexLessThanOrEqual, 471
IsShortLexOrdering, 449
IsSimple
for a character table, 1076
IsSimpleAlgebra, 925
IsSimpleGroup, 518
IsSimpleSemigroup, 766
IsSingleValued, 423
IsSL, 616
IsSolvable
for a character table, 1076
IsSolvableGroup, 517
IsSolvableTom, 1051
IsSortedList, 271
IsSpecialLinearGroup, 616
IsSpecialPcgs, 639
IsSPGeneralMapping, 435
IsSporadicSimple
for a character table, 1076
IsSSortedList, 271
IsStraightLineProgElm, 483
IsStraightLineProgram, 478
IsStream, 133
IsString, 347
IsStringRep, 351
Is*Struct*, 400
IsSubgroup, 495
IsSubgroupFpGroup, 659
IsSubgroupOfWholeGroupByQuotientRep, 677
IsSubgroupSL, 616
IsSubmonoidFpMonoid, 781
IsSubnormal, 496
IsSubnormallyMonomial
for a character, 1194
for a group, 1194
IsSubsemigroup, 758
IsSubsemigroupFpSemigroup, 781
IsSubset, 388
IsSubsetBlist, 295
IsSubsetLocallyFiniteGroup, 522
IsSubsetSet, 275
IsSubspacesVectorSpace, 896
IsSub*struct*, 402
IsSupersolvable
for a character table, 1076
IsSupersolvable
for groups with pcgs, 643
IsSupersolvableGroup, 518
IsSurjective, 423
IsSyllableAssocWordRep, 476

IsSyllableWordsFamily, 476
IsSymmetricBinaryRelation, 438
IsSymmetricGroup, 598
IsSymmetricInverseMonoid, 835
IsSymmetricInverseSemigroup, 835
IsTable, 245
IsTableOfMarks, 1045
IsTableOfMarksWithGens, 1057
IsToPcGroupGeneralMappingByImages, 565
IsToPcGroupHomomorphismByImages, 565
IsToPermGroupGeneralMappingByImages, 564
IsToPermGroupHomomorphismByImages, 564
IsTotal, 422
IsTotalOrdering, 446
IsTransformation, 790
IsTransformationCollection, 790
IsTransformationMonoid, 808
IsTransformationSemigroup, 808
IsTransitive
  for a character, 1132
  for a group, an action domain, etc., 580
  for a permutation group, 580
  for an external set, 580
IsTransitiveBinaryRelation, 438
IsTranslationInvariantOrdering, 448
IsTrivial, 386
IsTwoSidedIdeal, 848
IsTwoSidedIdealInParent, 848
IsUEALatticeElement, 976
IsUEALatticeElementCollection, 976
IsUEALatticeElementFamily, 976
IsUniqueFactorizationRing, 852
IsUnit, 853
IsUnivariatePolynomial, 994
IsUnivariatePolynomialRing, 1007
IsUnivariateRationalFunction, 993
IsUnknown, 1186
IsUpperAlphaChar, 352
IsUpperTriangularMat, 314
IsValidIdentifier, 49
IsVector, 415
IsVectorSpace, 894
IsVirtualCharacter, 1128
IsWeightLexOrdering, 450
IsWeightRepElement, 977
IsWeightRepElementCollection, 977
IsWeightRepElementFamily, 977
IsWellFoundedOrdering, 446
IsWeylGroup, 962
IsWholeFamily, 387
IsWLetterAssocWordRep, 475
IsWLetterWordsFamily, 475
IsWord, 462
IsWordCollection, 463
IsWordWithInverse, 462
IsWordWithOne, 462
IsWreathProductOrdering, 451
IsWritableFile, 129
IsXInfinity, 237
IsZero, 406
IsZeroGroup, 766
IsZeroSimpleSemigroup, 766
IsZeroSquaredElement, 418
IsZeroSquaredElementCollColl, 418
IsZeroSquaredElementCollection, 418
IsZeroSquaredRing, 853
IsZmodnZObj, 185
IsZmodnZObjNonprime, 185
IsZmodpZObj, 185
IsZmodpZObjLarge, 185
IsZmodpZObjSmall, 185
Iterated, 287
Iterator, 392
iterator
  for low index subgroups, 671
IteratorByBasis, 901
IteratorByFunctions, 394
IteratorList, 394
IteratorOfCartesianProduct
  for a list of lists, 282
  for several lists, 282
IteratorOfCombinations, 203
IteratorOfPartitions, 209
IteratorOfTuples, 206
IteratorSorted, 393

j_N (irrational value), 226
Jacobi, 193
JenningsLieAlgebra, 966
JenningsSeries, 527
JoinEquivalenceRelations, 443
JoinOfIdempotentPartialPermsNC, 816
JoinOfPartialPerms, 816

JoinStringsWithSeparator, 358
JordanDecomposition, 328

k_N (irrational value), 226
KappaPerp, 968
KB_REW, 786
kernel
  group homomorphism, 552
  of a matrix, 319
KernelOfAdditiveGeneralMapping, 431
KernelOfCharacter, 1130
KernelOfMultiplicativeGeneralMapping, 430
KernelOfTransformation, 802
KeyDependentOperation, 1286
KillingMatrix, 968
KnownAttributesOfObject, 165
KnownPropertiesOfObject, 169
KnownTruePropertiesOfObject, 169
KnowsDictionary, 369
KnowsHowToDecompose, 548
KnuthBendixRewritingSystem
  for a monoid and a reduction ordering, 787
  for a semigroup and a reduction ordering, 787
Krasner-Kaloujnine theorem, 717
KroneckerProduct, 317
KuKGenerators, 717

l_N (irrational value), 226
Lambda, 191
larger or equal, 56
larger test, 56
LargerQuotientBySubgroupAbelianization, 680
LargestElementGroup, 539
LargestElementStabChain, 608
LargestImageOfMovedPoint
  for a partial permutation, 823
  for a partial permutation coll, 823
  for a transformation, 802
  for a transformation coll, 802
LargestMovedPoint
  for a list or collection of permutations, 591
  for a partial perm, 822
  for a partial perm coll, 822
  for a permutation, 591
  for a transformation, 801
  for a transformation coll, 801
LargestUnknown, 1186
last, 77
last2, 77
last3, 77
LastSystemError, 124
LaTeX
  for a decomposition matrix, 1087
  for GAP objects, 364
  for permutation characters, 1148
  for the result of a straight line program, 480
LaTeXStringDecompositionMatrix, 1087
lattice base reduction, 341, 342
lattice basis reduction
  for virtual characters, 1137
LatticeByCyclicExtension, 536
LatticeGeneratorsInUEA, 976
LatticeSubgroups, 533
LatticeSubgroupsByTom, 1042
LaurentPolynomialByCoefficients, 1003
LaurentPolynomialByExtRep, 1018
LaurentPolynomialByExtRepNC, 1018
LClassOfHClass, 769
Lcm
  for (a ring and) a list of elements, 859
  for (a ring and) several elements, 859
LcmInt, 178
LcmOp, 859
LeadCoeffsIGS, 630
LeadingCoefficient, 998
LeadingCoefficientOfPolynomial, 1009
LeadingExponentOfPcElement, 626
LeadingMonomial, 998
LeadingMonomialOfPolynomial, 1008
LeadingTermOfPolynomial, 1009
left cosets, 503
LeftActingAlgebra, 943
LeftActingDomain, 864
LeftActingRingOfIdeal, 850
LeftAlgebraModule, 941
LeftAlgebraModuleByGenerators, 940
LeftDerivations, 954
LeftIdeal, 847
LeftIdealByGenerators, 849
LeftIdealNC, 848
LeftModuleByGenerators, 864

LeftModuleByHomomorphismToMatAlg, 944
LeftModuleGeneralMappingByImages, 908
LeftModuleHomomorphismByImages, 909
LeftModuleHomomorphismByImagesNC, 909
LeftModuleHomomorphismByMatrix, 909
LeftOne
  for a partial perm, 825
  for a transformation, 806
LeftQuotient, 409
LeftQuotient
  for words, 471
LeftShiftRowVector, 304
legacy, 1208
Legendre, 194
Length, 272
  for a associative word, 472
length
  of a word, 472
LengthsTom, 1047
LengthWPObj, 1293
LenstraBase, 891
LessThanFunction, 447
LessThanOrEqualFunction, 447
LetterRepAssocWord, 476
LevelsOfGenerators, 451
LeviMalcevDecomposition
  for Lie algebras, 933
LexicographicOrdering, 448
LGFirst, 640
LGLayers, 640
LGLength, 640
LGWeights, 640
library tables, 1065
LieAlgebra
  for an associative algebra, 953
  for field and generators, 953
LieAlgebraByStructureConstants, 952
LieBracket, 409
LieCenter, 955
LieCentralizer, 956
LieCentre, 955
LieCoboundaryOperator, 973
LieDerivedSeries, 957
LieDerivedSubalgebra, 956
LieFamily, 951
LieLowerCentralSeries, 958
LieNilRadical, 957
LieNormalizer, 956
LieObject, 951
LieSolvableRadical, 957
LieUpperCentralSeries, 958
LiftedInducedPcgs, 634
LiftedPcElement, 634
LinearAction, 641
LinearActionLayer, 642
LinearCharacters
  for a character table, 1075
  for a group, 1075
LinearCombination, 900
LinearCombinationPcgs, 626
LinearIndependentColumns, 340
LinearOperation, 641
LinearOperationLayer, 642
LinesOfStraightLineProgram, 478
List
  for a collection, 383
  for a list (and a function), 283
list
  sorted, 271
list and non-list
  difference, 262
  left quotient, 264
  mod, 264
  product, 263
  quotient, 264
list assignment
  operation, 246
list boundedness test
  operation, 246
list element
  access, 247
  assignment, 248
  operation, 246
list equal
  comparison, 257
list of available books, 29
list smaller
  comparison, 257
list unbind
  operation, 246
ListBlist, 294
ListN, 287
ListOfDigits, 176
ListPerm, 593

ListStabChain, 607
ListTransformation, 799
ListWithIdenticalEntries, 266
ListX, 287
LLL, 1137
LLL algorithm
  for Gram matrices, 342
  for vectors, 341
  for virtual characters, 1137
LLLReducedBasis, 341
LLLReducedGramMat, 342
LoadDynamicModule, 1204
LoadPackage, 1198
local, 65
Log, 235
Log10, 235
Log1p, 235
Log2, 235
logarithm
  discrete, 192
  of a root of unity, 222
LogFFE, 879
logical, 240
Logical conjunction, 242
Logical disjunction, 241
Logical negation, 242
logical operations, 241
LogInt, 174
LogMod, 192
LogModShanks, 192
LogPackageLoadingMessage, 1199
LogTo
  for a filename, 131
  for streams, 140
  stop logging, 131
LongestWeylWordPerm, 963
LookupDictionary, 369
loop
  for, 63
  read eval print, 77
  repeat, 62
  while, 61
loop over iterator, 64
loop over object, 64
loop over range, 63
loops
  leaving, 64
  restarting, 65
LowercaseString, 356
LowerCentralSeriesOfGroup, 527
LowIndexSubgroupsFpGroup, 671
LowIndexSubgroupsFpGroupIterator, 671
LQUO
  for a permutation and transformation, 796
  for a permutation or partial permutation and partial permutation, 828
Lucas, 213

m_N (irrational value), 226
Magma, 453
MagmaByGenerators, 454
MagmaByMultiplicationTable, 456
MagmaElement, 457
MagmaHomomorphismByFunctionNC, 429
MagmaRingModuloSpanOfZero, 985
MagmaWithInverses, 454
MagmaWithInversesByGenerators, 454
MagmaWithInversesByMultiplicationTable, 457
MagmaWithOne, 453
MagmaWithOneByGenerators, 454
MagmaWithOneByMultiplicationTable, 457
MagmaWithZeroAdjoined, 455
MakeConfluent, 487
MakeFloat, 238
MakeImmutable, 155
MakeReadOnlyGlobal, 52
MakeReadWriteGlobal, 52
map
  parametrized, 1169
MappedWord, 465
MappingByFunction
  by function (and inverse function) between two domains, 420
  by function and function that computes one preimage, 420
MappingGeneratorsImages, 564
MappingPermListList, 593
maps, 1158
MarksTom, 1046
MatAlgebra, 921
MatClassMultCoeffsCharTable, 1092
MathieuGroup, 722
MatLieAlgebra, 954

matrices
commutator, 312
Matrix, 777
matrix automorphisms, 1162
matrix spaces, 904
MatrixAlgebra, 921
MatrixAutomorphisms, 1111
MatrixByBlockMatrix, 333
MatrixLieAlgebra, 954
MatrixOfAction, 944
MatScalarProducts, 1129
MatTom, 1049
MaximalAbelianQuotient, 529
MaximalBlocks
for a group, an action domain, etc., 582
for an external set, 582
MaximalNormalSubgroups, 532
MaximalSubgroupClassReps, 532
MaximalSubgroups, 532
MaximalSubgroups
for groups with pcgs, 643
MaximalSubgroupsLattice, 534
MaximalSubgroupsTom, 1055
Maximum
for a list, 281
for various objects, 281
MaximumList, 282
meet strategy, 1303
MeetEquivalenceRelations, 443
MeetMaps, 1173
MeetOfPartialPerms, 817
MemoryUsage, 158
method, 1211
Mid, 236
MinimalElementCosetStabChain, 608
MinimalGeneratingSet, 539
MinimalGeneratingSet
for groups with pcgs, 643
MinimalNonmonomialGroup, 1196
MinimalNormalSubgroups, 532
MinimalPolynomial, 1000
over a field, 873
MinimalPolynomial
over a ring, 1000
MinimalStabChain, 604
MinimalSupergroupsLattice, 534
MinimalSupergroupsTom, 1055
MinimizedBombieriNorm, 1003
Minimum
for a list, 281
for various objects, 281
MinimumList, 282
MinusCharacter, 1180
mod
Integers, 185
Laurent polynomials, 990
lists, 264
rationals, 58
\mod, 408
for residue class rings, 184
for two pcgs, 632
mod, 57
arithmetic operators, 57
for character tables, 1073
residue class rings, 184
modular inverse, 58
modular remainder, 58
modular roots, 195
ModuleByRestriction, 946
ModuleOfExtension, 654
modulo, 57
arithmetic operators, 57
residue class rings, 184
ModuloPcgs, 631
MoebiusMu, 197
MoebiusTom, 1050
MolienSeries, 1146
MolienSeriesInfo, 1146
MolienSeriesWithGivenDenominator, 1148
Monoid
for a list, 761
for various generators, 761
MonoidByGenerators, 761
MonoidByMultiplicationTable, 762
MonoidOfRewritingSystem, 787
MonomialComparisonFunction, 1009
MonomialExtGrlexLess, 1012
MonomialExtrepComparisonFun, 1009
MonomialGrevlexOrdering, 1010
MonomialGrlexOrdering, 1010
MonomialLexOrdering, 1009
MonomialTotalDegreeLess, 1209
monomorphisms
find all, 562

MorClassLoop, 562
MostFrequentGeneratorFpGroup, 668
MovedPoints
for a list or collection of permutations, 591
for a partial perm, 821
for a partial perm coll, 821
for a permutation, 591
for a transformation, 800
for a transformation coll, 800
MTX, 1028
MTX.BasesCompositionSeries, 1031
MTX.BasesMaximalSubmodules, 1031
MTX.BasesMinimalSubmodules, 1031
MTX.BasesMinimalSupermodules, 1031
MTX.BasesSubmodules, 1030
MTX.BasisInOrbit, 1035
MTX.BasisModuleEndomorphisms, 1033
MTX.BasisModuleHomomorphisms, 1033
MTX.BasisRadical, 1031
MTX.BasisSocle, 1031
MTX.CollectedFactors, 1031
MTX.CompositionFactors, 1031
MTX.DegreeSplittingField, 1029
MTX.Dimension, 1029
MTX.Distinguish, 1034
MTX.Field, 1029
MTX.Generators, 1028
MTX.HomogeneousComponents, 1030
MTX.Homomorphism, 1034
MTX.Homomorphisms, 1034
MTX.Indecomposition, 1030
MTX.InducedAction, 1032
MTX.InducedActionFactorMatrix, 1032
MTX.InducedActionFactorModule, 1032
MTX.InducedActionMatrix, 1032
MTX.InducedActionMatrixNB, 1032
MTX.InducedActionSubmodule, 1032
MTX.InducedActionSubmoduleNB, 1032
MTX.InvariantBilinearForm, 1034
MTX.InvariantQuadraticForm, 1035
MTX.InvariantSesquilinearForm, 1034
MTX.IsAbsolutelyIrreducible, 1029
MTX.IsEquivalent, 1033
MTX.IsIndecomposable, 1029
MTX.IsIrreducible, 1029
MTX.IsomorphismIrred, 1033
MTX.IsomorphismModules, 1033
MTX.ModuleAutomorphisms, 1033
MTX.NormedBasisAndBaseChange, 1032
MTX.OrthogonalSign, 1035
MTX.ProperSubmoduleBasis, 1030
MTX.SubGModule, 1030
MTX.SubmoduleGModule, 1030
multiplication, 57
matrices, 311
matrix and matrix list, 312
matrix and scalar, 311
matrix and vector, 311
operation, 408
scalar and matrix, 311
scalar and matrix list, 312
scalar and vector, 300
vector and matrix, 311
vector and matrix list, 312
vector and scalar, 300
vectors, 301
MultiplicationTable
for a list of elements, 457
for a magma, 457
multiplicative order of an integer, 192
MultiplicativeNeutralElement, 460
MultiplicativeZero, 460
MultiplicativeZeroOp, 405
multiplicity
of constituents of a group character, 1129
Multiplier, 544
multisets, 274
MultRowVector, 304
Murnaghan components, 1144, 1145
MutableBasis, 903
MutableBasisOfClosureUnderAction, 929
MutableBasisOfIdealInNonassociative-
Algebra, 930
MutableBasisOfNonassociativeAlgebra,
930
MutableIdentityMat, 1209
MutableNullMat, 1209

Name, 157
NameFunction, 70
NameRNam, 378
NamesFilter, 161
NamesGVars, 53
NamesLocalVariablesFunction, 71

NamesOfComponents, 1224
NamesOfFusionSources, 1166
namespace, 49, 51
NamesSystemGVars, 53
NamesUserGVars, 54
NaturalCharacter
  for a group, 1126
  for a homomorphism, 1126
NaturalHomomorphismByGenerators, 429
NaturalHomomorphismByIdeal, 861
  for an algebra and an ideal, 937
NaturalHomomorphismByNormalSubgroup, 529
NaturalHomomorphismByNormalSubgroupNC, 529
NaturalHomomorphismBySubAlgebraModule, 947
NaturalHomomorphismBySubspace, 910
NaturalHomomorphismOfLieAlgebraFromNilpotentGroup, 967
NaturalLeqPartialPerm, 830
NaturalPartialOrder, 835
NearAdditiveGroup, 840
NearAdditiveGroupByGenerators, 841
NearAdditiveMagma, 840
NearAdditiveMagmaByGenerators, 840
NearAdditiveMagmaWithZero, 840
NearAdditiveMagmaWithZeroByGenerators, 840
NearlyCharacterTablesFamily, 1069
negative number, 57
NegativeRoots, 961
NegativeRootVectors, 961
NestingDepthA, 259
NestingDepthM, 260
NewAttribute, 1218
NewAttribute
  example, 1242
NewCategory, 1217
NewConstructor, 1220
NewDictionary, 368
NewFamily, 1221
NewFilter, 1219
NewFloat, 238
NewInfoClass, 103
newline, 47
NewmanInfinityCriterion, 683
NewOperation, 1220
NewProperty, 1219
NewRepresentation, 1218
NewRepresentation
  example, 1243
NewType, 1222
NextIterator, 393
NextPrimeInt, 181
NF, 887
NiceAlgebraMonomorphism, 938
NiceBasis, 912
NiceBasisFiltersInfo, 914
NiceFreeLeftModule, 912
NiceFreeLeftModuleInfo, 912
NiceMonomorphism, 555
NiceMonomorphismAutomGroup, 560
NiceObject, 555
NiceVector, 912
NilpotencyClassOfGroup, 517
NilpotentQuotientOfFpLieAlgebra, 970
NK, 226
NOAUTO, 1198
NonabelianExteriorSquare, 545
NonnegativeIntegers, 172
NonnegIntScalarProducts, 1176
NonNilpotentElement, 968
Norm, 874
  for a class function, 1129
Norm
  of character, 1129
NormalBase, 875
NormalClosure, 510
NormalFormIntMat, 338
NormalIntersection, 510
NormalizedElementOfMagmaRingModuloRelations, 985
NormalizedWhitespace, 357
Normalizer
  for a group and a group element, 509
  for two groups, 509
normalizer, 509
NormalizerInGLnZ, 620
NormalizerInGLnZBravaisGroup, 621
NormalizersTom, 1052
NormalizerTom, 1052
NormalizeWhitespace, 357
NormalSeriesByPcgs, 638

NormalSubgroupClasses, 1114
NormalSubgroupClassesInfo, 1113
NormalSubgroups, 532
NormedRowVector, 301
NormedRowVectors, 908
NormedVectors, 1209
not, 242
NrArrangements, 204
NrBasisVectors, 903
NrCombinations, 204
NrComponentsOfPartialPerm, 824
NrComponentsOfTransformation, 804
NrConjugacyClasses, 508
    for a character table, 1076
NrConjugacyClassesGL, 729
NrConjugacyClassesGU, 729
NrConjugacyClassesPGL, 729
NrConjugacyClassesPGU, 730
NrConjugacyClassesPSL, 730
NrConjugacyClassesPSU, 730
NrConjugacyClassesSL, 729
NrConjugacyClassesSLIsogeneous, 730
NrConjugacyClassesSU, 729
NrConjugacyClassesSUIsogeneous, 730
NrDerangements, 207
NrFixedPoints
    for a partial perm, 821
    for a partial perm coll, 821
NrInputsOfStraightLineProgram, 478
NrMovedPoints
    for a list or collection of permutations, 592
    for a partial perm, 821
    for a partial perm coll, 821
    for a permutation, 592
    for a transformation, 800
    for a transformation coll, 800
NrOrderedPartitions, 210
NrPartitions, 209
NrPartitionsSet, 208
NrPartitionTuples, 212
NrPermutationsList, 207
NrPolyhedralSubgroups, 1090
NrPrimitiveGroups, 744
NrRestrictedPartitions, 211
NrSubsTom, 1047
NrTransitiveGroups, 732
NrTuples, 206
NrUnorderedTuples, 205
NthRootsInGroup, 509
NullAlgebra, 922
NullMat, 315
NullspaceIntMat, 334
NullspaceMat, 319
NullspaceMatDestructive, 319
NullspaceModQ, 332
Number, 284
number
    Bell, 201
    binomial, 200
    Stirling, of the first kind, 202
    Stirling, of the second kind, 202
number field, 888
number fields
    Galois group, 891
NumberArgumentsFunction, 70
NumberFFVector, 303
NumberIrreducibleSolvableGroups, 746
NumberPerfectGroups, 739
NumberPerfectLibraryGroups, 739
NumberSmallGroups, 735
NumbersString, 359
NumberSyllables, 474
NumberTransformation, 793
numerator
    of a rational, 216
NumeratorOfModuloPcgs, 632
NumeratorOfRationalFunction, 992
NumeratorRat, 216

ObjByExtRep, 1232
    for creating a UEALattice element, 976
Objectify, 1222
ObjectifyWithAttributes, 1222
obsolete, 1208
OCOneCocycles, 543
octal character codes, 350
OctaveAlgebra, 921
od, 63
OldGeneratorsOfPresentation, 704
Omega, 516
    construct an orthogonal group, 727
OmniGraffle, 533
ONanScottType, 598
OnBreak, 84

OnBreakMessage, 86
One, 404
  for a partial perm, 825
one cohomology, 541
OneAttr, 403
OneCoboundaries, 541
OneCocycles
  for a group and a pcgs, 541
  for generators and a group, 541
  for generators and a pcgs, 541
  for two groups, 541
OneFactorBound, 1003
OneImmutable, 403
OneIrreducibleSolvableGroup, 746
One*Library*Group, 731
OneMutable, 404
OneOfPcgs, 625
OneOp, 404
OnePrimitiveGroup, 731
OneSameMutability, 404
OneSM, 404
OneSmallGroup, 735
OneTransitiveGroup, 731
OnIndeterminates
  as a permutation action, 570
OnLeftInverse, 568
OnLines, 570
OnLines
  example, 724
OnPairs, 568
OnPoints, 567
OnQuit, 122
OnRight, 567
OnSets, 568
OnSetsDisjointSets, 569
OnSetsSets, 568
OnSetsTuples, 569
OnSubspacesByCanonicalBasis, 571
OnSubspacesByCanonicalBasis-
    Concatenations, 571
OnTuples, 568
OnTuplesSets, 569
OnTuplesTuples, 569
PCore
  see PCore, 510
Operation, 1208
operation, 1211
OperationAlgebraHomomorphism
  action on a free left module, 937
  action w.r.t. a basis of the module, 937
OperationHomomorphism, 1208
operations
  for booleans, 241
Operations for algebraic elements, 1021
operators, 49
  arithmetic, 57
  associativity, 58
  for cyclotomics, 224
  for lists, 258
  precedence, 57
options, 33
  command line, filenames, 37
  command line, internal, 38
  under UNIX, 33
or, 241
Orbit, 571
OrbitFusions, 1169
OrbitishFO, 1290
OrbitLength, 573
OrbitLengths
  for a group, a set of seeds, etc., 573
  for an external set, 573
OrbitLengthsDomain
  for a group and a set of seeds, 573
  of an external set, 573
OrbitPerms, 596
OrbitPowerMaps, 1162
Orbits
  attribute, 572
  operation, 572
Orbits
  as attributes for external sets, 1288
OrbitsDomain
  for a group and an action domain, 572
  of an external set, 572
OrbitsishOperation, 1289
OrbitsPerms, 596
OrbitStabChain, 607
OrbitStabilizer, 573
OrbitStabilizerAlgorithm, 574
Order, 407
  for a class function, 1124
order
  of a group, 491

of a list, collection or domain, 387
of the prime residue group, 191
ordered partitions
internal representation, 1297
OrderedPartitions, 209
ordering
booleans, 241
of records, 376
OrderingByLessThanFunctionNC, 445
OrderingByLessThanOrEqualFunctionNC, 445
OrderingOfRewritingSystem, 486
OrderingOnGenerators, 448
OrderingsFamily, 445
OrderMod, 192
OrderOfRewritingSystem, 486
OrdersClassRepresentatives, 1077
OrdersTom, 1047
Ordinal, 361
ordinary character, 1128
OrdinaryCharacterTable
for a character table, 1075
for a group, 1075
OrthogonalComponents, 1144
OrthogonalEmbeddings, 343
OrthogonalEmbeddingsSpecialDimension, 1139
output
suppressing, 77
OutputLogTo
for a filename, 131
for streams, 140
stop logging output, 131
OutputTextFile, 141
OutputTextNone, 145
OutputTextString, 143
OutputTextUser, 142
Overlaps, 236
overload, 1215

p-group, 522
package, 1197
PACKAGE_DEBUG, 1199
PACKAGE_ERROR, 1199
PACKAGE_INFO, 1199
PackageVariablesInfo, 1205
PACKAGE_WARNING, 1199
PadicCoefficients, 341
PadicExtensionNumberFamily, 1025
PadicNumber
for a p-adic extension family and a list, 1025
for a p-adic extension family and a rational, 1025
for a pure p-adic numbers family and a list, 1025
for pure padics, 1023
PadicValuation, 856
Pager, 31
PageSource, 72
Parametrized, 1171
parametrized maps, 1158
Parent, 401
ParentPcgs, 629
ParseRelators, 660
partial order, 439
PartialFactorization, 182
PartialOrderByOrderingFunction, 441
PartialOrderOfHasseDiagram, 440
PartialPerm
for a dense image, 814
for a domain and image, 814
PartialPermFamily, 814
PartialPermOp, 815
PartialPermOpNC, 815
Partitions, 208
partitions
improper, of an integer, 209
ordered, of an integer, 209
restricted, of an integer, 210
PartitionsGreatestEQ, 210
PartitionsGreatestLE, 210
PartitionsSet, 208
PartitionTuples, 212
PcElementByExponents, 626
PcElementByExponentsNC, 626
PCentralLieAlgebra, 966
PCentralNormalSeriesByPcgsPGroup, 636
PCentralSeries, 527
PcGroupCode, 657
PcGroupFpGroup, 648
PcGroupWithPcgs, 651
Pcgs, 623
PcgsByPcSequence, 624
PcgsByPcSequenceNC, 624

PcgsCentralSeries, 635
PcgsChiefSeries, 637
PcgsElementaryAbelianSeries
for a group, 634
for a list of normal subgroups, 634
Pcgs_OrbitStabilizer, 643
PcgsPCentralSeriesPGroup, 636
PClassPGroup, 522
PCore, 510
PcSeries, 625
perfect groups, 737
PerfectGroup
for a pair [order, index], 738
for group order (and index), 738
PerfectIdentification, 738
PerfectResiduum, 513
Perform, 280
Permanent, 213
PermBounds, 1154
PermCharInfo, 1149
PermCharInfoRelative, 1150
PermChars, 1151
PermCharsTom
from a character table, 1059
via fusion map, 1059
PermComb, 1154
PermLeftQuoPartialPerm, 829
PermLeftQuoPartialPermNC, 829
PermLeftQuoTransformation, 797
PermLeftQuoTransformationNC, 797
PermList, 593
PermListList, 281
Permutation
for a group, an action domain, etc., 578
for an external set, 578
permutation character, 1182
permutation characters
possible, 1148
PermutationCharacter
for a group, an action domain, and a function, 1127
for two groups, 1127
PermutationCycle, 579
PermutationGModule, 1028
PermutationMat, 316
PermutationOfImage, 796
PermutationsFamily, 590
PermutationsList, 206
PermutationTom, 1045
Permuted, 283
as a permutation action, 570
for a class function, 1123
PGL, 728
PGU, 728
Phi, 191
point stabilizer, 573
PolynomialByExtRep, 1017
PolynomialByExtRepNC, 1017
PolynomialCoefficientsOfPolynomial, 997
PolynomialDivisionAlgorithm, 1011
PolynomialModP, 1001
PolynomialReducedRemainder, 1011
PolynomialReduction, 1011
PolynomialRing
for a ring and a list of indeterminate numbers, 1005
for a ring and a list of indeterminates, 1005
for a ring and a list of names (and an exclusion list), 1005
for a ring and a rank (and an exclusion list), 1005
POmega, 729
PopOptions, 122
Position, 266
PositionBound, 269
PositionCanonical, 267
PositionFirstComponent, 270
PositionNonZero, 270
PositionNot, 269
PositionNthOccurrence, 267
PositionProperty, 268
Positions, 266
PositionSet, 268
PositionsOp, 266
PositionSorted, 267
PositionSortedOp, 267
PositionsProperty, 269
PositionStream, 138
PositionSublist, 270
PositionWord, 472
positive number, 57
PositiveIntegers, 172
PositiveRoots, 960
PositiveRootVectors, 961

possible permutation characters, 1148
PossibleClassFusions, 1167
PossibleFusionsCharTableTom, 1058
PossiblePowerMaps, 1160
power, 57
  matrix, 312
  meaning for class functions, 1122
  of words, 471
power set, 203
PowerMap, 1159
PowerMapByComposition, 1162
PowerMapOp, 1159
PowerMapsAllowedBySymmetrizations, 1181
PowerMod, 860
PowerModCoeffs, 308
PowerModInt, 179
PowerPartition, 212
PowerSubalgebraSeries, 927
PQuotient, 678
precedence, 57
precedence test
  for permutations, 590
PrecisionFloat, 236
PrefrattiniSubgroup, 512
PrefrattiniSubgroup
  for groups with pcgs, 643
PreImage
  set of preimages of a collection under a general mapping, 427
  set of preimages of the range of a general mapping, 427
  unique preimage of an element under a general mapping, 427
PreImageElm, 426
PreImagePartialPerm, 829
PreImages
  set of preimages of a collection under a general mapping, 427
  set of preimages of an elm under a general mapping, 427
  set of preimages of the range of a general mapping, 427
PreImagesElm, 426
PreImagesOfTransformation, 798
PreImagesRange, 426
PreImagesRepresentative, 426
PreImagesSet, 427
preorder, 439
PresentationFpGroup, 684
PresentationNormalClosure, 691
PresentationNormalClosureRrs, 691
PresentationSubgroup, 687
PresentationSubgroupMtc, 689
PresentationSubgroupRrs
  for a group and a coset table (and a string), 687
  for two groups (and a string), 687
PresentationViaCosetTable, 685
previous result, 77
PrevPrimeInt, 181
PrimalityProof, 180
primary subgroup generators, 706
PrimaryGeneratorWords, 689
prime residue group, 190
  exponent, 191
  generator, 193
  order, 191
PrimeBlocks, 1085
PrimeBlocksOp, 1085
PrimeDivisors, 182
PrimeField, 871
PrimePGroup, 522
PrimePowersInt, 184
PrimeResidues, 190
Primes, 179
primitive, 582
primitive root modulo an integer, 193
PrimitiveElement, 871
PrimitiveGroup, 744
PrimitiveGroupsIterator, 744
PrimitiveIdentification, 745
PrimitiveIndexIrreducibleSolvableGroup, 747
PRIMITIVE_INDICES_MAGMA, 745
PrimitivePolynomial, 1001
PrimitiveRoot, 882
PrimitiveRootMod, 193
Print, 81
PrintAmbiguity, 1176
PrintArray, 317
PrintCharacterTable, 1095
PrintCSV, 147
PrintFactorsInt, 183
PrintFormattingStatus, 140

PrintObj, 82
for a character table, 1092
for a ffe, 884
for a string, 348
for a table of marks, 1043
for class functions, 1124
PrintString, 354
PrintTo, 130
for streams, 139
ProbabilityShapes, 1002
procedure call, 60
procedure call with arguments, 60
Process, 148
PROD_GF2MAT_GF2MAT_ADVANCED, 332
PROD_GF2MAT_GF2MAT_SIMPLE, 332
Product, 286
product
of words, 471
rational functions, 990
ProductCoeffs, 307
ProductOfStraightLinePrograms, 482
ProductSpace, 927
ProductX, 289
ProfileFunctions, 108
ProfileGlobalFunctions, 107
ProfileMethods, 108
ProfileOperations, 107
ProfileOperationsAndMethods, 108
ProjectedInducedPcgs, 633
ProjectedPcElement, 633
Projection
for a domain, 422
for a domain and a positive integer, 422
for group products, 718
for two domains, 422
Projection
example for direct products, 713
example for semidirect products, 714
example for subdirect products, 715
example for wreath products, 716
ProjectionMap, 1171
projections
find all, 561
ProjectiveActionHomomorphismMatrix-
Group, 615
ProjectiveActionOnFullSpace, 615
ProjectiveGeneralLinearGroup, 728
ProjectiveGeneralUnitaryGroup, 728
ProjectiveOmega, 729
ProjectiveOrder, 331
ProjectiveSpecialLinearGroup, 728
ProjectiveSpecialUnitaryGroup, 728
ProjectiveSymplecticGroup, 729
prompt, 77
partial, 77
PRump, 514
PseudoRandom, 391
for finitely presented groups, 664
PSL, 728
PSP, 729
PSp, 729
PSU, 728
PthPowerImage
for basis and element, 965
for element, 965
for element and integer, 966
PthPowerImages, 965
PurePadicNumberFamily, 1023
PushOptions, 121

Quadratic, 229
quadratic residue, 193, 194
QuaternionAlgebra, 920
QuaternionGroup, 721
QUIET, 1209
QUIT, 90
QUIT
emergency quit, 90
quit
in emergency, 90
QUITTING, 90
QuoInt, 176
Quotient, 846
quotient
for finitely presented groups, 660
matrices, 312
matrix and matrix list, 312
matrix and scalar, 312
of free monoid, 785
of free semigroup, 782
of words, 471
rational functions, 990
scalar and matrix, 312
scalar and matrix list, 312

vector and matrix, 312
QuotientFromSCTable, 920
QuotientMod, 859
QuotientPolynomialsExtRep, 1018
QuotientRemainder, 857
QuotientSemigroupCongruence, 768
QuotientSemigroupHomomorphism, 768
QuotientSemigroupPreimage, 768
QuotRemLaurpols, 996

r_N (irrational value), 225
RadicalGroup, 513
RadicalOfAlgebra, 932
Random
for a list, 187
for a list or collection, 391
for a range of integers, 187
for integers, 176
for lower and upper bound, 391
for rationals, 217
random element
of a list or collection, 391
random seed, 392
RandomBinaryRelationOnPoints, 440
RandomInvertibleMat, 318
RandomIsomorphismTest, 657
RandomList, 392
RandomMat, 317
RandomPartialPerm
for a positive integer, 818
for a set of positive integers, 818
for domain and image, 818
RandomPrimitivePolynomial, 884
RandomSource, 189
RandomTransformation, 793
RandomUnimodularMat, 318
Range
of a general mapping, 423
range, 289
RankAction
for a group, an action domain, etc., 581
for an external set, 581
RankFilter, 161
RankMat, 318
RankOfPartialPerm, 819
RankOfPartialPermCollection, 819
RankOfPartialPermSemigroup, 834
RankOfTransformation
for a transformation and a list, 799
for a transformation and a positive integer, 799
RankPGroup, 523
Rat, 217
for floats, 238
for strings, 360
RationalClass, 508
RationalClasses, 508
RationalFunctionByExtRep, 1017
RationalFunctionByExtRepNC, 1017
RationalFunctionByExtRepWithCancellation, 1019
RationalFunctionsFamily, 1014
RationalizedMat, 230
Rationals, 215
RClassOfHClass, 769
Read, 129
for streams, 136
read eval print loop, 77
ReadAll, 137
ReadAllLine, 145
ReadAsFunction, 130
for streams, 136
ReadByte, 136
ReadCommandLineHistory, 94
ReadCSV, 146
ReadLine, 137
ReadlineInitLine, 93
ReadPackage, 1201
ReadPkg, 1209
ReadTest, 113
for streams, 136
RealClasses, 1081
RealizableBrauerCharacters, 1157
RealPart, 228
RecNames, 373
record
component access, 373
component assignment, 374
component variable, 373
component variable assignment, 374
record assignment
operation, 378
record boundness test
operation, 378

record component
operation, 378
record unbind
operation, 378
recursion, 65
RedispatchOnCondition, 1214
redisplay a help section, 29
redisplay with next help viewer, 29
ReduceCoeffs, 307
ReduceCoeffsMod, 307
ReducedAdditiveInverse, 487
ReducedCharacters, 1137
ReducedClassFunctions, 1136
ReducedComm, 487
ReducedConfluentRewritingSystem, 785
ReducedConjugate, 487
ReducedDifference, 487
ReducedForm, 486
ReducedGroebnerBasis
for a list and a monomial ordering, 1013
for an ideal and a monomial ordering, 1013
ReducedInverse, 487
ReducedLeftQuotient, 487
ReducedOne, 487
ReducedPcElement, 627
ReducedPower, 487
ReducedProduct, 487
ReducedQuotient, 487
ReducedScalarProduct, 487
ReducedSum, 487
ReducedZero, 487
ReduceRules, 486
ReduceStabChain, 609
Ree, 723
ReeGroup, 723
ReesCongruenceOfSemigroupIdeal, 767
ReesMatrixSemigroup, 772
ReesMatrixSemigroupElement, 774
ReesMatrixSubsemigroup, 773
ReesZeroMatrixSemigroup, 772
ReesZeroMatrixSemigroupElement, 774
ReesZeroMatrixSubsemigroup, 773
RefinedPcGroup, 650
ReflectionMat, 317
reflexive relation, 438
ReflexiveClosureBinaryRelation, 440
regular, 581
regular action, 576
RegularActionHomomorphism, 577
RegularModule, 1100
relations, 419
RelationsOfFpSemigroup, 784
RelativeBasis, 899
RelativeBasisNC, 899
RelativeDiameter, 236
relatively prime, 58
RelativeOrderOfPcElement, 625
RelativeOrders, 624
RelativeOrders
of a pcgs, 624
RelatorsOfFpGroup, 662
remainder
operation, 408
remainder of a quotient, 177
RemInt, 177
Remove, 251
remove
an element from a set, 276
RemoveCharacters, 357
RemoveFile, 132
RemoveOuterCoeffs, 305
RemoveRelator, 695
RemoveSet, 276
RemoveStabChain, 609
repeat loop, 62
ReplacedString, 357
representation
as a sum of two squares, 198
RepresentationsOfObject, 165
Representative, 388
representative
of a list or collection, 388
RepresentativeAction, 575
RepresentativeLinearOperation, 939
RepresentativeOperation, 1208
RepresentativesContainedRightCosets, 505
RepresentativesFusions, 1169
RepresentativeSmallest, 388
RepresentativesMinimalBlocks
for a group, an action domain, etc., 583
for an external set, 583
RepresentativesPerfectSubgroups, 535
RepresentativesPowerMaps, 1162

RepresentativesSimpleSubgroups, 535
RepresentativeTom, 1058
RepresentativeTomByGenerators, 1058
RepresentativeTomByGeneratorsNC, 1058
RequirePackage, 1209
Reread, 132
REREADING, 132
RereadPackage, 1201
RereadPkg, 1209
Reset, 187
ResetFilterObj, 1220
ResetOptionsStack, 122
residue
    quadratic, 193, 194
RespectsAddition, 430
RespectsAdditiveInverses, 430
RespectsInverses, 429
RespectsMultiplication, 429
RespectsOne, 429
RespectsScalarMultiplication, 432
RespectsZero, 431
RestrictedClassFunction, 1134
RestrictedClassFunctions, 1134
RestrictedLieAlgebraByStructure-
        Constants, 953
RestrictedMapping, 422
RestrictedPartialPerm, 816
RestrictedPartitions, 210
RestrictedPerm, 593
RestrictedPermNC, 593
RestrictedTransformation, 795
RestrictedTransformationNC, 795
RestrictOutputsOfSLP, 481
Resultant, 999
ResultOfStraightLineProgram, 479
return, 84
    no value, 68
    with value, 69
return from break loop, 84
ReturnFail, 74
ReturnFalse, 74
ReturnTrue, 73
Reversed, 279
ReverseNaturalPartialOrder, 835
RewindStream, 138
RewriteWord, 671
right cosets, 502
RightActingAlgebra, 943
RightActingRingOfIdeal, 850
RightAlgebraModule, 941
RightAlgebraModuleByGenerators, 940
RightCoset, 502
RightCosets, 502
RightCosetsNC, 502
RightDerivations, 954
RightIdeal, 847
RightIdealByGenerators, 849
RightIdealNC, 848
RightModuleByHomomorphismToMatAlg, 944
RightOne
    for a partial perm, 825
    for a transformation, 806
RightShiftRowVector, 304
RightTransversal, 504
Ring
    for a collection, 844
    for ring elements, 844
RingByGenerators, 845
RingGeneralMappingByImages, 860
RingHomomorphismByImages, 861
RingHomomorphismByImagesNC, 861
RingWithOne
    for a collection, 851
    for ring elements, 851
RingWithOneByGenerators, 851
RNamObj
    for a positive integer, 378
    for a string, 378
root
    of 1 modulo an integer, 195
    of an integer, 175
    of an integer modulo another, 194
    of an integer, smallest, 175
RootInt, 175
RootMod, 194
RootOfDefiningPolynomial, 871
roots of unity, 218
RootsMod, 195
RootsOfPolynomial, 995
RootsOfUPol, 996
RootsUnityMod, 195
RootSystem, 960
Round, 236
RoundCyc, 221

row spaces, 904
Rows, 777
RREF, 318
Rules, 485
Runtime, 106
Runtimes, 106

s_N (irrational value), 225
SameBlock, 1086
save, 42
SaveCommandLineHistory, 94
SaveOnExitFile, 90
SaveWorkspace, 42
saving on exit, 90
ScalarProduct
    for characters, 1129
Schreier, 687
Schreier-Sims
    random, 600
Schur multiplier, 544
SchurCover, 544
SchurCoverOfSymmetricGroup, 546
scope, 50
Sec, 235
Sech, 235
SecHMSM, 364
secondary subgroup generators, 706
SecondsDMYhms, 364
SeekPositionStream, 138
SemidirectProduct
    for a group of automorphisms and a group, 713
    for acting group, action, and a group, 713
SemiEchelonBasis, 906
SemiEchelonBasisNC, 906
SemiEchelonMat, 323
SemiEchelonMatDestructive, 323
SemiEchelonMats, 324
SemiEchelonMatsDestructive, 324
SemiEchelonMatTransformation, 324
Semigroup
    for a list, 757
    for various generators, 757
semigroup, 757
SemigroupByGenerators, 758
SemigroupByMultiplicationTable, 760
SemigroupIdealByGenerators, 767
SemigroupOfRewritingSystem, 787
semiregular, 581
SemiSimpleType, 959
sequence
    Bernoulli, 201
    Fibonacci, 212
    Lucas, 213
Set, 384
set difference
    of collections, 390
set stabilizer, 573
SetAssertionLevel, 105
SetCommutator, 649
SetConjugate, 649
SetCrystGroupDefaultAction, 621
SetCyclotomicsLimit, 231
SetDefaultInfoOutput, 104
SetElmWPObj, 1293
SetEntrySCTable, 919
SetFilterObj, 1219
SetFloats, 238
SetGasmanMessageStatus, 119
SetHelpViewer, 30
SetIndeterminateName, 989
SetInfoHandler, 104
SetInfoLevel, 103
SetInfoOutput, 104
SetName, 157
SetNameObject, 83
SetPackagePath, 1199
SetParent, 401
SetPower, 649
SetPrintFormattingStatus, 140
SetRecursionTrapInterval, 117
SetReducedMultiplication, 662
Sets, 244
sets, 274
Setter, 166
setter, 166
SetUserPreference, 39
SetX, 288
ShallowCopy, 156
ShallowCopy
    for lists, 254
ShiftedCoeffs, 308
ShiftedPadicNumber, 1024
short vectors spanning a lattice, 341, 1137

ShortestVectors, 343
ShortLexLeqPartialPerm, 830
ShortLexOrdering, 449
ShowAdditionTable, 843
ShowArgument, 100
ShowArguments, 99
ShowDetails, 100
ShowGcd, 858
ShowImpliedFilters, 161
ShowMethods, 100
ShowMultiplicationTable, 843
ShowOtherMethods, 100
ShowPackageVariables, 1205
ShowUserPreferences, 39
ShrinkAllocationPlist, 257
ShrinkAllocationString, 352
ShrinkRowVector, 304
Shuffle, 279
SiftedPcElement, 626
SiftedPermutation, 607
SiftedVector, 908
Sigma, 196
SigmaL, 727
sign
  of an integer, 174
SignFloat, 236
SignInt, 174
SignPartition, 211
SignPerm, 592
SimpleGroup, 520
SimpleGroupsIterator, 521
SimpleLieAlgebra, 955
SimpleSystem, 961
SimplifiedFpGroup, 687
SimplifyPresentation, 696
SimsNo, 745
SimultaneousEigenvalues, 331
Sin, 235
SinCos, 235
SingleCollector, 649
singlequote character, 350
singlequotes, 347
Sinh, 235
SIntChar, 360
Size, 387
  for a character table, 1076
Size
  for groups with pcgs, 643
size
  of a list or collection, 387
SizeBlist, 294
SizeConsiderFunction, 538
SizeNumbersPerfectGroups, 739
SizeOfFieldOfDefinition, 1156
SizesCentralisers, 1078
SizesCentralizers, 1078
SizesConjugacyClasses, 1078
SizeScreen, 97
SizesPerfectGroups, 738
SizeStabChain, 606
SL
  for dimension and a field size, 725
  for dimension and a ring, 724
SlotUsagePattern, 482
smaller
  associative words, 471
  elements of finitely presented groups, 661
  for pcwords, 647
  for transformations, 797
  nonassociative words, 464
  rational functions, 991
smaller or equal, 56
smaller test, 56
SmallerDegreePermutation-
    Representation, 596
SmallestGeneratorPerm, 591
SmallestIdempotentPower
  for a partial perm, 824
  for a transformation, 804
SmallestImageOfMovedPoint
  for a partial permutation, 823
  for a partial permutation coll, 823
  for a transformation, 801
  for a transformation coll, 801
SmallestMovedPoint
  for a list or collection of permutations, 591
  for a partial perm, 822
  for a partial perm coll, 822
  for a permutation, 591
  for a transformation, 800
  for a transformation coll, 800
SmallestRootInt, 175
SmallGeneratingSet, 540
SmallGroup

for a pair [order, index], 734
for group order and index, 734
SmallGroupsInformation, 736
SmallSimpleGroup, 521
Smith normal form, 1209
SmithNormalFormIntegerMat, 337
SmithNormalFormIntegerMatTransforms, 337
SMTX.AbsoluteIrreducibilityTest, 1036
SMTX.AlgEl, 1037
SMTX.AlgElCharPol, 1037
SMTX.AlgElCharPolFac, 1037
SMTX.AlgElMat, 1037
SMTX.AlgElNullspaceDimension, 1038
SMTX.AlgElNullspaceVec, 1037
SMTX.CentMat, 1038
SMTX.CentMatMinPoly, 1038
SMTX.CompleteBasis, 1037
SMTX.Getter, 1036
SMTX.GoodElementGModule, 1035
SMTX.IrreducibilityTest, 1036
SMTX.MatrixSum, 1036
SMTX.MinimalSubGModule, 1036
SMTX.MinimalSubGModules, 1036
SMTX.RandomIrreducibleSubGModule, 1035
SMTX.Setter, 1036
SMTX.SortHomGModule, 1035
SMTX.Subbasis, 1037
SO, 726
Socle, 513
SocleTypePrimitiveGroup, 599
SolutionIntMat, 334
SolutionMat, 320
SolutionMatDestructive, 320
SolutionNullspaceIntMat, 334
SolvableQuotient
for a f.p. group and a list of primes, 679
for a f.p. group and a list of tuples, 679
for a f.p. group and a size, 679
Sort, 272
SortBy, 272
Sorted Lists as Collections, 380
SortedCharacters, 1109
SortedCharacterTable
relative to the table of a factor group, 1109
w.r.t. a normal subgroup, 1109
w.r.t. a series of normal subgroups, 1109
SortedList, 383
SortedSparseActionHomomorphism, 576
SortedTom, 1044
Sortex, 273
SortingPerm, 274
SortParallel, 273
Source, 424
SourceOfIsoclinicTable, 1107
SP
for dimension and a ring, 726
for dimension and field size, 726
Sp
for dimension and a ring, 726
for dimension and field size, 726
space, 47
SparseActionHomomorphism, 576
SparseCartanMatrix, 962
SparseHashTable, 371
SparseIntKey, 370
special character sequences, 349
SpecialLinearGroup
for dimension and a field size, 724
for dimension and a ring, 724
SpecialOrthogonalGroup, 726
SpecialPcgs
for a group, 640
for a pcgs, 640
SpecialSemilinearGroup, 727
SpecialUnitaryGroup, 725
SplitCharacters, 1102
SplitExtension, 654
SplitExtensions, 656
SplitString, 356
SplittingField, 994
Spreadsheet, 146
SQ
synonym of SolvableQuotient, 679
Sqrt, 409
Square, 236
square root
of an integer, 175
SquareRoots, 461
SSortedList, 384
StabChain
for a group (and a record), 602
for a group and a base, 602
StabChainBaseStrongGenerators, 604

StabChainImmutable, 603
StabChainMutable
for a group, 603
for a homomorphism, 603
StabChainOp, 603
StabChainOptions, 604
Stabilizer, 573
StabilizerOfExternalSet, 586
StabilizerPcgs, 643
Stack trace, 87
StandardAssociate, 854
StandardAssociateUnit, 855
StandardizeTable, 669
StandardWreathProduct, 716
StarCyc, 228
StartlineFunc, 71
START_TEST, 113
State, 187
Stirling number of the first kind, 202
Stirling number of the second kind, 202
Stirling1, 202
Stirling2, 202
STOP_TEST, 113
StoredGroebnerBasis, 1013
StoreFusion, 1166
StraightLineProgElm, 483
StraightLineProgGens, 483
StraightLineProgram
for a list of lines (and the number of generators), 478
for a string and a list of generators names, 478
StraightLineProgramNC
for a list of lines (and the number of generators), 478
for a string and a list of generators names, 478
StraightLineProgramsTom, 1056
StreamsFamily, 134
StretchImportantSLPElement, 484
strictly sorted list, 272
String, 355
for a cyclotomic, 220
StringDate, 363
StringFactorizationWord, 661
StringNumbers, 359
StringOfResultOfStraightLineProgram, 480
StringPP, 355
strings
equality of, 353
inequality of, 353
lexicographic ordering of, 353
StringTime, 364
StripLineBreakCharacters, 355
StrongGeneratorsStabChain, 606
StronglyConnectedComponents, 441
Struct, 398
*Struct*ByGenerators, 398
StructuralCopy, 156
StructuralCopy
for lists, 254
structure constant, 1091, 1092
StructureConstantsTable, 918
StructureDescription, 500
*Struct*WithGenerators, 398
SU, 725
SubadditiveGroup, 841
SubadditiveGroupNC, 841
SubadditiveMagma, 841
SubadditiveMagmaNC, 841
SubadditiveMagmaWithZero, 841
SubadditiveMagmaWithZeroNC, 841
Subalgebra, 922
SubAlgebraModule, 944
SubalgebraNC, 922
SubalgebraWithOne, 923
SubalgebraWithOneNC, 923
SubdirectProduct, 715
SubdirectProducts, 715
Subdomains, 402
Subfield, 870
SubfieldNC, 870
Subfields, 872
Subgroup, 494
for a group, 494
subgroup fusions, 1163
subgroup generators tree, 706
SubgroupByPcgs, 630
SubgroupByProperty, 496
SubgroupNC, 494
SubgroupOfWholeGroupByCosetTable, 670

SubgroupOfWholeGroupByQuotient-
Subgroup, 676
SubgroupProperty, 610
subgroups
polyhedral, 1090
SubgroupShell, 497
SubgroupsSolvableGroup, 537
sublist, 247
access, 247
assignment, 249
operation, 248
sublist assignment
operation, 250
Submagma, 454
SubmagmaNC, 454
SubmagmaWithInverses, 454
SubmagmaWithInversesNC, 454
SubmagmaWithOne, 454
SubmagmaWithOneNC, 454
Submodule, 864
SubmoduleNC, 864
Submonoid, 761
SubmonoidNC, 761
SubnearAdditiveGroup, 841
SubnearAdditiveGroupNC, 841
SubnearAdditiveMagma, 841
SubnearAdditiveMagmaNC, 841
SubnearAdditiveMagmaWithZero, 841
SubnearAdditiveMagmaWithZeroNC, 841
SubnormalSeries, 525
Subring, 846
SubringNC, 846
SubringWithOne, 851
SubringWithOneNC, 851
Subsemigroup, 757
SubsemigroupNC, 757
subset test
for collections, 388
subsets, 203
Subspace, 895
SubspaceNC, 895
Subspaces, 896
SubstitutedWord
replace a subword by a given word, 473
replace an interval by a given word, 473
SubsTom, 1046
Sub*struct*, 402
Sub*struct*NC, 402
SubSyllables, 474
subtract
a set from another, 277
SubtractBlist, 297
subtraction, 57
matrices, 311
matrix and scalar, 311
rational functions, 990
scalar and matrix, 311
scalar and matrix list, 312
scalar and vector, 300
vector and scalar, 300
vectors, 300
SubtractSet, 277
Subword, 472
Successors, 439
Sum, 286
SumFactorizationFunctionPcgs, 638
SumIntersectionMat, 325
SumX, 288
Sup, 236
SupersolvableResiduum, 513
support
email address, 27
SupportedCharacterTableInfo, 1067
SurjectiveActionHomomorphismAttr, 587
SuzukiGroup, 723
SylowComplement, 514
SylowSubgroup, 514
SylowSystem, 515
symmetric group
power map, 212
symmetric power, 1144
symmetric relation, 438
SymmetricClosureBinaryRelation, 441
SymmetricGroup
for a degree, 722
for a domain, 722
SymmetricInverseMonoid, 834
SymmetricInverseSemigroup, 834
SymmetricParentGroup, 598
SymmetricParts, 1144
SymmetricPowerOfAlgebraModule, 979
Symmetrizations, 1143
symmetrizations
orthogonal, 1144

symplectic, 1145
SymplecticComponents, 1145
SymplecticGroup
for dimension and a ring, 726
for dimension and field size, 726
syntax errors, 77
system getter, 165
system setter, 165
Sz, 723

t_N (irrational value), 225
table automorphisms, 1169, 1183
table of chapters for help books, 29
table of sections for help books, 29
TableAutomorphisms, 1111
TableOfMarks
for a group, 1041
for a matrix, 1041
for a string, 1041
TableOfMarksByLattice, 1042
TableOfMarksComponents, 1046
TableOfMarksCyclic, 1060
TableOfMarksDihedral, 1060
TableOfMarksFamily, 1045
TableOfMarksFrobenius, 1061
tables, 1062, 1065
tabulator, 47
Tan, 235
Tanh, 235
Tau, 196
TCENUM, 666
TeachingMode, 98
TemporaryGlobalVarName, 54
Tensored, 1133
TensorProductGModule, 1028
TensorProductOfAlgebraModules
for a list of algebra modules, 978
for two algebra modules, 978
Test, 114
test
for a primitive root, 193
for a rational, 216
for records, 372
for set equality, 275
TestConsistencyMaps, 1175
Tester, 166
tester, 166
TestHomogeneous, 1190
TestInducedFromNormalSubgroup, 1192
TestJacobi, 919
TestMonomial
for a character, 1192
for a character and a Boolean, 1192
for a group, 1192
for a group and a Boolean, 1192
TestMonomialQuick
for a character, 1194
for a group, 1194
TestMonomialUseLattice, 1193
TestPackageAvailability, 1201
TestPerm1, 1153
TestPerm2, 1153
TestPerm3, 1153
TestPerm4, 1153
TestPerm5, 1153
TestQuasiPrimitive, 1191
TestRelativelySM
for a character, 1195
for a character and a normal subgroup, 1195
for a group, 1195
for a group and a normal subgroup, 1195
TestSubnormallyMonomial
for a character, 1194
for a group, 1194
then, 60
ThreeGroup library, 734
TietzeWordAbstractWord, 691
time, 107
Trace
for a field element, 874
for a matrix, 874
of a matrix, 313
TracedCosetFpGroup, 665
TraceImmediateMethods, 102
TraceMat, 313
TraceMethods
for a list of operations, 101
for operations, 101
TracePolynomial, 873
TransferDiagram, 1174
Transformation
for a list and function, 791
for a source and destination, 791
for an image list, 791

TransformationByImageAndKernel
for an image and kernel, 792
TransformationFamily, 790
TransformationList
for an image list, 791
TransformationListList
for a source and destination, 791
TransformationNumber, 793
TransformationOp, 792
TransformationOpNC, 792
TransformingPermutations, 1112
TransformingPermutationsCharacterTables, 1112
transitive, 580
transitive relation, 438
TransitiveClosureBinaryRelation, 441
TransitiveGroup, 732
TransitiveIdentification, 733
Transitivity
for a character, 1132
for a group and an action domain, 580
for an external set, 580
TranslatorSubalgebra, 948
transporter, 575
TransposedMat, 316
TransposedMatAttr, 316
TransposedMatDestructive, 316
TransposedMatImmutable, 316
TransposedMatMutable, 316
TransposedMatOp, 316
TransposedMatrixGroup, 614
TriangulizedIntegerMat, 336
TriangulizedIntegerMatTransform, 336
TriangulizedMat, 318
TriangulizedNullspaceMat, 319
TriangulizedNullspaceMatDestructive, 319
TriangulizeIntegerMat, 336
TriangulizeMat, 319
TrimPartialPerm, 831
TrimTransformation, 806
TrivialCharacter
for a character table, 1126
for a group, 1126
TrivialGroup, 720
TrivialIterator, 394
TrivialSubalgebra, 923
TrivialSubgroup, 511
TrivialSubmagmaWithOne, 461
TrivialSubmodule, 865
TrivialSubmonoid, 762
TrivialSubnearAdditiveMagmaWithZero, 843
TrivialSubspace, 896
Trunc, 236
TryCosetTableInWholeGroup, 669
TryGcdCancelExtRepPolynomials, 1019
TryNextMethod, 1213
tuple stabilizer, 573
Tuples, 205
TwoClosure, 611
TwoCoboundaries, 653
TwoCocycles, 653
TwoCohomology, 653
TwoGroup library, 734
TwoSidedIdeal, 847
TwoSidedIdealByGenerators, 848
TwoSidedIdealNC, 848
TwoSquares, 198
type
boolean, 240
cyclotomic, 218
records, 372
strings, 347
TypeObj, 170
TypeOfDefaultGeneralMapping, 436
TzEliminate
for a presentation (and a generator), 698
for a presentation (and an integer), 698
TzFindCyclicJoins, 700
TzGo, 695
TzGoGo, 697
TzImagesOldGens, 704
TzInitGeneratorImages, 704
TzNewGenerator, 695
TzOptions, 710
TzPreImagesNewGens, 705
TzPrint, 693
TzPrintGeneratorImages, 705
TzPrintGenerators, 692
TzPrintLengths, 693
TzPrintOptions, 711
TzPrintPairs, 694
TzPrintPresentation, 693

TzPrintRelators, 692
TzPrintStatus, 693
TzSearch, 698
TzSearchEqual, 699
TzSort, 685
TzSubstitute
for a presentation (and an integer and 0/1/2), 700
for a presentation and a word, 700
TzSubstituteCyclicJoins, 704

u_N (irrational value), 225
UglyVector, 912
Unbind
unbind a list entry, 252
unbind a record component, 377
unbind a variable, 51
Unbind\., 378
Unbind\[\], 247
UnbindElmWPObj, 1293
UnbindGlobal, 53
UnderlyingCharacteristic
for a character, 1078
for a character table, 1078
UnderlyingCharacterTable, 1119
UnderlyingElement
fp group elements, 663
fp semigroup elements, 783
UnderlyingExternalSet, 587
UnderlyingFamily, 952
UnderlyingGeneralMapping, 424
UnderlyingGroup
for character tables, 1070
for tables of marks, 1048
UnderlyingInjectionZeroMagma, 456
UnderlyingLeftModule, 900
UnderlyingLieAlgebra, 960
UnderlyingMagma, 982
UnderlyingRelation, 424
UnderlyingRingElement, 952
UnderlyingSemigroup
for a Rees 0-matrix semigroup, 778
for a Rees matrix semigroup, 778
UnInstallCharReadHookFunc, 146
Union
for a list, 389
for various collections, 389
union
of collections, 389
of sets, 276
Union2, 389
UnionBlist
for a list, 295
for various boolean lists, 295
Unique, 278
UniteBlist, 296
UniteBlistList, 296
UniteSet, 276
Units, 854
UnivariatenessTestRationalFunction, 996
UnivariatePolynomial, 995
UnivariatePolynomialByCoefficients, 995
UnivariatePolynomialRing
for a ring (and a name and an exclusion list), 1007
for a ring (and an indeterminate number), 1007
UnivariateRationalFunctionByCoefficients, 1004
UniversalEnvelopingAlgebra, 969
UNIX
features, 33
options, 33
UNIXSelect, 135
Unknown, 1185
UnloadSmallGroupsData, 736
UnorderedTuples, 205
UnprofileFunctions, 108
UnprofileMethods, 108
until, 62
UntraceMethods
for a list of operations, 102
for operations, 102
UpdateMap, 1172
UpEnv, 88
UpperCentralSeriesOfGroup, 527
UpperSubdiagonal, 326
UseBasis, 866
UseFactorRelation, 410
UseIsomorphismRelation, 411
UserPreference, 39
UseSubsetRelation, 410
utilities for editing GAP files, 96

v_N (irrational value), 225
ValidatePackageInfo, 1204
Valuation, 1024
Value
for a univariate rat. function, a value (and a one), 999
for rat. function, a list of indeterminates, a value (and a one), 999
ValueCochain, 972
ValueGlobal, 52
ValueMolienSeries, 1147
ValueOption, 122
ValuePol, 306
ValuesOfClassFunction, 1119
VectorSpace, 894
VectorSpaceByPcgsOfElementaryAbelianGroup, 641
vi, 96
View, 81
ViewObj, 82
for a character table, 1092
for a ffe, 884
for a string, 348
for a table of marks, 1042
for class functions, 1124
ViewString, 354
vim, 96
virtual character, 1128
virtual characters, 1116
VirtualCharacter
for a character table and a list, 1125
for a group and a list, 1125

w_N (irrational value), 225
WeakPointerObj, 1292
web sites
for GAP, 27
WedgeGModule, 1028
WeekDay, 363
WeightLexOrdering, 449
WeightOfGenerators, 450
WeightsTom, 1050
WeightVecFFE, 305
WeylGroup, 963
WeylOrbitIterator, 964
Where, 87
while loop, 61
WordAlp, 356
words
in generators, 498
Wreath product embedding, 717
WreathProduct, 716
WreathProductImprimitiveAction, 716
WreathProductOrdering, 451
WreathProductProductAction, 717
WriteAll, 139
WriteByte, 138
WriteGapIniFile, 39
WriteLine, 139

X
for a family and a number, 988
for a ring (and a name, and an exclusion list), 988
for a ring (and a number), 988
x_N (irrational value), 225

y_N (irrational value), 225

Z
for field size, 876
for prime and degree, 876
ZClassRepsQClass, 620
Zero, 404
for a partial perm, 826
ZeroAttr, 404
ZeroCoefficient, 983
ZeroCoefficientRatFun, 1016
ZeroImmutable, 404
ZeroMapping, 421
ZeroMutable, 404
ZeroOp, 404
ZeroSameMutability, 404
ZeroSM, 404
Zeta, 236
ZippedProduct, 1018
ZippedSum, 1018
ZmodnZ, 184
ZmodnZObj
for a residue class family and integer, 185
for two integers, 185
ZmodpZ, 184
ZmodpZNC, 184
ZumbroichBase, 890
Zuppos, 536