GAP
Release 4.3
05 May 2002
Reference Manual

The GAP Group
http://www.gap-system.org
Acknowledgement

We would like to thank the many people who have made contributions of various kinds to the development of GAP since 1986, in particular:


The following list gives the authors, indicated by A, who designed the code in the first place as well as the current maintainers, indicated by M of the various modules of which GAP is composed. Since the process of modularization was started only recently, there might be omissions both in scope and in contributors. The compilers of the manual apologize for any such errors and promise to rectify them in future editions.

Kernel
Frank Celler (A), Steve Linton (AM), Frank Lübeck (AM), Werner Nickel (AM), Martin Schönert (A)

Automorphism groups of finite pc groups
Bettina Eick (AM)

Binary Relations
Robert Morse (AM), Andrew Solomon (A)

Classes in nonsolvable groups
Alexander Hulpke (AM)

Classical Groups
Thomas Breuer (AM), Frank Celler (A), Stefan Kohl (AM), Frank Lübeck (AM), Heiko Theißen (A)

Congruences of magmas, semigroups and monoids
Robert Morse (AM), Andrew Solomon (A)
Cosets and Double Cosets
  Alexander Hulpke (AM)

Cyclotomics
  Thomas Breuer (AM)

Dixon-Schneider Algorithm
  Alexander Hulpke (AM)

Documentation Utilities
  Frank Celler (A), Heiko Theißen (A), Alexander Hulpke (A), Willem de Graaf (A), Steve Linton (A), Werner Nickel (A), Greg Gamble (AM)

Factor groups
  Alexander Hulpke (AM)

Finitely presented groups
  Volkmann Felsch (AM), Alexander Hulpke (AM), Martin Schoenert (A)

Finitely presented monoids and semigroups
  Isabel Araujo (AM), Derek Holt (A), Alexander Hulpke (A), Götz Pfeiffer (A), Andrew Solomon (AM)

Group actions
  Heiko Theissen (A) and Alexander Hulpke (AM)

Homomorphism search
  Alexander Hulpke (AM)

Homomorphisms for finitely presented groups
  Alexander Hulpke (AM)

Intersection of subgroups of finite pc groups
  Frank Celler (A), Bettina Eick (AM)

Irreducible Modules over finite fields for finite pc groups
  Bettina Eick (AM)

Isomorphism testing with random methods
  Hans Ulrich Besche (AM), Bettina Eick (AM)

Multiplier and Schur cover
  Werner Nickel (A), Alexander Hulpke (AM)

One-Cohomology and Complements
  Frank Celler (A), Alexander Hulpke (AM)

Partition Backtrack algorithm
  Heiko Theissen (A), Alexander Hulpke (M)

Permutation group composition series
  Ákos Seress (AM)

Permutation group homomorphisms
  Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Permutation Group Pegs
  Heiko Theißen (A), Alexander Hulpke (M)

Primitive groups library
  Heiko Theissen (A), Alexander Hulpke (M)

Properties and attributes of finite pc groups
  Frank Celler (A), Bettina Eick (AM)

Random Schreier-Sims
  Ákos Seress (AM)
Acknowledgement

Rational Functions
Frank Celler (A) and Alexander Hulpke (AM)

Semigroup relations
Isabel Araujo (A), Robert F. Morse (AM), Andrew Solomon (A)

Special Pcs for finite pc groups
Bettina Eick (AM)

Stabilizer Chains
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Strings and Characters
Martin Schönert (A), Frank Celler (A), Thomas Breuer (A), Frank Lübeck (AM)

Subgroup lattice
Martin Schönert (A), Alexander Hulpke (AM)

Subgroup lattice for solvable groups
Alexander Hulpke (AM)

Subgroup presentations
Volkmar Felsch (AM)

The Help System
Frank Celler (A), Frank Lübeck (AM)

Tietze transformations
Volkmar Felsch (AM)

Transformation semigroups
Isabel Araujo (A), Robert Arthur (A), Robert F. Morse (AM), Andrew Solomon (A)

Transitive groups library
Alexander Hulpke (AM)

Two-cohomology and extensions of finite pc groups
Bettina Eick (AM)

Lie algebras
Thomas Breuer (A), Craig Struble (A), Juergen Wisliceny (A), Willem A. de Graaf (AM)

GAP for MacOS
Burkhard Höfling (AM)
Copyright Notice

1 About the GAP Reference Manual 19

1.1 Manual Conventions 21

1.2 Credit 22

2 The Help System 23

2.1 Invoking the Help 23

2.2 Browsing through the Sections 23

2.3 Changing the Help Viewer 24

2.4 The Pager Command 26

3 Running GAP 28

3.1 Command Line Options 28

3.2 Advanced Features of GAP 30

3.3 Running GAP under MacOS 32

3.4 The .gaprc file 34

3.5 Completion Files 35

3.6 The Compiler 35

3.7 Suitability for Compilation 36

3.8 Compiling Library Code 36

3.9 CRC Numbers 37

3.10 Saving and Loading a Workspace 37

4 The Programming Language 39

4.1 Language Overview 39

4.2 Lexical Structure 40

4.3 Symbols 40

4.4 Whitespaces 41

4.5 Keywords 41

4.6 Identifiers 42

4.7 Expressions 42

4.8 Variables 43

4.9 More About Global Variables 44

4.10 Function Calls 46

4.11 Comparisons 47

4.12 Arithmetic Operators 48

4.13 Statements 49

4.14 Assignments 50

4.15 Procedure Calls 50

4.16 If 51

4.17 While 52

4.18 Repeat 52

4.19 For 53

4.20 Break 55

4.21 Continue 55

4.22 Function 55

4.23 Return 58

4.24 The Syntax in BNF 59

5 Functions 61

5.1 Information about a function 61

5.2 Calling a function with a list argument that is interpreted as several arguments 62

5.3 Functions that do nothing 62

5.4 Function Types 63
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Main Loop and Break Loop</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>Main Loop</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>View and Print</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>Break Loops</td>
<td>67</td>
</tr>
<tr>
<td>6.4</td>
<td>Variable Access in a Break Loop</td>
<td>71</td>
</tr>
<tr>
<td>6.5</td>
<td>Error</td>
<td>72</td>
</tr>
<tr>
<td>6.6</td>
<td>ErrorCount</td>
<td>73</td>
</tr>
<tr>
<td>6.7</td>
<td>Leaving GAP</td>
<td>73</td>
</tr>
<tr>
<td>6.8</td>
<td>Line Editing</td>
<td>73</td>
</tr>
<tr>
<td>6.9</td>
<td>Editing Files</td>
<td>75</td>
</tr>
<tr>
<td>6.10</td>
<td>Editor Support</td>
<td>75</td>
</tr>
<tr>
<td>6.11</td>
<td>SizeScreen</td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>Debugging and Profiling Facilities</td>
<td>77</td>
</tr>
<tr>
<td>7.1</td>
<td>Recovery from NoMethodFound-Errors</td>
<td>77</td>
</tr>
<tr>
<td>7.2</td>
<td>ApplicableMethod</td>
<td>78</td>
</tr>
<tr>
<td>7.3</td>
<td>Tracing Methods</td>
<td>79</td>
</tr>
<tr>
<td>7.4</td>
<td>Info Functions</td>
<td>80</td>
</tr>
<tr>
<td>7.5</td>
<td>Assertions</td>
<td>81</td>
</tr>
<tr>
<td>7.6</td>
<td>Timing</td>
<td>81</td>
</tr>
<tr>
<td>7.7</td>
<td>Profiling</td>
<td>82</td>
</tr>
<tr>
<td>7.8</td>
<td>Information about the version used</td>
<td>84</td>
</tr>
<tr>
<td>7.9</td>
<td>Test Files</td>
<td>84</td>
</tr>
<tr>
<td>7.10</td>
<td>Debugging Recursion</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>Options Stack</td>
<td>87</td>
</tr>
<tr>
<td>9</td>
<td>Files and Filenames</td>
<td>89</td>
</tr>
<tr>
<td>9.1</td>
<td>Portability</td>
<td>89</td>
</tr>
<tr>
<td>9.2</td>
<td>GAP Root Directory</td>
<td>89</td>
</tr>
<tr>
<td>9.3</td>
<td>Directories</td>
<td>90</td>
</tr>
<tr>
<td>9.4</td>
<td>Filename</td>
<td>91</td>
</tr>
<tr>
<td>9.5</td>
<td>Special Filenames</td>
<td>92</td>
</tr>
<tr>
<td>9.6</td>
<td>File Access</td>
<td>92</td>
</tr>
<tr>
<td>9.7</td>
<td>File Operations</td>
<td>93</td>
</tr>
<tr>
<td>10</td>
<td>Streams</td>
<td>96</td>
</tr>
<tr>
<td>10.1</td>
<td>Categories for Streams and the StreamsFamily</td>
<td>96</td>
</tr>
<tr>
<td>10.2</td>
<td>Operations applicable to All Streams</td>
<td>97</td>
</tr>
<tr>
<td>10.3</td>
<td>Operations for Input Streams</td>
<td>98</td>
</tr>
<tr>
<td>10.4</td>
<td>Operations for Output Streams</td>
<td>100</td>
</tr>
<tr>
<td>10.5</td>
<td>File Streams</td>
<td>102</td>
</tr>
<tr>
<td>10.6</td>
<td>User Streams</td>
<td>103</td>
</tr>
<tr>
<td>10.7</td>
<td>String Streams</td>
<td>103</td>
</tr>
<tr>
<td>10.8</td>
<td>Input-Output Streams</td>
<td>104</td>
</tr>
<tr>
<td>10.9</td>
<td>Dummy Streams</td>
<td>105</td>
</tr>
<tr>
<td>10.10</td>
<td>Handling of Streams in the Background</td>
<td>105</td>
</tr>
<tr>
<td>11</td>
<td>Processes</td>
<td>107</td>
</tr>
<tr>
<td>11.1</td>
<td>Process</td>
<td>107</td>
</tr>
<tr>
<td>11.2</td>
<td>Exec</td>
<td>108</td>
</tr>
<tr>
<td>12</td>
<td>Objects and Elements</td>
<td>109</td>
</tr>
<tr>
<td>12.1</td>
<td>Objects</td>
<td>109</td>
</tr>
<tr>
<td>12.2</td>
<td>Elements as equivalence classes</td>
<td>109</td>
</tr>
<tr>
<td>12.3</td>
<td>Sets</td>
<td>110</td>
</tr>
<tr>
<td>12.4</td>
<td>Domains</td>
<td>110</td>
</tr>
<tr>
<td>12.5</td>
<td>Identical Objects</td>
<td>110</td>
</tr>
<tr>
<td>12.6</td>
<td>Mutability and Copyability</td>
<td>111</td>
</tr>
<tr>
<td>12.7</td>
<td>Duplication of Objects</td>
<td>113</td>
</tr>
<tr>
<td>12.8</td>
<td>Other Operations Applicable to any Object</td>
<td>114</td>
</tr>
<tr>
<td>13</td>
<td>Types of Objects</td>
<td>115</td>
</tr>
<tr>
<td>13.1</td>
<td>Families</td>
<td>115</td>
</tr>
<tr>
<td>13.2</td>
<td>Filters</td>
<td>116</td>
</tr>
<tr>
<td>13.3</td>
<td>Categories</td>
<td>117</td>
</tr>
<tr>
<td>13.4</td>
<td>Representation</td>
<td>119</td>
</tr>
<tr>
<td>13.5</td>
<td>Attributes</td>
<td>120</td>
</tr>
<tr>
<td>13.6</td>
<td>Setter and Tester for Attributes</td>
<td>121</td>
</tr>
<tr>
<td>13.7</td>
<td>Properties</td>
<td>123</td>
</tr>
</tbody>
</table>
23.2 Row Vectors over Finite Fields . . . 210
23.3 Coefficient List Arithmetic . . . . 211
23.4 Shifting and Trimming Coefficient Lists 212
23.5 Functions for Coding Theory . . . . 212
23.6 Vectors as coefficients of polynomials 213

24 Matrices 215
24.1 Categories of Matrices . . . . . 215
24.2 Operators for Matrices . . . . . 216
24.3 Properties and Attributes of Matrices 218
24.4 Matrix Constructions . . . . . 219
24.5 Random Matrices . . . . . . . 222
24.6 Matrices Representing Linear Equations and the Gaussian Algorithm . . . 222
24.7 Eigenvectors and eigenvalues . . . 223
24.8 Elementary Divisors . . . . . 224
24.9 Echelonized Matrices . . . . . 224
24.10 Matrices as Basis of a Row Space . 225
24.11 Triangular Matrices . . . . . . 226
24.12 Matrices as Linear Mappings . . . 227
24.13 Matrices over Finite Fields . . . 228
24.14 Block Matrices . . . . . . . 230

25 Integral matrices and lattices 232
25.1 Normal Forms over the Integers . . 232
25.2 Decompositions . . . . . . . 235
25.3 Lattice Reduction . . . . . 237
25.4 Orthogonal Embeddings . . . . 238

26 Strings and Characters 240
26.1 Special Characters . . . . . 242
26.2 Internally Represented Strings . . 243
26.3 Recognizing Characters . . . . 244
26.4 Comparisons of Strings . . . . 244
26.5 Operations to Produce or Manipulate Strings . . . . . . . 245
26.6 Operations to Evaluate Strings . . 247
26.7 Calendar Arithmetic . . . . . . 249

27 Records 251
27.1 Accessing Record Elements . . . . 252
27.2 Record Assignment . . . . . . 252
27.3 Identical Records . . . . . . 253
27.4 Comparisons of Records . . . . 254
27.5 IsBound and Unbind for Records . 255
27.6 Record Access Operations . . . . 256

28 Collections 257
28.1 Collection Families . . . . . . 257
28.2 Lists and Collections . . . . . 258
28.3 Attributes and Properties for Collections . . . . . . . . . . . 261
28.4 Operations for Collections . . . . 263
28.5 Membership Test for Collections . 264
28.6 Random Elements . . . . . . 265
28.7 Iterators . . . . . . . . . 266

29 Orderings 269
29.1 Building new orderings . . . . . 269
29.2 Properties and basic functionality . 270
29.3 Orderings on families of associative words . . . . . . . . . . . 271

30 Domains and their Elements 275
30.1 Operational Structure of Domains . 275
30.2 Equality and Comparison of Domains 276
30.3 Constructing Domains . . . . . 276
30.4 Changing the Structure . . . . . 277
30.5 Changing the Representation . . . 278
30.6 Domain Categories . . . . . . 278
30.7 Parents . . . . . . . . . 279
30.8 Constructing Subdomains . . . . 280
30.9 Operations for Domains . . . . 280
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.10</td>
<td>Attributes and Properties of Elements</td>
<td>281</td>
</tr>
<tr>
<td>30.11</td>
<td>Comparison Operations for Elements</td>
<td>284</td>
</tr>
<tr>
<td>30.12</td>
<td>Arithmetic Operations for Elements</td>
<td>285</td>
</tr>
<tr>
<td>30.13</td>
<td>Relations Between Domains</td>
<td>285</td>
</tr>
<tr>
<td>30.14</td>
<td>Useful Categories of Elements</td>
<td>288</td>
</tr>
<tr>
<td>30.15</td>
<td>Useful Categories for all Elements of a Family</td>
<td>290</td>
</tr>
<tr>
<td>31</td>
<td>Mappings</td>
<td>292</td>
</tr>
<tr>
<td>31.1</td>
<td>Creating Mappings</td>
<td>293</td>
</tr>
<tr>
<td>31.2</td>
<td>Properties and Attributes of (General) Mappings</td>
<td>294</td>
</tr>
<tr>
<td>31.3</td>
<td>Images under Mappings</td>
<td>295</td>
</tr>
<tr>
<td>31.4</td>
<td>Preimages under Mappings</td>
<td>296</td>
</tr>
<tr>
<td>31.5</td>
<td>Arithmetic Operations for General Mappings</td>
<td>298</td>
</tr>
<tr>
<td>31.6</td>
<td>Mappings which are Compatible with Algebraic Structures</td>
<td>298</td>
</tr>
<tr>
<td>31.7</td>
<td>Magma Homomorphisms</td>
<td>298</td>
</tr>
<tr>
<td>31.8</td>
<td>Mappings that Respect Multiplication</td>
<td>299</td>
</tr>
<tr>
<td>31.9</td>
<td>Mappings that Respect Addition</td>
<td>300</td>
</tr>
<tr>
<td>31.10</td>
<td>Linear Mappings</td>
<td>300</td>
</tr>
<tr>
<td>31.11</td>
<td>Ring Homomorphisms</td>
<td>301</td>
</tr>
<tr>
<td>31.12</td>
<td>General Mappings</td>
<td>301</td>
</tr>
<tr>
<td>31.13</td>
<td>Technical Matters Concerning General Mappings</td>
<td>301</td>
</tr>
<tr>
<td>32</td>
<td>Relations</td>
<td>303</td>
</tr>
<tr>
<td>32.1</td>
<td>General Binary Relations</td>
<td>303</td>
</tr>
<tr>
<td>32.2</td>
<td>Properties and Attributes of Binary Relations</td>
<td>303</td>
</tr>
<tr>
<td>32.3</td>
<td>Binary Relations on Points</td>
<td>305</td>
</tr>
<tr>
<td>32.4</td>
<td>Closure Operations and Other Constructors</td>
<td>305</td>
</tr>
<tr>
<td>32.5</td>
<td>Equivalence Relations</td>
<td>306</td>
</tr>
<tr>
<td>32.6</td>
<td>Attributes of and Operations on Equivalence Relations</td>
<td>306</td>
</tr>
<tr>
<td>32.7</td>
<td>Equivalence Classes</td>
<td>307</td>
</tr>
<tr>
<td>33</td>
<td>Magmas</td>
<td>308</td>
</tr>
<tr>
<td>33.1</td>
<td>Magma Categories</td>
<td>308</td>
</tr>
<tr>
<td>33.2</td>
<td>Magma Generation</td>
<td>309</td>
</tr>
<tr>
<td>33.3</td>
<td>Magmas Defined by Multiplication Tables</td>
<td>310</td>
</tr>
<tr>
<td>33.4</td>
<td>Attributes and Properties for Magmas</td>
<td>311</td>
</tr>
<tr>
<td>34</td>
<td>Words</td>
<td>314</td>
</tr>
<tr>
<td>34.1</td>
<td>Categories of Words and Nonassociative Words</td>
<td>314</td>
</tr>
<tr>
<td>34.2</td>
<td>Comparison of Words</td>
<td>316</td>
</tr>
<tr>
<td>34.3</td>
<td>Operations for Words</td>
<td>316</td>
</tr>
<tr>
<td>34.4</td>
<td>Free Magmas</td>
<td>317</td>
</tr>
<tr>
<td>34.5</td>
<td>External Representation for Nonassociative Words</td>
<td>318</td>
</tr>
<tr>
<td>35</td>
<td>Associative Words</td>
<td>319</td>
</tr>
<tr>
<td>35.1</td>
<td>Categories of Associative Words</td>
<td>319</td>
</tr>
<tr>
<td>35.2</td>
<td>Free Groups, Monoids and Semigroups</td>
<td>320</td>
</tr>
<tr>
<td>35.3</td>
<td>Comparison of Associative Words</td>
<td>322</td>
</tr>
<tr>
<td>35.4</td>
<td>Operations for Associative Words</td>
<td>323</td>
</tr>
<tr>
<td>35.5</td>
<td>Operations for Associative Words by their Syllables</td>
<td>324</td>
</tr>
<tr>
<td>35.6</td>
<td>Representations for Associative Words</td>
<td>325</td>
</tr>
<tr>
<td>35.7</td>
<td>The External Representation for Associative Words</td>
<td>327</td>
</tr>
<tr>
<td>35.8</td>
<td>Straight Line Programs</td>
<td>327</td>
</tr>
<tr>
<td>35.9</td>
<td>Straight Line Program Elements</td>
<td>331</td>
</tr>
<tr>
<td>36</td>
<td>Rewriting Systems</td>
<td>332</td>
</tr>
<tr>
<td>36.1</td>
<td>Operations on rewriting systems</td>
<td>332</td>
</tr>
<tr>
<td>36.2</td>
<td>Operations on elements of the algebra</td>
<td>333</td>
</tr>
<tr>
<td>36.3</td>
<td>Properties of rewriting systems</td>
<td>334</td>
</tr>
<tr>
<td>36.4</td>
<td>Developing rewriting systems</td>
<td>334</td>
</tr>
<tr>
<td>37</td>
<td>Groups</td>
<td>335</td>
</tr>
<tr>
<td>37.1</td>
<td>Group Elements</td>
<td>335</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.2</td>
<td>Creating Groups</td>
<td>335</td>
</tr>
<tr>
<td>37.3</td>
<td>Subgroups</td>
<td>337</td>
</tr>
<tr>
<td>37.4</td>
<td>Closures of (Sub)groups</td>
<td>338</td>
</tr>
<tr>
<td>37.5</td>
<td>Expressing Group Elements as Words in Generators</td>
<td>339</td>
</tr>
<tr>
<td>37.6</td>
<td>Cosets</td>
<td>340</td>
</tr>
<tr>
<td>37.7</td>
<td>Transversals</td>
<td>341</td>
</tr>
<tr>
<td>37.8</td>
<td>Double Cosets</td>
<td>342</td>
</tr>
<tr>
<td>37.9</td>
<td>Conjugacy Classes</td>
<td>343</td>
</tr>
<tr>
<td>37.10</td>
<td>Normal Structure</td>
<td>345</td>
</tr>
<tr>
<td>37.11</td>
<td>Specific and Parametrized Subgroups</td>
<td>346</td>
</tr>
<tr>
<td>37.12</td>
<td>Sylow Subgroups and Hall Subgroups</td>
<td>348</td>
</tr>
<tr>
<td>37.13</td>
<td>Subgroups characterized by prime powers</td>
<td>349</td>
</tr>
<tr>
<td>37.14</td>
<td>Group Properties</td>
<td>349</td>
</tr>
<tr>
<td>37.15</td>
<td>Numerical Group Attributes</td>
<td>352</td>
</tr>
<tr>
<td>37.16</td>
<td>Subgroup Series</td>
<td>352</td>
</tr>
<tr>
<td>37.17</td>
<td>Factor Groups</td>
<td>355</td>
</tr>
<tr>
<td>37.18</td>
<td>Sets of Subgroups</td>
<td>356</td>
</tr>
<tr>
<td>37.19</td>
<td>Subgroup Lattice</td>
<td>358</td>
</tr>
<tr>
<td>37.20</td>
<td>Specific Methods for Subgroup Lattice Computations</td>
<td>360</td>
</tr>
<tr>
<td>37.21</td>
<td>Special Generating Sets</td>
<td>363</td>
</tr>
<tr>
<td>37.22</td>
<td>1-Cohomology</td>
<td>363</td>
</tr>
<tr>
<td>37.23</td>
<td>Schur Covers and Multipliers</td>
<td>366</td>
</tr>
<tr>
<td>37.24</td>
<td>Tests for the Availability of Methods</td>
<td>366</td>
</tr>
<tr>
<td>38.7</td>
<td>Groups of Automorphisms</td>
<td>375</td>
</tr>
<tr>
<td>38.8</td>
<td>Calculating with Group Automorphisms</td>
<td>376</td>
</tr>
<tr>
<td>38.9</td>
<td>Searching for Homomorphisms</td>
<td>377</td>
</tr>
<tr>
<td>38.10</td>
<td>Representations for Group Homomorphisms</td>
<td>379</td>
</tr>
<tr>
<td>39.1</td>
<td>About Group Actions</td>
<td>380</td>
</tr>
<tr>
<td>39.2</td>
<td>Basic Actions</td>
<td>381</td>
</tr>
<tr>
<td>39.3</td>
<td>Orbits</td>
<td>383</td>
</tr>
<tr>
<td>39.4</td>
<td>Stabilizers</td>
<td>384</td>
</tr>
<tr>
<td>39.5</td>
<td>Elements with Prescribed Images</td>
<td>385</td>
</tr>
<tr>
<td>39.6</td>
<td>The Permutation Image of an Action</td>
<td>386</td>
</tr>
<tr>
<td>39.7</td>
<td>Action of a group on itself</td>
<td>387</td>
</tr>
<tr>
<td>39.8</td>
<td>Permutations Induced by Elements and Cycles</td>
<td>388</td>
</tr>
<tr>
<td>39.9</td>
<td>Tests for Actions</td>
<td>389</td>
</tr>
<tr>
<td>39.10</td>
<td>Block Systems</td>
<td>390</td>
</tr>
<tr>
<td>39.11</td>
<td>External Sets</td>
<td>392</td>
</tr>
<tr>
<td>39.12</td>
<td>Legacy Operations</td>
<td>394</td>
</tr>
<tr>
<td>40.1</td>
<td>Permutations</td>
<td>396</td>
</tr>
<tr>
<td>40.2</td>
<td>Moved Points of Permutations</td>
<td>397</td>
</tr>
<tr>
<td>40.3</td>
<td>Sign and Cycle Structure</td>
<td>398</td>
</tr>
<tr>
<td>40.4</td>
<td>Creating Permutations</td>
<td>398</td>
</tr>
<tr>
<td>41.1</td>
<td>The Natural Action</td>
<td>400</td>
</tr>
<tr>
<td>41.2</td>
<td>Computing a Permutation Representation</td>
<td>401</td>
</tr>
<tr>
<td>41.3</td>
<td>Symmetric and Alternating Groups</td>
<td>401</td>
</tr>
<tr>
<td>41.4</td>
<td>Primitive Groups</td>
<td>402</td>
</tr>
<tr>
<td>41.5</td>
<td>Stabilizer Chains</td>
<td>403</td>
</tr>
<tr>
<td>41.6</td>
<td>Randomized Methods for Permutation Groups</td>
<td>404</td>
</tr>
</tbody>
</table>
41.7 Construction of Stabilizer Chains   406
41.8 Stabilizer Chain Records         407
41.9 Operations for Stabilizer Chains 409
41.10 Low Level Routines to Modify and Create Stabilizer Chains 410
41.11 Backtrack                      411
41.12 Working with large degree permutation groups                     412

42 Matrix Groups                   414
42.1 Attributes and Properties for Matrix Groups                      414
42.2 Actions of Matrix Groups                                           415
42.3 GL and SL                                                          415
42.4 Invariant Forms                                                   416
42.5 Matrix Groups in Characteristic 0                                 417
42.6 Acting OnRight and OnLeft                                          419

43 Polycyclic Groups            420
43.1 Polycyclic Generating Systems                                     420
43.2 Computing a Pcgs                                                   421
43.3 Defining a Pcgs Yourself                                           421
43.4 Elementary Operations for a Pcgs                                  422
43.5 Elementary Operations for a Pcgs and an Element                   423
43.6 Exponents of Special Products                                     424
43.7 Subgroups of Polycyclic Groups - Induced Pegs                    425
43.8 Subgroups of Polycyclic Groups - Canonical Pegs                   426
43.9 Factor Groups of Polycyclic Groups - Modulo Pegs                  427
43.10 Factor Groups of Polycyclic Groups in their Own Representation   428
43.11 Pegs and Normal Series                                           429
43.12 Sum and Intersection of Pegs                                      430
43.13 Special Pegs                                                      430
43.14 Action on Subfactors Defined by a Pegs                            432
43.15 Orbit Stabilizer Methods for Polycyclic Groups                   434
43.16 Operations which have Special Methods for Groups with Pegs       434
43.17 Conjugacy Classes in Solvable Groups                             434

44 Pc Groups                    436
44.1 The family pcgs                                                   437
44.2 Elements of pc groups                                             437
44.3 Pc groups versus fp groups                                        438
44.4 Constructing Pc Groups                                            438
44.5 Computing Pc Groups                                               439
44.6 Saving a Pc Group                                                 441
44.7 Operations for Pc Groups                                          441
44.8 2-Cohomology and Extensions                                       441
44.9 Coding a Pc Presentation                                          444
44.10 Random Isomorphism Testing                                       445

45 Finitely Presented Groups  446
45.1 Creating Finitely Presented Groups                                447
45.2 Comparison of Elements of Finitely Presented Groups               448
45.3 Preimages in the Free Group                                       448
45.4 Operations for Finitely Presented Groups                         449
45.5 Coset Tables and Coset Enumeration                                450
45.6 Standardization of coset tables                                  453
45.7 Coset tables for subgroups in the whole group                    454
45.8 Augmented Coset Tables and Rewriting                             454
45.9 Low Index Subgroups                                               455
45.10 Converting Groups to Finitely Presented Groups                  456
45.11 New Presentations and Presentations for Subgroups               458
Contents

45.12 Preimages under Homomorphisms from an FpGroup . . . . . . . . . 459
45.13 Quotient Methods . . . . . . . 460
45.14 Abelian Invariants for Subgroups . . . . 461
45.15 Testing Finiteness of Finitely Presented Groups . . . . . . . . 462

46 Presentations and Tietze Transformations 464
46.1 Creating Presentations . . . . . 464
46.2 SimplifiedFpGroup . . . . . . 466
46.3 Subgroup Presentations . . . . . 467
46.4 Relators in a Presentation . . . . 470
46.5 Printing Presentations . . . . . 470
46.6 Changing Presentations . . . . . 472
46.7 Tietze Transformations . . . . . 472
46.8 Elementary Tietze Transformations 475
46.9 Tietze Transformations that introduce new Generators . . . . . . . 477
46.10 Tracing generator images through Tietze transformations . . . . . 480
46.11 DecodeTree . . . . . . . . . 482
46.12 Tietze Options . . . . . . . . 485

47 Group Products 487
47.1 Direct Products . . . . . . . . 487
47.2 Semidirect Products . . . . . . 488
47.3 Subdirect Products . . . . . . 489
47.4 Wreath Products . . . . . . . 490
47.5 Embeddings and Projections for Group Products . . . . . . . . 491

48 Group Libraries 492
48.1 Basic Groups . . . . . . . . . 492
48.2 Classical Groups . . . . . . . 494
48.3 Conjugacy Classes in Classical Groups . . . . 498
48.4 Constructors for Basic Groups . . . . . 498
48.5 Selection Functions . . . . . . . 499
48.6 Transitive Permutation Groups . . . . 500
48.7 Small Groups . . . . . . . . 501
48.8 Finite Perfect Groups . . . . . . 504
48.9 Primitive Permutation Groups . . . . 508
48.10 Irreducible Solvable Matrix Groups 510
48.11 Irreducible Maximal Finite Integral Matrix Groups . . . . . . . 511

49 Semigroups 520
49.1 Making transformation semigroups 522
49.2 Ideals of semigroups . . . . . . 522
49.3 Congruences for semigroups . . . . 523
49.4 Quotients . . . . . . . . . 523
49.5 Green’s Relations . . . . . . . 523
49.6 Rees Matrix Semigroups . . . . . 525

50 Monoids 527
51 Finitely Presented Semigroups and Monoids 529
51.1 Creating Finitely Presented Semigroups 531
51.2 Comparison of Elements of Finitely Presented Semigroups . . . . . 532
51.3 Preimages in the Free Semigroup . . . . 532
51.4 Finitely presented monoids . . . . . 533
51.5 Rewriting Systems and the Knuth-Bendix Procedure . . . . . . 534
51.6 Todd-Coxeter Procedure . . . . . 535

52 Transformations 536
53 Additive Magmas (preliminary) 539
53.1 (Near-)Additive Magma Categories 539
53.2 (Near-)Additive Magma Generation 540
53.3 Attributes and Properties for (Near-)Additive Magmas . . . . . . 541
53.4 Operations for (Near-)Additive Magmas . . . . . . . . 542
71.2 Class Fusions between Character Tables 773
71.3 Parametrized Maps . . . . . . 778
71.4 Subroutines for the Construction of Power Maps . . . . . . . 785
71.5 Subroutines for the Construction of Class Fusions . . . . . . . 788

72 Monomiality Questions 790
72.1 Character Degrees and Derived Length 791
72.2 Primitivity of Characters . . . . 791
72.3 Testing Monomiality . . . . . . 793
72.4 Minimal Nonmonomial Groups . . 796

73 Installing GAP 797
73.1 Installation for the Impatient . . . 797
73.2 Getting GAP . . . . . . . . . 798
73.3 GAP for UNIX . . . . . . . . 800
73.4 Installation of GAP for UNIX . . 800
73.5 Known Problems of the Configure Process . . . . . . . . . . 805
73.6 Problems on Particular Systems . . 805
73.7 Optimization and Compiler Options 806
73.8 GAP for Windows . . . . . . . 806
73.9 Copyright of GAP for Windows . . 807
73.10 Installation of GAP for Windows . 807
73.11 Expert Windows installation . . . 810
73.12 GAP for MacOS . . . . . . . . 811
73.13 Copyright of GAP for MacOS . . 811
73.14 Installation of GAP for MacOS . . 812
73.15 Testing for the System Architecture 814
73.16 Porting GAP . . . . . . . . . 815
73.17 The Documentation . . . . . . 816
73.18 HTML Font Setup . . . . . . 817
73.19 If Things Go Wrong . . . . . . 817

74 GAP Packages 819
74.1 Installing GAP Packages . . . . 819
74.2 Installing a GAP Package in your home directory . . . . . . . . 820
74.3 Loading a GAP Package . . . . 821
74.4 Functions for GAP Packages . . . 822

Bibliography 825
Index 831
Copyright Notice

Copyright © (1987–2002) 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. See http://www.gnu.org/licenses/gpl.html or the file ‘GPL’ in the etc’ directory of the GAP distribution for details.

If you obtain GAP please send us a short notice to that effect, e.g., an e-mail message to the address gap@dcs.st-and.ac.uk, containing your full name and address. This allows 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.

Specifically, please refer to

[GAP] The GAP Group, GAP --- Groups, Algorithms, and Programming,
   Version 4.3; 2002
   (http://www.gap-system.org)

(Should the reference style require full addresses please use: “Centre for Interdisciplinary Research in Computational Algebra, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland; Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany”)

GAP is distributed by us without any warranty, to the extent permitted by applicable state law. We distribute GAP as is without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

The entire risk as to the quality and performance of the program is with you. Should GAP prove defective, you assume the cost of all necessary servicing, repair or correction.

In no case unless required by applicable law will we, and/or any other party who may modify and redistribute GAP as permitted above, be liable to you for damages, including lost profits, lost monies or other special, incidental or consequential damages arising out of the use or inability to use GAP.

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 bug-fixes, improvements and new functions. So again we would appreciate it if you would inform us about all modifications you make.
1 About the GAP Reference Manual

This is one of four parts of the GAP documentation, the others being the GAP Tutorial, a beginner’s introduction to GAP, Programming in GAP and Extending GAP, which provide information for those who want to write their own GAP extensions.

This manual, the GAP reference manual contains the official definitions of GAP functions. It should give all information to someone who wants to use GAP as it is. It is not intended to be read cover-to-cover.

This manual is divided into chapters. Each chapter is divided into sections and, within each section, important definitions are numbered. References are therefore triples.

Chapter 2 describes the help system, which provides online access to the information of the manual. Chapter 3 gives technical advice for running GAP. Chapter 4 introduces the GAP language, while the next chapters deal with the environment provided by GAP for the user. These are followed by the main bulk of chapters which is devoted to various mathematical structures that GAP can handle.

Pages are numbered consecutively in each of the four manuals.

1.1 Manual Conventions

The printed manual uses different text styles for several purposes. Note that the online help may use other symbols to express the meanings listed below.

*text* Text printed in boldface is used to emphasize single words or phrases.

_text_ Text printed in italics is used for arguments in the descriptions of functions and for other place holders. It means that you should not actually enter this text into GAP but replace it by appropriate text depending on what you want to do. For example when we write that you should enter ?section to see the section with the name section, section serves as a place holder, indicating that you can enter the name of the section that you want to see at this place.

_text_ Text printed in a monospaced (all characters have the same width) typewriter font is used for names of variables and functions and other text that you may actually enter into your computer and see on your screen. Such text may contain place holders printed in italics as described above. For example when the information for IsPrime says that the form of the call is IsPrime( n ) this means that you should actually enter the strings “IsPrime)” and “)”, without the quotes, but replace the n with the number (or expression) that you want to test.

Oper( arg1, arg2[, opt] ) F

starts a subsection on the command Oper that takes two arguments arg1 and arg2 and an optional third argument opt. As in the above example, the letter F at the end of a line that starts with a little black triangle in the left margin indicates that the command is a simple function. Other possible letters at the
Chapter 1. About the GAP Reference Manual

end of such a line are \texttt{A}, \texttt{P}, \texttt{O}, \texttt{C}, \texttt{R}, and \texttt{V}; they indicate “Attribute”, “Property”, “Operation”, “Category”, “Representation” (see Chapter 13), or “Variable”, respectively.

In the printed manual, \textbf{mathematical formulas} are typeset in italics (actually math italics), and subscripts and superscripts are actually lowered and raised.

Longer \textbf{examples} are usually paragraphs of their own. Everything on the lines with the prompts \texttt{gap>}, except the prompts themselves of course, is the input you have to type; everything else is GAP’s response.

In the printed manual, examples are printed in a monospaced typewriter font.

1.2 Credit

The manual tries to give credit to designers and implementors of major parts of GAP. For many parts of the GAP code it is impossible to give detailed credit, because over the time of its development many persons have contributed from first ideas, even in prerunners of GAP such as CAS or SOGOS, via first implementations, improvements, and even total reimplementations. The documentation of the code gives further details, but again, it suffers from the same problem. We have attempted to give attributions with the different chapters of the manual where this seemed to be possible, but we apologise for all (unavoidable) shortcomings of this attempt.
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.

1 ▶ ?[book:] [?] topic

For an explanation and some examples see 2.7.

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.

As 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 dvi-files (produced by TeX) or files in Acrobat’s pdf-format or via a Web-browser. This is explained in section 2.3.

Details of the string normalization process

Here now is precisely 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.

1 ▶ ?>

The two help commands ?< and ?> allow to browse through a whole help book. ?< displays the section preceding the previously shown section, and ?> takes you to the section following the previously shown one.
Chapter 2. The Help System

2▷ ?>>
▷ ??<

?<< takes you back to the first section of the current chapter, which gives an overview of the sections described in this chapter. If you are already in this section ?<< takes you to the first section of the previous chapter. ?>> takes you to the first section of the next chapter.

3▷ ?-
▷ ?+

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.

4▷ ?books

This command shows a list of 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 RequirePackage (see 74.3.1) before you can use the described functions.

5▷ ?[book:]sections
▷ ?[book:]chapters

These commands show tables of content for all available, respectively the matching books.

6▷ ?
▷ ?&

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 2.3.1 below.

2.3 Changing the Help Viewer

Books of the GAP help system 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 well readable in this format.

- dvi
  The standard output format of \TeX. Only useful if \TeX is installed on your system. Can be used for printing a help book and onscreen reading. Some books include hyperlink information in this format which can be useful for onscreen reading.

- ps
  Postscript format. Can be printed on most systems and also be used with an onscreen viewer.

- pdf
  Adobe's pdf-format. Can also be used for printing and onscreen reading on most current systems (with freely available software). Some books have hyperlink information included 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 also has the problem that complicated formulae may be not well readable since there is no syntax
Section 3. Changing the Help Viewer

for formulae in HTML. Some books use special symbol fonts for formulae and need an appropriate Web-browser for correct display.

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

```
1 ▶ SetHelpViewer( viewer1, viewer2, ... )
```

This command takes an arbitrary number of arguments which must be strings describing a viewer. The recognized viewer 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 2.2.6.

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 will probably put a call of `SetHelpViewer` into your `.gaprc` file (see 3.4).

"screen"

This is the default setting. The help is shown in text-format using the `Pager` command explained in the next section 2.4.1. (Hint: Some formatting procedures assume that your terminal displays at least 80 characters per line, if this is not the case some sections may look very bad. Furthermore the terminal (window) should use a fixed width font and we suggest to take one with ISO-8859-1 (also called latin1) encoding.

"netscape"

If a book is available in HTML-format this is shown using the (already running) `netscape` Web-browser. Note, that for some books the browser must be configured to use symbol fonts.

"lynx"

If a book is available in HTML-format this is shown using the text based `lynx` Web-browser inside the terminal running GAP. Formulae which use symbol fonts may be unreadable.

"internet config"

(for Apple Macintosh with MacOS) If a book is available in HTML-format this is shown in a Web browser. The web browser used is the program set to handle the `file` protocol in the program `Internet Config` (System 7 and 8) resp. the `Internet` control panel (System 9 and System X). For some browsers (e.g., Internet Explorer), you may have enter the GAP command `HELP_MAC_PROTOCOL := "file:";` for this to work correctly. If you wish to use the online html version of the manual, you may use `HELP_EXTERNAL_URL := "http://www.gap-system.org/";`. Note that `HELP_EXTERNAL_URL := "":` switches back to the local html files. It may be a good idea to put the relevant line in the `gap.rc` file (see 3.4).

"xdvi"

(on X-windows systems) If a book is available in dvi-format it is shown with the onscreen viewer program `xdvi`. (Of course, `xdvi` and `TeX` must be installed on your system.) This program doesn’t allow remote commands, so usually for each shown topic a new `xdvi` is launched. You can try to compile the program `GAPPATH/etc/xrmtcmd.c` and to put the executable `xrmtcmd` into your `PATH`. Then this viewer tries to reuse one running `xdvi` for each help book.

"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. (Hint: On many systems `xpdf` shows a very
bad display quality, this is due to a wrong or missing font configuration. One needs to set certain X-resources; for more details follow the Problems link at

\url{http://www.foolabs.com/xpdf/}

"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 doesn’t 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.

"less" or "more"

This is the same as "screen" but additionally the PAGER and PAGER_OPTIONS variables are set, see the next section 2.4 for more details.

Please, send ideas for further viewer commands to Gap-Trouble@dcs.st-and.ac.uk

### 2.4 The Pager Command

GAP contains a builtin pager which shows a text string which doesn’t 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.

```
1 \texttt{\textbf{ Pager( lines )}}
```

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 \texttt{less} or \texttt{more}. GAP assumes that this program has a command line option \texttt{+nr} which starts the display of the text with line number \texttt{nr}. Which pager is used can be controlled by setting the variable \texttt{PAGER}. The default setting is \texttt{PAGER := "builtin"}; which means that the internal pager is used.

On UNIX systems you probably want to set \texttt{PAGER := "less"}; or \texttt{PAGER := "more"}; you can do this for example in your \texttt{.gaprc} file. In that case you can also tell GAP a list of standard options for the external pager. These are specified as list of strings in the variable \texttt{PAGER_OPTIONS}.

Example:

```
PAGER := "less";
PAGER_OPTIONS := ["-f", "-r", ",-a", ",-i", ",-M", ",-j2"];
```

The argument \texttt{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 \texttt{.lines} as in (1) or (2) and optional further components

In case (3) currently the following additional components are used:

\texttt{.formatted}

\texttt{can be} \texttt{false} \texttt{or} \texttt{true}. If set to \texttt{true} the builtin pager tries to show the text exactly as it is given (avoiding GAPs automatic line breaking)

\texttt{.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).
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.

```gap
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," ")));;
```
This chapter informs about command line options for GAP under UNIX (see 3.1, 3.2), options and features of GAP on the Macintosh (see 3.3), the .gaprc file (see 3.4), completion files (see 3.5), the GAP compiler (see 3.6, 3.7, 3.8), and how to save and load a GAP workspace (see 3.10).

### 3.1 Command Line Options

When you start GAP under UNIX, 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 -q-g` is illegal, use `gap -q-g` instead. Some options require an argument, this must follow the option and be separated by a space, e.g., `gap -m 256k`, it is not correct to say `gap -m256k` instead.

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

As is described in Chapter 73 (see 73.4), usually you will not execute GAP directly. Instead you will call a shell script, with the name `gap`, which in turn executes GAP. This shell 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.

- **-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.

- **-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.

- **-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.

- **-e**
  tells GAP not to quit when receiving a `ctr-D` on an empty input line (see 6.3.1). This option should not be used when the input is a file or pipe.

- **-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.

- **-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.

\textbf{\textit{-x} length}

With this option you can tell \texttt{GAP} how long lines are. \texttt{GAP} uses this value to decide when to split long lines. After starting \texttt{GAP} you may use \texttt{SizeScreen} (see 6.11.1) 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 ASCII terminal. 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.

\textbf{\textit{-y} length}

With this option you can tell \texttt{GAP} how many lines your screen has. \texttt{GAP} uses this value to decide after how many lines of on-line help it should wait. After starting \texttt{GAP} you may use \texttt{SizeScreen} (see 6.11.1) 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 ASCII terminal. 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.

\textbf{\textit{-g}}

tells \texttt{GAP} to print a information message every time a full garbage collection is performed.

\begin{verbatim}
#G FULL 44580/2479kb live 57304/4392kb dead 734/4096kb free
\end{verbatim}

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 KBytes from a total allocated memory of 4096 KBytes are available afterwards.

\textbf{\textit{-g} \textit{-g}}

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

\begin{verbatim}
#G PART 9405/961kb+live 7525/1324kb+dead 2541/4096kb free
\end{verbatim}

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

\textbf{\textit{-m} memory}

tells \texttt{GAP} to allocate \texttt{memory} bytes at startup time. If the last character of \texttt{memory} is \texttt{k} or \texttt{K} it is taken as KBytes, if the last character is \texttt{m} or \texttt{M} \texttt{memory} is taken as MBytes and if it is ‘g’ or ‘G’ it is taken as Gigabytes.

Under UNIX the default amount of memory allocated by \texttt{GAP} is 24 MBytes. The amount of memory should be large enough so that computations do not require too many garbage collections. On the other hand, if \texttt{GAP} allocates more virtual memory than is physically available, it will spend most of the time paging.

\textbf{\textit{-o} memory}

tells \texttt{GAP} to allocate at most \texttt{memory} bytes. If the last character of \texttt{memory} is \texttt{k} or \texttt{K} it is taken as KBytes, if the last character is \texttt{m} or \texttt{M} \texttt{memory} is taken as MBytes and if it is ‘g’ or ‘G’ it is taken as Gigabytes.

Under UNIX the default amount is 256 MBytes. If more than this amount is required during the \texttt{GAP} session, \texttt{GAP} prints an error messages and enters a break loop.

\textbf{\textit{-K} memory}

is like \textit{-o} above. But while the latter actually allocates more memory if the system allows it and then prints a warning inside a break loop the \textit{-K} options tells \texttt{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.

-1 path_list

can be used to modify GAP’s list of root directories (see 9.2). Before the option -l is used for the first time, the only root directory is ./, i.e., GAP has only one root directory which is the current directory. 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.

path_list should be a list of directories separated by semicolons. No whitespace is permitted before or after a semicolon. Each directory name should end with a pathname separator, i.e., /, but GAP will silently add one if it is missing. 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 viewed in the variable GAP_ROOT_PATHS.

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 init.g it will print the following warning

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.

-r

The option -r tells GAP not to read the user supplied ~/.gaprc files.

-L filename

The option -L tells GAP to load a saved workspace. See section 3.10.

-R filename ...

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

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 14 filenames on the command line. If a file cannot be opened GAP will print an error message and will abort.

3.2 Advanced Features of GAP

The following options are in general not needed for the normal operation of GAP. They are mostly used for debugging.

-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.
If the last character of memory is \text{k} or \text{K} it is taken as KBytes and if the last character is \text{m} or \text{M} memory is taken as MBytes.

\text{-A}
By default, some GAP packages (see 74) are loaded, if present, into the GAP session when it starts. This option disables (actually toggles) this behaviour, which can be useful for debugging or testing.

\text{-B architecture}
Executable binary files that form part of GAP or of a GAP package are kept in a subdirectory of the \text{bin} directory with in 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 \text{-B} option.

\text{-D}
The -D option tells GAP to print short messages when it is reading or completing files or loading modules. The message,

\text{#I READ_GAP_ROOT: loading 'lib/kernel.g' as GAP file}

 tells you that GAP has started to read the library file \text{lib/kernel.g}.

\text{#I READ_GAP_ROOT: loading 'lib/kernel.g' statically}
tells you that GAP has used the compiled version of the library file \text{lib/kernel.g}. This compiled module was statically linked to the GAP kernel at the time the kernel was created.

\text{#I READ_GAP_ROOT: loading 'lib/kernel.g' dynamically}
tells you that GAP has loaded the compiled version of the library file \text{lib/kernel.g}. This compiled module was dynamically loaded to the GAP kernel at runtime from a corresponding .so file.

\text{#I completing 'lib/domain.gd'}
tells you that GAP has completed the file \text{lib/domain.gd}. See 3.5 for more information about completion of files.

\text{-M}
tells GAP not to check for, nor to use, compiled versions of library files.

\text{-N}
tells GAP not to check for, nor to use, completion files, see 3.5.

\text{-O}
enables a GAP 3 compatibility mode, in which (for instance) the values \text{false} and \text{fail} are identified. Use of this mode is not recommended other than as a transitional step in porting GAP 3 code to GAP 4, because the GAP 4 library may not work reliably in this mode.

\text{-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.

\text{-X}
tells GAP to do a consistency check of the library file and the corresponding completion file when reading the completion file.

\text{-Y}
tells GAP to do a consistency check of the library file and the corresponding completion file when completing the library file.

\text{-i filename}
changes the name of the init file from the default \text{init.g} to \text{filename}.

Additional options, \text{-C}, \text{-U}, \text{-P}, \text{-W} and \text{-z} are used internally in the GAP compiler and/or on specific operating systems.
3.3 Running GAP under MacOS

This section describes the features of GAP for MacOS that differ from those described earlier in this chapter. Since you cannot enter command line options directly when you launch the GAP application on a Macintosh, another mechanism is being used: Hold down any of the command (apple), option, control or shift keys or space bar when launching the GAP application, e.g., by double-clicking on its icon. Please note that some keys have side effects (e.g., pressing the option key usually closes Findere windows), and that System X behaves slightly differently from other systems.

A dialog box will open, into which you can enter the desired GAP command line options, as described in 3.1. For example, if you want GAP to start with a workspace of 32 megabytes, the dialog box should contain the following text:

\[-m \text{32m}\]

Note that the dialog box may already contain settings which you have previously saved. The OK button accepts the command line for the current GAP session, and the Save button can be used to save these options for subsequent GAP sessions. The command line options will be saved in a text file called GAP options in the Preferences folder in the system folder. You may also modify the file GAP options directly; note that changes only take effect the next time you launch GAP.

There are three additional command line option on the Mac.

\[-z \text{n}\]

sets the time between checks for events (keystrokes, mouse clicks etc.) to \(n/60\) second. Lower values make GAP more responsive but computations are somewhat slower. A value greater than 60 is not recommended, the default value for \(n\) is 6.

\[-p \text{m}\]

sets the amount of memory required for printing. The reason is that printer drivers may require quite a bit of memory, and may even crash if not enough is found. To prevent this, GAP will not print unless at least the specified amount of memory is available. The default value is 64 Kilobytes, which is enough for the Apple LaserWriter printer driver. Setting the printing memory to 0 disables printing altogether.

\[-w \text{m}\]

sets the size of the log window to \(m\) bytes. This means that if the text in the log window exceeds this amount, then lines at the beginning of the log are deleted. The default value is 32 Kilobytes.

The following command line options work differently on the Mac.

\[-a\]

On the Mac, the \(-a\) option has a different meaning from the one described in 3.2. On the Mac, it must be used to reserve memory for loading dynamic libraries into GAP. See 3.6 for details about dynamic libraries (and note that the PPC version of GAP for MacOS can use dynamic libraries).

\[-f, -n\]

The \(-f\) and \(-n\) command line options do not have any effect on the Mac.

\[-e\]

The \(-e\) command line option enables \(ctr-D\).

\[-o\]

The \(-o\) command line option should not normally be used on the Mac. The value set by the \(-o\) option is only used if it is lower than the size of the workspace that would normally be available for GAP.
The file called `.gaprc` on UNIX systems (see 3.4) is called `gap.rc` on the Mac; it must be in the same folder as the GAP application.

All interaction between GAP and you takes place via the GAP log window: this is where GAP prints its messages and waits for your input. The amount of text in this window is limited (see the `-W` command line option above), so don’t be surprised if old GAP messages are deleted from the beginning of the text when this limit is reached. The reason for deleting old lines is that otherwise GAP may run out of memory just because of the messages it has printed.

GAP for the Mac now remembers the font and text size (which can be set choosing Format... in the Edit menu) as well as the window position of the GAP log window from one session to the next.

Almost all of the GAP editing keys described in Section 6.8 work on the Mac. In addition, GAP for MacOS also supports the usual editing keys on the Mac, such as Copy and Paste, Undo, arrow keys (also with shift, option and command). Note that you can also move forward and backward in the command line history by pressing `ctrl`-`arrow down` and `ctrl`-`arrow up`.

Note that Quit in GAP’s file menu works differently from the quit GAP command (see 6.3.1): Quit in the file menu always quits the GAP application, it cannot be used to quit from a break loop.

GAP for MacOS also contains a simple built-in text editor, which is mainly intended to create GAP files. New, Open..., Save and Close from the File menu work in the usual way.

The Read... and LogTo commands in the File menu work basically like the corresponding GAP commands (see 9.7). The only difference is that GAP will prompt you for the file with a standard Mac file opening dialog, so you do not have to enter the path name yourself. (You will see the file’s path name in the log window afterwards). Note that if a file you want to read is open in GAP’s built-in editor, then GAP will read the file from the edit window, not from the disk.

If you press the shift key while choosing Read... from the File menu, the menu item will change to Reread... which will then use the GAP command Reread (see 9.7.13) to read the chosen file.

The Read... command in the File menu changes to Read if the front window belongs to a file in GAP’s built-in editor – choosing Read then makes GAP read that file – and while the file is being read, the File menu item changes to Abort Read. You cannot close the file’s window while it is being read by GAP – choose Abort Read first.

Garbage collection messages, which are switched on and off by the `-g` command line option (see 3.1) can also be switched on and off by choosing Show garbage collections and Show partial collections from the Window menu.

If Always scroll to printout is selected in the Window menu, GAP will always scroll the GAP log window so that you can see what GAP is currently printing. Otherwise, the GAP log window is only scrolled to the current print position when GAP prints its prompt and waits for you to enter a command. Note that you may see text lines disappear even if Always scroll to printout is off – this happens if you are viewing the text at the beginning of the log window and some lines are just being deleted from the log because it has exceeded its 32000 character limit.

The contents of the Help menu should be quite self-explanatory. Note that, unlike in GAP 3 for the Mac, the online help is not displayed in a separate window, nor is the online help available while GAP is computing.

Holding down the Command (Apple) key while selecting text does the same as selecting the text and choosing Find selection in table of contents from the Help menu, holding down both Command and Option keys while selecting tries to find the selection in the index.

When you want to refer to files or folders in GAP (for example in the Read, PrintTo, AppendTo, LogTo commands), or have to specify files or folders for a command line option, these files must be identified by UNIX style path names. (Presently, GAP for MacOS also supports Mac path names, but this may change in the future.)
Users who are familiar with UNIX path names may skip the rest of this section, noting that the working
directory (i.e., folder) is the one in which the GAP application resides, and that file names on the Mac are
not case sensitive.

Paths are strings used to describe where a file is stored on a hard disk. There are two ways for specifying
UNIX path names: absolute and relative paths. An absolute path starts with a /, then the name of the
disk where the file is located, another /, then a list of folders, each containing the next one, separated by
/, and finally the name of the file, which resides in the last folder in the list. For instance, if your hard disk
is called My HD, and your file program.g resides (or should be created) in the folder programs in the folder
documents on My HD, the absolute path name to that file is

/My HD/documents/programs/program.g

Relative path names work similarly, except that the starting point is not a disk but the folder in which the
GAP application program resides. Relative path names are formed like absolute ones, except that they do
not start with a /. Thus, if you want to access the file temp.g in the folder tmp in the GAP folder, you may
use the following path name: tmp/temp.g. It is also possible to move upward to a parent folder: suppose that
the folder containing GAP is called applications, which contains a folder editor which in turn contains
the file 'program.g', then you could access this file by the path ../editor/program.g. The path ../ refers
to the GAP folder itself, and ../ refers to “the folder above”.

Note also that GAP for the Mac follows (resolves) aliases to folders and files.

### 3.4 The .gaprc file

When you start GAP, it looks for the file with the name .gaprc in your home directory (on UNIX systems).
On a Macintosh the equivalent to the .gaprc file is gap.rc, and for it to be read it must be in the same
folder as the GAP application. If such a file is found it is read after libname/init.g, but before any of the
files mentioned on the command line are read. You can use this file for your private customizations. 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.

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

If you have already a .gaprc file for GAP 3, its settings might not be compatible with GAP 4. In this case it
has to be removed. On UNIX Systems the following .gaprc file can be used to load alternatively a .gap3rc
or a .gap4rc file from your home directory.

```gap
if IsBound(Permutations) then
  # GAP 3
  Exec("echo \"READ(\\"'pwd "'/.gap3rc\\");\" > /tmp/jJj");
else
  # GAP 4
  Exec("echo \"READ(\\"'pwd "'/.gap4rc\\");\" > /tmp/jJj");
fi;
Read("/tmp/jJj");
```
3.5 Completion Files

The standard distribution of GAP already contains completion files so in general you do not need to create these files by yourself.

When starting, GAP reads in the whole library. As this takes some time, library files are normally condensed into completion files. These completion files contain the basic skeleton of the library but not the function bodies. When a function body is required, for example because you want to execute the corresponding function, the library file containing the function body is completed.

Completion files reduce the startup time of GAP drastically. However, this technique also means that the information stored in the completion files and the library must be consistent. If you change a library file without recreating the completion files disaster is bound to happen.

Bugfixes distributed for GAP will also update the completion files. Therefore you only need to update them if you have changed the library by yourself.

However, if you are modifying a library file a more convenient way is to use the -X option (see 3.1) that allows you (in most cases) to use the completion files for the unchanged parts of library files and avoids using the completion files for the changed parts. After you have finished modifying the library files you can recreate the completion files using:

\begin{verbatim}
CreateCompletionFiles()

CreateCompletionFiles(path)
\end{verbatim}

To create completion files you must have write permissions to path, which defaults to the first root directory. Start GAP with the -N option (to suppress the reading of any existing completion files), then execute the command CreateCompletionFiles(path);, where path is a string giving a path to the home directory of GAP (the directory containing the lib directory).

This produces, in addition to lots of informational output, the completion files.

\begin{verbatim}
$ gap4 -N
gap> CreateCompletionFiles();
#I converting "gap4/lib/read2.g" to "gap4/lib/read2.co"
#I parsing "gap4/lib/process.gd"
#I parsing "gap4/lib/listcoef.gi"
...
\end{verbatim}

3.6 The Compiler

The GAP compiler GAC creates C code from GAP code and then calls the system’s C compiler to produce machine code from it. This can result in a speedup (see section 3.7 for more details).

To use the compiler to produce dynamically loadable modules, call it with the -d option:

\begin{verbatim}
M193 /home/ahulpke > gap4/bin/i386-ibm-linux-gcc2/gac -d test.g
gap4/bin/i386-ibm-linux-gcc2/gac -C /tmp/5827_test.c test.g Init_Dynamic
gcc -fpic -ansi -Wall -O2 -o /tmp/5827_test.o -I
gap4/bin/i386-ibm-linux-gcc2/.../src -c /tmp/5827_test.c
ld -Bshareable -x -o test.so /tmp/5827_test.o
rm -f /tmp/5827_test.o
rm -f /tmp/5827_test.c
\end{verbatim}

This produces a file file.so.
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 3.9).

```gap
gap> LoadDynamicModule("./test.so");
gap> CrcFile("test.g");
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);
```

If you want to see or modify the intermediate C code, you can also instruct the compiler to produce only the C files by using the option `-C` instead of `-d`.

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.

### 3.7 Suitability for Compilation

Typically algorithms spend large parts of their runtime only in small parts of the code. The design of GAP reflects this situation with kernel methods for many time critical calculations such as matrix or permutation arithmetic.

Compiling an algorithm whose time critical parts are already in the kernel of course will give disappointing results: Compilation will only speed up the parts that are not already in the kernel and if they make up a small part of the runtime, the overall gain is small.

Routines that benefit from compilation are those which do extensive operations with basic data types, such as lists or small integers.

### 3.8 Compiling Library Code

The most tempting code to compile is probably the library. This section describes the mechanism used to make GAP recognize compiled versions of library files. Note however that there is no point in compiling the whole library as typically only few functions benefit from compilation as described in Section 3.7.

All files that come with GAP are read using the internal function `READ_GAP_ROOT`. This function then checks whether a compiled version of the file exists and if its CRC number (see 3.9) matches the file. If it does, the compiled version is loaded. Otherwise the file is read. You can start GAP with the `-D -N` option to see information printed about this process.

To make GAP find the compiled versions, they must be put in the `bin/systemname/compiled` directory (systemname is the name you gave for compilation, for example `i386-ibm-linux-gcc2`). They have to be called according to the following scheme: Suppose the file is `humpty/dumpty.gi` in the GAP home directory. Then the compiled version will be `bin/systemname/compiled/humpty/gi/dumpty.so`. That is, the directory hierarchy is mirrored under the `compiled` directory. A further directory level is added for the suffix of the file, and the suffix of the compiled version of the file is set to `.so` (as produced by the compiler).

For example we show how to compile the `combinat.gi` file on a Linux machine. Suppose we are in the home directory of the gap distribution.
bin/gac -d lib/combinat.gi

creates a file combinat.so. We now put it in the right place, creating also the necessary directories:

mkdir bin/i386-ibm-linux-gcc2/compiled
mkdir bin/i386-ibm-linux-gcc2/compiled/lib
mkdir bin/i386-ibm-linux-gcc2/compiled/lib/gi
mv combinat.so bin/i386-ibm-linux-gcc2/compiled/lib/gi

If you now start GAP and look, for example, at the function Binomial, defined in combinat.gi, you see it is indeed compiled:

gap> Print(Binomial);
function ( <<arg-1>>, <<arg-2>> )
  <<compiled code>>
end

The command line option -M disables the loading of compiled modules and always reads code from the library.

3.9 CRC Numbers

CRC (cyclic redundancy check) numbers provide a certain method of doing checksums. They are used by GAP to check whether files have changed. Whenever files are “condensed” – for example for completion files (see Section 3.5) or when compiling files (see Section 3.6) – such a checksum is computed implicitly and stored within the condensed file.

When reading a condensed version of the file instead of the original one, the CRC checksum, which is computed via CrcFile (see 9.7.11), can be used to check whether the original has been changed in the meantime, e.g.

gap> CrcFile("lib/morpheus.gi");
2705743645

3.10 Saving and Loading a Workspace

\textbf{SaveWorkspace( filename )}

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 \texttt{SaveWorkspace} was called.

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

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

\textbf{\texttt{-L filename}}

A saved workspace can be loaded by starting GAP with the option \texttt{-L} (see 3.1). This will start GAP and load the workspace.
Under UNIX, it is possible to compress savefiles using `gzip`. Compression typically reduces the size of a workspace by a factor 3 or 4. If GAP is started with a compressed savefile (omit the `.gz` ending), it will try to locate `gzip` on the system and uncompress the file automatically while reading it.

```
you@unix> gap -L savefile
gap> a;
1
```

We cannot guarantee that saved workspaces are portable between different system architectures or over different versions of GAP or its library.

If compiled modules had been loaded into GAP before the workspace was saved, they will be loaded into the new GAP session during the workspace loading process. If they are not available then the load will fail. Additional compiled modules will not be used, even if they are available, although they may be loaded later using `Reread` (see 9.7.13). `SaveWorkspace` may sometimes produce warning messages, as in

```
gap> SaveWorkspace("b5");
#W bad bag id 4 found, 0 saved
#W bad bag id 20 found, 0 saved
true
```

A small number of such messages can probably be ignored (they arise because the garbage collector may not always collect all dead objects, and dead objects may contain data that `SaveWorkspace` does not know how to process).
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.

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

\[ 1 + 2 \times 3; \]

GAP takes this external representation and creates a tree-like internal representation, which we can picture as follows
This expression is then evaluated. To do this GAP first evaluates the right subexpression $2 \times 3$. 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 $\times$ 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, 4.6, and 4.24, the semantic rules are given in sections 4.7, 4.8, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.22.1, and the chapters describing the individual data types.

4.2 Lexical Structure

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

Digits, uppercase and lowercase letters, space, tab, newline, and the special characters

```
+ - * / \  \\  \
1 * / \  \\  \
2 \ /  \\
```

Other characters will be signalled as illegal. Inside strings (see section 4.3 and chapter 26) 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 consisting entirely of lowercase letters (see 4.5). An identifier is a sequence of letters and digits that contains at least one letter and is not a keyword (see 4.6). An integer is a sequence of digits (see 14). A string is a sequence of arbitrary characters enclosed in double quotes (see 26).

Operator and delimiter symbols are

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

Note also that during the process of scanning all whitespace is removed (see 4.4).
Section 5. Keywords

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 keywords are

```
and  do    elif else    end fi
for  function if  in local mod
not  od  or  repeat return then
until while  quit  QUIT  break rec
continue
```

Note that (almost) all keywords are written in lowercase and that they are case sensitive. For example only 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: A number of tokens that appear to be normal identifiers representing functions or literals of various kinds 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:

```
false  true  IsBound  Unbind  TryNextMethod
Info  Assert  SaveWorkspace  fail
```
4.6 Identifiers

An identifier is used to refer to a variable (see 4.8). An identifier consists of letters, digits, and underscores, and must contain at least one letter or underscore. An identifier is terminated by the first character not in this class. Examples of valid identifiers are

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

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.

4.7 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 4.13). Expressions appear as right hand sides of assignments (see 4.14), as actual arguments in function calls (see 4.10), 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, 4.10, 14, 40, 26, 4.22.1s, 21, 27), 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 4.11 and 28.5). The arithmetic operators are +, -, *, /, mod, and ^ (see 4.12). The logical operators are not, and, and or (see 20.3).

```
  gap> 2 * 2;  # a very simple expression with value
              4
  gap> 2 * 2 + 9 = Fibonacci(7) and Fibonacci(13) in Primes;
  true      # a more complex expression
```

For the precedence of operators, see 4.11.
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 4.14). 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 4.10).

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 4.22.1). 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.

```gap
  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 programming languages like PASCAL.

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 PASCAL has 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 PASCAL requires for every variable.

In GAP on the other hand each variable just points to a value, and different variables can share the same value.
**Chapter 4. The Programming Language**

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 in the user’s code. Certain special facilities are provided for manipulating these 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.

```plaintext
gap> xx := 17;
17
gap> IsReadOnlyGlobal("xx");
false
gap> xx := 15;
15
gap> MakeReadOnlyGlobal("xx");
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
```

A group of functions are also supplied for accessing and altering the values assigned to global variables. Use of these functions differs from the use of assignment, `Unbind` and `IsBound` 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.

4. \textbf{ValueGlobal( name )} 
returns the value currently bound to the global variable named by the string \textit{name}. An error is raised if no value is currently bound.

5. \textbf{IsBoundGlobal( name )} 
returns \texttt{true} if a value currently bound to the global variable named by the string \textit{name} and \texttt{false} otherwise.

6. \textbf{UnbindGlobal( name )} 
removes any value currently bound to the global variable named by the string \textit{name}. Nothing is returned. A warning is given if \textit{name} was not bound. The global variable named by \textit{name} must be writable, otherwise an error is raised.

7. \textbf{BindGlobal( name, val )} 
sets the global variable named by the string \textit{name} to the value \textit{val}, provided it is writable, and makes it read-only. If \textit{name} already has 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 Categories). Caution should be exercised in using these functions, especially \texttt{BindGlobal} and \texttt{UnbindGlobal} as unexpected changes in global variables can be very confusing for the user.

\begin{verbatim}
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
\end{verbatim}

Finally, there are a group of functions dealing with the \texttt{global namespace}.

8. \textbf{NamesGVars()}
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.

9. \textbf{NamesSystemGVars()}
This function returns an immutable sorted list of all the global variable names created by the GAP library when GAP was started.

10. \textbf{NamesUserGVars()}
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.
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 Function Calls

\[ \text{function-var( } \text{arg-expr, } \text{arg-expr, } \ldots \text{) } \]

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 27) 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-exps 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 4.22.1 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-exps 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 4.23) 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 4.15). 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.

\[ \text{gap> Fibonacci( 11 ); } \]
\[ \# \text{ a call to the function 'Fibonacci' with actual argument '11'} \]
\[ 89 \]

\[ \text{gap> RightCosets( G, Intersection( U, V ) );}; \]
\[ \# \text{ a call to the operation 'RightCosets'} \]
\[ \# \text{ where the second actual argument is another function call} \]
Section 11. Comparisons

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.

```gap
gap> Size( fpgrp : hard, tcselection := "external" );
# a call to 'Size' passing the options 'hard' (value 'true') and # 'tcselection' (value the string "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

```gap
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.

4.11 Comparisons

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. Note that any two objects can be compared, i.e., = and <> will never signal an error. 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.

< 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.

Only for the following kinds of objects, an ordering via < of objects in different families (see 13.1) is supported. Rationals (see 16.1.1) are smallest, next are cyclotomics (see 18.1.3), followed by finite field elements (see 57.1.1); finite field elements in different characteristics are compared via their characteristics, next are permutations (see 40), followed by the boolean values true, false, and fail (see 20), characters (such as 'a', see 26), and lists (see 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 \(a = b \in c = d\), you must use \((a = b) \in (c = d)\) instead. The comparison operators have higher precedence than the logical operators (see 20.3), but lower precedence than the arithmetic operators (see 4.12). Thus, for example, \((a * b) = c\) and \(d\) is interpreted as \(((a * b) = c)\) and \(d\).

\[
\text{gap> 2 * 2 + 9 = Fibonacci(7); } \quad \text{# a comparison where the left operand is an expression}
\]

For the underlying operations of the operators introduced above, see 30.11.

### 4.12 Arithmetic Operators

The arithmetic operators are \(+\), \(-\), \(*\), \(/\), \(\mod\), and \(^\cdot\). The meanings (semantics) of those operators generally depend on the types of the operands involved, and, except for \(\mod\), 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 \(0 \ldots \lfloor b \rfloor - 1\) 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 \(0 \ldots \lfloor b \rfloor - 1\) 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 gcd 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 \(\lfloor b \rfloor\) 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 \(-1\).
Section 13. Statements

* 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 \(^\) has the highest precedence, followed by the unary operators + and −, which are followed by the multiplicative operators \(*\), \(\text{/}\), and \(\text{mod}\), 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 illegal 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. \(*\), \(\text{/}\), \(\text{mod}\), +, 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 4.11 and 28.5) and the logical operators (see 20.3). Thus, for example, \(a * b = c\) and \(d\) is interpreted, \(((a * b) = c) \text{ and } d\).

\[
\text{gap}\gg 2 * 2 + 9; \# \text{ a very simple arithmetic expression}
\]
13

For other arithmetic operations, and for the underlying operations of the operators introduced above, see 30.12.

4.13 Statements

Assignments (see 4.14), Procedure calls (see 4.15), if statements (see 4.16), while (see 4.17), repeat (see 4.18) and for loops (see 4.19), and the return statement (see 4.23) 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.

\[
\text{gap}\gg \text{if } i < 0 \text{ then } k = 16/i; \text{ fi;}
\]
Syntax error: := expected
if i < 0 then k = 16/i; fi;

\[
\text{gap}\gg
\]

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. There is nothing like PASCAL’s BEGIN-END, instead 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.
4.14 Assignments

\[ \text{var} := \text{expr}; \]

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 27.2). 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.

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

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 6.3). 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.

\[
\text{gap} > \text{f1} := \text{function}( x ) \text{Print( "value: ", x, "\n" )}; \text{end;;} \\
\text{gap} > \text{f2} := \text{function}( x ) \text{return f1( x )}; \text{end;;} \\
\text{gap} > \text{f2}( 4 ); \\
\text{value: 4} \\
\text{Function Calls: <func> must return a value at} \\
\text{return f1( x );} \\
\text{<function>( <arguments> ) called from read-eval-loop} \\
\text{Entering break read-eval-print loop, you can 'quit;' to quit to outer loop,} \\
\text{or you can return a value for the result to continue} \\
\text{brk> return "hello";} \\
\"hello"
\]

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: x”), 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.

4.15 Procedure Calls

\[ \text{procedure-var}(); \] \\
\[ \text{procedure-var}( \text{arg-expr} [, \text{arg-expr}, \ldots ] ); \]

The procedure call has the effect of calling the procedure procedure-var. A procedure call is done exactly like a function call (see 4.10). 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.
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 which sorts a list and returns the corresponding permutation of the entries (see 21.18.3).

4.16 If

\begin{verbatim}
if bool-expr1 then statements1 \{ elif bool-expr2 then statements2 \}[ else statements3 ] fi;
\end{verbatim}

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

\begin{verbatim}
if expr1 then if expr2 then stats1 else stats2 fi; fi;
\end{verbatim}

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

\begin{verbatim}
if expr1 then if expr2 then stats1 fi; else stats2 fi;
\end{verbatim}

the else part belongs to the first if statement.

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

\begin{verbatim}
abs := if x > 0 then x; else -x; fi;
\end{verbatim}

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 6.3) 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.

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

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 4.18) 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 6.3) 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.

```gap
gap> i := 0;; s := 0;;
gap> while s <= 200 do
>   i := i + 1; s := s + i^2;
> od;
gap> s;
204 # sum of the first i squares larger than 200
```

A `while` loop may be left prematurely using `break`, see 4.20.

4.18 Repeat

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 4.17) 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 6.3) 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.

```gap
gap> i := 0;; s := 0;;
gap> repeat
>   i := i + 1; s := s + i^2;
> until s > 200;
gap> s;
204 # sum of the first i squares larger than 200
```

A `repeat` loop may be left prematurely using `break`, see 4.20.
Section 19. For

4.19 For

```plaintext
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.

```plaintext
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).

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

corresponds to the more common

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

in other programming languages.

```plaintext
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.

```plaintext
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.
Chapter 4. The Programming Language

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

\textit{It is also possible to have a for-loop run over an iterator (see 28.7). In this case the for-loop is equivalent to}

\begin{verbatim}
while not IsDoneIterator(iterator) do
  variable := NextIterator(iterator)
  statements
od;
\end{verbatim}

\textit{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 \texttt{Iterator(object)}. If this is successful then the loop is taken to run over the iterator returned.}

gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
group := 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

\begin{verbatim}
for variable in domain do
\end{verbatim}

should thus normally be the same as

\begin{verbatim}
for variable in AsList(domain) do
\end{verbatim}

but may use much less storage, as the iterator may be more compact than a list of all the elements. See 28.7 for details about iterators.

A for loop may be left prematurely using \texttt{break}, see 4.20. 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.
4.20 Break

The statement `break;` causes an immediate exit from the innermost loop enclosing it. It is an error to use this statement other than inside a loop.

```gap
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;
  (1,4,3)(2,6,5)
gap> break;
A break statement can only appear inside a loop
```

4.21 Continue

The statement `continue;` causes the rest of the current iteration of the innermost loop enclosing it to be skipped. The next iteration begins immediately. It is an error to use this statement other than inside a loop.

```gap
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)
gap> continue;
A continue statement can only appear inside a loop
```

4.22 Function

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 4.10.

The following is an example of a function definition. It is a function to compute values of the Fibonacci sequence (see 17.3.1).
Chapter 4. The Programming Language

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 4.10), 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

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 $2 \cdot \text{fib}(n)-1$ steps to compute $\text{fib}(n)$, while the iterative version of $\text{fib}$ needs only $n-2$ steps. Both are not optimal however, the library function Fibonacci only needs about $\log(n)$ steps.

As noted in Section 4.10, 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 (see 21.16.1) might be encoded and demonstrates the “optional argument” scenario.

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;
>     end;
> end;;

gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
Section 22. Function

```gap
> 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.

```gap
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 \texttt{Sum} (see 21.20.23) 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 \( \text{arg} \) then \texttt{NumberArguments} (Function(\( f \)) returns \(-1\) (see 5.1.2).

The argument \( \text{arg} \) 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 \texttt{CallFuncList} (see 5.2.1) is provided for this purpose.

Also see Chapter 5.

2\> \texttt{arg-ident} -> \texttt{expr}

This is a shorthand for

```gap
function ( \text{arg-ident} ) \texttt{return} \texttt{expr}; \texttt{end}.
```

\texttt{arg-ident} must be a single identifier, i.e., it is not possible to write functions of several arguments this way. Also \texttt{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

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

When the definition of a function \texttt{fun1} is evaluated inside another function \texttt{fun2}, GAP binds all the identifiers inside the function \texttt{fun1} that are identifiers of an argument or a local of \texttt{fun2} to the corresponding variable.
This set of bindings is called the environment of the function \textit{fun1}. When \textit{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 \texttt{push} and \texttt{pop} in the record returned by \texttt{Stack} access the local variable \texttt{stack} of \texttt{Stack}. When \texttt{Stack} is called, a new variable for the identifier \texttt{stack} is created. When the function definitions of \texttt{push} and \texttt{pop} are then evaluated (as part of the \texttt{return} statement) each reference to \texttt{stack} is bound to this new variable. Note also that the two stacks \texttt{A} and \texttt{B} do not interfere, because each call of \texttt{Stack} creates a new variable for \texttt{stack}.

```gap
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
  );;
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 \texttt{GAP} is not very efficient.

### 4.23 Return

1. `return;`

In this form \texttt{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.

2. `return expr;`

In this form \texttt{return} terminates the call of the innermost function that is currently executing, and returns the value of the expression \texttt{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 6.3). \texttt{return;} has the effect that the computation continues where it was interrupted by an error or the user hitting \texttt{ctr-C}. \texttt{return expr;} can be used to continue execution after an error. What happens with the value \texttt{expr} depends on the particular error.

For examples of \texttt{return} statements, see the functions \texttt{fib} and \texttt{Stack} in Chapter 5.
4.24 The Syntax in BNF

This section contains the definition of the GAP syntax in Backus-Naur form. A few recent additions to the syntax may be missing from this definition. Also, the actual rules for identifier names implemented by the system, are somewhat more permissive than those given below (see section 4.6).

A BNF is a set of rules, whose left side is the name of a syntactical construct. Those names are enclosed in angle brackets and written in *italics*. The right side of each rule contains a possible form for that syntactic construct. Each right side may contain names of other syntactic constructs, again enclosed in angle brackets and written in *italics*, or character sequences that must occur literally; they are written in *typewriter style*.

Furthermore each righthand side can contain the following metasymbols written in *boldface*. If the right hand side contains forms separated by a pipe symbol (|) this means that one of the possible forms can occur. If a part of a form is enclosed in square brackets ([ ]) this means that this part is optional, i.e. might be present or missing. If part of the form is enclosed in curly braces ({ }) this means that the part may occur arbitrarily often, or possibly be missing.
Ident := a|...|z|A|...|Z| \{a|...|z|A|...|Z|0|...|9\} \\
Var := Ident \\
| Var . Ident \\
| Var . ( Expr ) \\
| Var [ Expr ] \\
| Var { Expr } \\
| Var ( [ Expr , Expr ] ) \\
| Var !. Ident \\
| Var !. ( Expr ) \\
| Var ![ Expr ] \\
| [ Expr , Expr ] . Expr \\
| List ' List ' \\
Record := rec([ Ident := Expr {, Ident := Expr } ] ) \\
Permutation := ( Expr {, Expr } ) { ( Expr {, Expr } ) } \\
Function := function ( [ Ident {, Ident } ] ) \\
| local Ident {, Ident } ; \\
| Statements \\
end \\
| Ident -> Expr \\
Char := 'any character' \\
String := " { any character } " \\
Int := 0|1|...|9 {0|1|...|9} \\
Atom := Int \\
| Var \\
| Permutation \\
| Char \\
| String \\
| Function \\
| List \\
| Record \\
| { not } true \\
| { not } false \\
Factor := {+/-} Atom [ " {+/-} Atom ] \\
Term := Factor { */mod Factor } \\
Arith := Term { +/- Term } \\
Rel := { not } Arith { =|= |<>|<|>|<=> in Arith } \\
And := Rel { and Rel } \\
Logical := And { or And } \\
Expr := Logical \\
| Var \\
Statement := Expr \\
| Var := Expr \\
| if Expr then Statements \\
| { elif Expr then Statements } \\
| else Statements fi \\
| for Var in Expr do Statements od \\
| while Expr do Statements od \\
| repeat Statements until Expr \\
| return [ Expr ] \\
| break \\
| quit \\
| QUIT \\
Statements := [ Statement ; ] \\
| ;
The section 4.22.1 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

1 ▶ NameFunction( func )

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, "unknown" is returned.

```gap
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);
"NAME_FUNC"
```

2 ▶ NumberArgumentsFunction( func )

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.

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

```gap
CallFuncList( func, args )
```
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`.

```gap
gap> CallFuncList(+, [6, 7]);
gap> \+(6, 7);
gap> 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 4.22.1); see the following code fragment.

```gap
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`.

```gap
gap> PrintNumberFromDigits := function ( arg )
>    CallFuncList( Print, arg );
>    Print( "\n" );
>    end;
function( arg ) ... end
gap> PrintNumberFromDigits( 1, 9, 7, 3, 2 );
gap> PrintDigits := function ( arg )
>    Print( arg );
>    Print( "\n" );
>    end;
function( arg ) ... end
gap> PrintDigits( 1, 9, 7, 3, 2 );
gap> PrintNumberFromDigits( 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` is often used as family predicate in `InstallMethod` (see 2.2.1 in “Programming in GAP”).

```gap
ReturnTrue( ... )
```
This function takes any number of arguments, and always returns `true`.

```gap
ReturnFalse( ... )
```
This function takes any number of arguments, and always returns `false`.

```gap
ReturnFail( ... )
```
This function takes any number of arguments, and always returns `fail`.

```gap
IdFunc( obj )
```
returns `obj`. 
5.4 Function Types

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

1. \texttt{IsFunction( \textit{obj} )}
   
is the category of functions.

2. \texttt{IsOperation( \textit{obj} )}
   
is the category of operations. Every operation is a function, but not vice versa.

3. \texttt{FunctionsFamily}
   
is the family of all functions.
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 (see 6.2) 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

```
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 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`.

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

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

### 6.2 View and Print

1. **View**
   ```
   View( obj1, obj2... )
   ```
   **F**

   `View` shows the objects `obj1`, `obj2`... etc. in a short form on the standard output. `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.

2. **Print**
   ```
   Print( obj1, obj2... )
   ```
   **F**

   Also `Print` shows the objects `obj1`, `obj2`... etc. on the standard output. The difference compared to `View` 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.
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 26). Some characters preceded by a backslash, such as `\n`, are processed specially (see chapter 26.1). `PrintTo` can be used to print to a file (see 9.7.3).

```gap
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
gap> g:= SmallGroup(12,5);;
<pc group with 3 generators>
gap> View(g);
<pc group with 3 generators>gap>
```

3 ▶ `ViewObj( obj )` O

4 ▶ `Display( obj )` O

The functions `View` and `Print` 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 2 in the programmer’s manual). The only exceptions are strings (see Chapter 26), for which the default `PrintObj` and `ViewObj` methods as well as the function `View` print also the enclosing doublequotes, whereas `Print` 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` it. If one wants to supply a “short form” for `View`, one can install additionally a method for `ViewObj`.

```gap
c:= CharacterTable( "A5" );
CharacterTable( "A5" )
gap> Display(c);

  2  2  2
  3  1  1
  5  1  1
```

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.
One can assign a string to an object that \texttt{Print} will use instead of the default used by \texttt{Print}, via \texttt{SetName} (see 12.8.1). Also, \texttt{Name} (see 12.8.2) returns the string previously assigned to an object for printing, via \texttt{SetName}. The following is an example in the current context of character tables.

\begin{verbatim}
gap> c := CharacterTable( "A5" ); CharacterTable( "A5" )
gap> SetName( c, "tblA5" ); c; tblA5
gap> Name( c ); "tblA5"
\end{verbatim}

6.3 Break Loops

When an error has occurred or when you interrupt \texttt{GAP} (usually by hitting \texttt{ctrl-C}) \texttt{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, \texttt{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 \texttt{gap>} to \texttt{brk>} to indicate that you are in a break loop.

\begin{verbatim}
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
\end{verbatim}

If errors occur within a break loop \texttt{GAP} enters another break loop at a \textit{deeper level}. This is indicated by a number appended to \texttt{brk}:

\begin{verbatim}
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>
\end{verbatim}

There are two ways to leave a break loop.
1. quit

The first is to *quit* the break loop. To do this you enter `quit`; or type the `eof` (end of file) 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.

```
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 quitting a break loop generally leaves the options stack with options you no longer want. The function `ResetOptionsStack` (see 8) removes all options on the options stack, and this is the sole intended purpose of this function.

2. return

The other way is to *return* from a break loop. To do this you type `return`; or `return expr;`. 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 21.1.2) or you may have to return a *value* (by typing: `return value;`) 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.

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

3. OnBreak

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` (see 6.3.5).

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

gap> 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* level 1 (see Chapter 7.4).
Note that for break loops entered by a call to `Error`, the lines after “Entering break read-eval-print loop ...” and before the `brk>` prompt can also be customised, namely by redefining `OnBreakMessage` (see 6.3.4).

Also, note that one can achieve the effect of changing `OnBreak` locally. As mentioned above, the default value of `OnBreak` is `Where`. Thus, a call to `Error` (see 6.5.1) generally gives a trace back up to five levels of calling functions. Conceivably, we might like to have a function like `Error` 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.)

```gap
gap> ErrorNoTraceBack := function(arg) # arg is a special variable that GAP
> # knows to treat as a list of arg'ths
> local SavedOnBreak, ENTBOnBreak;
> SavedOnBreak := OnBreak; # save the 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`.

```gap
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` with the same arguments to show the difference.

```gap
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.

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

When a break loop is entered by a call to `Error` (see 6.5.1) the message after the “Entering break read-eval-print loop ...” line is produced by the function `OnBreakMessage`, which just like `OnBreak` (see 6.3.3) is a user-configurable global variable that is a `function` with no arguments.
Chapter 6. Main Loop and Break Loop

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).

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

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

```
gap> Error("!

gap> Enter 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 6.3.3) can be adapted to achieve
this. The function CosetTableFromGensAndRels (see 45.5.5) is an example in the GAP code where the idea
is actually used.

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 (see 6.3.3) has
been set back to its default value.)

```
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 (see 6.4.1) may be necessary to access them.
Section 4. Variable Access in a Break Loop

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 3.2) to avoid loading compiled code.

(Function calls to Info 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.

6.4 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.

\[
\begin{align*}
&\text{DownEnv( } [nr] \text{ ) } \\
&\text{UpEnv( } [nr] \text{ ) }
\end{align*}
\]

DownEnv moves up \( nr \) steps in the environment and allows one to inspect variables on this level; if \( nr \) is negative it steps down in the environment again; \( nr \) defaults to 1 if not given. UpEnv acts similarly to DownEnv but in the reverse direction. (The names of DownEnv and UpEnv are the wrong way ’round; I guess it all depends on which direction defines is “up” – just use DownEnv and get used to that.)

```
gap> OnBreak := function() Where(0); end;; # eliminate back-tracing on
   # entry to break loop

gap> test:= function( n )
   > if n > 3 then Error( "!
   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();
```
Chapter 6. Main Loop and Break Loop

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.

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.

6.5 Error

Error( messages... )

Error signals an error from within a function. First the messages messages are printed, this is done exactly as if Print (see 6.2) were called with these arguments. Then a break loop (see 6.3) 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 ErrorCount

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.

An emergency way to leave GAP is to enter

```
QUIT
```

to any gap> or brk> or brk.nn> prompt.

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.

```
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.

If, when GAP is exiting due to a quit or end-of-file (ie 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

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.

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.
Chapter 6. Main Loop and Break Loop

`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.

`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. This history is limited to about 8000 characters.

`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-0` accepts this line and perform a `ctrl-N`.

Finally there are a few miscellaneous commands.

`ctrl-Y` 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 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. Note that Read 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 command may be used to leave GAP, start an editor, and read in the edited file automatically.

\begin{verbatim}
\texttt{Edit(filename)}
\end{verbatim}

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. You should set the GAP variable EDITOR to the name of the editor that you usually use, e.g., /usr/ucb/vi. This can for example be done in your .gaprc file (see the sections on operating system dependent features in Chapter 73).

6.10 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 and a subdirectory .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 in this .vim directory and copy 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:

\begin{verbatim}
if has("syntax")
  syntax on " Default to no syntax highlighting
endif

" For GAP files
augroup gap
  " Remove all gap autocmds
  au!
  autocmd BufRead,BufNewFile *.g,*.gi,*.gd source ~/.vim/gap.vim
  autocmd BufRead,BufNewFile *.g,*.gi,*.gd set filetype=gap comments=s:## \\
  ,m:## \\
  ,e:## \\
  " I'm using the external program 'par' for formatting 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
\end{verbatim}

See the headers of the two mentioned files for additional comments. Adjust details according to your personal taste.

Setup files for (x)emacs are contained in the etc/emacs subdirectory.
6.11 SizeScreen

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.

With one argument that is a list, SizeScreen sets the size of the screen; \( x \) is the length of each line, \( y \) is the number of lines. Either value \( x \) or \( y \) may be missing (i.e. left unbound), to leave this value unaffected. It returns the new values. Note that those parameters can also be set with the command line options \(-x \ x\) and \(-y \ y\) (see Section 3.1).

To check/change whether line breaking occurs for files and streams see 10.4.9 and 10.4.8.

The screen width must be between 20 and 256 characters (inclusive) and the depth at least 10 lines. Values outside this range will be adjusted to the nearest endpoint of the range.
This chapter describes some functions that are useful mainly for debugging and profiling purposes.

The sections 7.2.1 and 7.3 show how to get information about the methods chosen by the method selection mechanism (see chapter 2 in the programmer’s manual).

The final sections describe functions for collecting statistics about computations (see 7.6.1, 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:

```
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 6.3.1) and that the information about the incident is no longer accessible afterwards.

1► **ShowArguments( )**

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.

2► **ShowArgument( nr )**

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.

3► **ShowDetails( )**

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`. The last number is called `Choice` and is printed as an ordinal. So if exactly `k` methods were found but called `TryNextMethod` and there were no more methods it says `Choice: kth`. 

---

1. ShowArguments( )
2. ShowArgument( nr )
3. ShowDetails( )
Chapter 7. Debugging and Profiling Facilities

4 ▶ ShowMethods( )
▶ ShowMethods( verbosity )

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 ApplicableMethod, see 7.2.1). 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. With verbosity 3 not only all installed methods but also the reasons why they do not match are displayed.

5 ▶ ShowOtherMethods( )
▶ ShowOtherMethods( verbosity )

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 ApplicableMethod, see 7.2.1). 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 ApplicableMethod

1 ▶ ApplicableMethod( opr, args [, printlevel ] )
▶ ApplicableMethod( opr, args, printlevel, nr )
▶ ApplicableMethod( opr, args, printlevel, "all" )
▶ ApplicableMethodTypes( opr, args [, printlevel ] )
▶ ApplicableMethodTypes( opr, args, printlevel, nr )
▶ ApplicableMethodTypes( opr, args, printlevel, "all" )

In the first form, 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.

In the second form, if nr is a positive integer 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 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 \texttt{ApplicableMethod} is called then it returns either the desired result or the string \texttt{TRY\_NEXT\_METHOD}, which corresponds to a call to \texttt{TryNextMethod} in the method and means that the method selection would call the next applicable method.

\textbf{Note:} The kernel provides special treatment for the infix operations $\backslash+$, $\backslash-$, $\backslash*$, $\backslash/$, $\backslash^\wedge$, $\mod$ and $\in$. For some kernel objects (notably cyclotomic numbers, finite field elements and vectors thereof) it calls kernel methods circumventing the method selection mechanism. Therefore for these operations \texttt{ApplicableMethod} may return a method which is not the kernel method actually used.

\texttt{ApplicableMethod} does not work for constructors (for example \texttt{GeneralLinearGroupCons} is a constructor).

The function \texttt{ApplicableMethodTypes} takes the \texttt{types} or \texttt{filters} of the arguments as argument (if only filters are given of course family predicates cannot be tested).

### 7.3 Tracing Methods

1. \texttt{TraceMethods( oprs )}
   
   After the call of \texttt{TraceMethods} with a list \texttt{oprs} of operations, whenever a method of one of the operations in \texttt{oprs} is called the information string used in the installation of the method is printed.

2. \texttt{UntraceMethods( oprs )}
   
   turns the tracing off for all operations in \texttt{oprs}.

   ```
   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 ] );
   ```

3. \texttt{TraceImmediateMethods( flag )}
   
   If \texttt{flag} is true, tracing for all immediate methods is turned on. If \texttt{flag} is false it is turned off. (There is no facility to trace specific immediate methods.)

   ```
   gap> TraceImmediateMethods( true );
   gap> g:= Group( (1,2,3), (1,2) );;
   #I immediate: Size
   #I immediate: IsCommutative
   #I immediate: IsTrivial
   #I immediate: IsStabChainViaChainSubgroup
   #I immediate: IsChainTypeGroup
   gap> Size( g );
   #I immediate: IsNonTrivial
   #I immediate: Size
   #I immediate: IsStabChainViaChainSubgroup
   #I immediate: IsChainTypeGroup
   #I immediate: IsNonTrivial
   #I immediate: IsPerfectGroup
   #I immediate: IsEmpty
   6
   gap> TraceImmediateMethods( false );
   ```
Chapter 7. Debugging and Profiling Facilities

This example gives an explanation for the two calls of the “system getter” for Size. Namely, there are immediate methods that access the known size of the group. Note that the group \texttt{g} was known to be finitely generated already before the size was computed, the calls of the immediate method for IsFinitelyGeneratedGroup after the call of Size have other arguments than \texttt{g}.

7.4 Info Functions

The Info 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 (see 7.4.6) which initially has level 1.

1. \texttt{NewInfoClass( name )} creates a new info class with name \texttt{name}.

2. \texttt{DeclareInfoClass( name )} creates a new info class with name \texttt{name} and binds it to the global variable \texttt{name}. The variable must previously be writable, and is made readonly by this function.

3. \texttt{SetInfoLevel( infoclass, level )} sets the info level for \texttt{infoclass} to \texttt{level}.

4. \texttt{InfoLevel( infoclass )} returns the info level of \texttt{infoclass}.

5. \texttt{Info( infoclass, level, info [,moreinfo . . .] )}

   If the info level of \texttt{infoclass} is at least \texttt{level} the remaining arguments (\texttt{info} and possibly \texttt{moreinfo} and so on) are evaluated and viewed, preceded by ‘\#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.)

   \begin{verbatim}
   gap> InfoExample:=NewInfoClass("InfoExample");;
   gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
   #I one
   gap> InfoLevel(InfoExample,2);
   #I two
   #I three
   gap> Info(InfoExample,3,Length(Combinations([1..9999])));
   \end{verbatim}

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 \texttt{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. In this case, the message is triggered if the level of any of the classes is high enough.

```gap
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
```

is an info class to which general warnings are sent at level 1, which is its default level. More specialised warnings are Info-ed 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.

```gap
SetAssertionLevel( lev )

assigns the global assertion level to lev. By default it is zero.
```

```gap
AssertionLevel()

returns the current assertion level.
```

```gap
Assert( lev, cond )
Assert( lev, cond, message )
```

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

```gap
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

```gap
Runtime()

Runtime returns the time spent by GAP in milliseconds as an integer. This is usually the cpu time, i.e., not the wall clock time. Also time spent by subprocesses of GAP (see 11.1.1) is not counted.
```

```gap
time;

in the read-eval-print loop returns 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 during runtime.

1 ProfileOperations( [true/false] )

When called with argument true, this function starts profiling of all operations. Old profiling information is cleared. When called with false it stops profiling of all operations. Recorded information is still kept, so you can display it even after turning the profiling off.

When called without argument, profiling information for all profiled operations is displayed (see 7.7.8).

2 ProfileOperationsAndMethods( [true/false] )

When called with argument true, this function starts profiling of all operations and their methods. Old profiling information is cleared. When called with false it 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 called without argument, profiling information for all profiled operations and their methods is displayed (see 7.7.8).

3 ProfileMethods( ops )

starts profiling of the methods for all operations in ops.

4 UnprofileMethods( ops )

stops profiling of the methods for all operations in ops. Recorded information is still kept, so you can display it even after turning the profiling off.

5 ProfileFunctions( funcs )

turns profiling on for all function in funcs. You can use ProfileGlobalFunctions (see 7.7.7) to turn profiling on for all globally declared functions simultaneously.

6 UnprofileFunctions( funcs )

turns profiling off for all function in funcs. Recorded information is still kept, so you can display it even after turning the profiling off.

7 ProfileGlobalFunctions( true )

ProfileGlobalFunctions(false) turns on profiling for all functions that have been declared via DeclareGlobalFunction. A function call with the argument false turns it off again.

8 DisplayProfile( )

In the first form, DisplayProfile displays the profiling information for profiled operations, methods and functions. If an argument funcs is given, only profiling information for the functions in funcs is given. The information for a profiled function is only displayed if the number of calls to the function or the total time spent in the function exceeds a given threshold (see 7.7.9).

Profiling information is displayed in a list of lines for all functions (also operations and methods) which are profiled. For each function, “count” gives the number of times the function has been called. “self” gives the time spent in the function itself, “child” the time spent in profiled functions called from within this function. The list is sorted according to the total time spent, that is the sum “self”+“child”.

DisplayProfile( funcs )
Section 7. Profiling

This variable is a list \([cnt, time]\) of length two. \texttt{DisplayProfile} will only display lines for functions which are called at least \(cnt\) times or whose total time ("self"+"child") is at least \(time\). The default value of \texttt{PROFILETHRESHOLD} is \([10000, 30]\).

\texttt{ClearProfile()} clears all stored profiling information.

```
gap> ProfileOperationsAndMethods(true);
gap> ConjugacyClasses(PrimitiveGroup(24,1));;
gap> ProfileOperationsAndMethods(false);
gap> DisplayProfile();
```

```
[the following is excerpted from a much longer list]
1620  170  90  CycleStructurePerm: default method
1620  20  260  CycleStructurePerm
114658  280  0  Size: for a list that is a collection
287  20  290  Meth(CyclesOp)
287  0  310  CyclesOp
26  0  330  Size: for a conjugacy class
2219  50  380  Size
2  0  670  Meth(StabChainOp)
32  0  670  Meth(IsSubgroup)
48  10  670  Meth(IsSubset: for two collections (loop over the ele*
98  1  670  Meth(ClosureGroup)
1  0  710  Meth(DerivedSubgroup)
1  0  720  Meth(StabilizerOfExternalSet)
107  0  970  Meth(CentralizerOp)
926  0  970  Meth(CentralizerOp)
819  2100  2340  Meth(IN)
1  10  4890  Meth(StabChainOp,DerivedSubgroup);
1  0  5720  Meth(StabChainOp,DerivedSubgroup);
2  0  5740  Meth(StabChainOp,DerivedSubgroup);
552  10  830  Meth(StabChainOp,DerivedSubgroup);
6920  TOTAL
```

```
gap> DisplayProfile(StabChainOp,DerivedSubgroup); # only two functions
```

```
[the following is excerpted from a much longer list]
1  0  780  Meth(StabChainOp)
29  0  810  Meth(StabChainOp)
6920  OTHER
6920  TOTAL
```

Note that profiling (even the command \texttt{ProfileOperationsAndMethods(true)}) can take substantial time and \texttt{GAP} will perform much more slowly when profiling than when not.
DisplayCacheStats( ) displays statistics about the different caches used by the method selection.

ClearCacheStats( ) clears all statistics about the different caches used by the method selection.

### 7.8 Information about the version used

DisplayRevision( ) Displays the revision numbers of all loaded files from the library.

### 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.

ReadTest( name-file ) reads a test file. A test file starts with a line

```gap
START_TEST("arbitrary identifier string");
```

(Note that the `gap>` prompt is part of the line!) It continues by lines of GAP input and corresponding output. The input lines again start with the `gap>` prompt (or 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).

The test file stops with a line

```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 `combinat.tst`.

### 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 (6.3) allowing you to terminate the calculation, or enter `return`; to continue it.

```gap
dive := function(depth) if depth>1 then dive(depth-1); fi; return; end;
dive(100);
OnBreak := function() Where(1); end; # shorter traceback
dive(6000);
recursion depth trap (5000) at
dive( depth - 1 );
called from
dive( depth - 1 ); called from...
```
Section 10. Debugging Recursion

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 procedure

\[
\text{SetRecursionTrapInterval}( \text{interval} )
\]

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.

```gap
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 small 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 small 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
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:

```plaintext
PushOptions( rec( <stuff> ) );
DoSomething( <args> );
PopOptions();
```

This can be abbreviated, to `DoSomething( args : stuff );` with a small additional abbreviation of `stuff` permitted. See 4.10.2 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 a binary operation, or other function with special syntax.

At some time, an options predicate or something of the kind may be added to method selection.

An alternative to this system is the use of additional optional arguments in procedure calls. This is not felt to be adequate 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.

### 1 PushOptions( options record )

This function pushes a record of options onto the global option stack. Note that `PushOption(rec(opt := fail))` has the effect of resetting option \(opt\), since an option that has never been set has the value `fail` returned by `ValueOptions`.

Note that there is no check for misspelt or undefined options.

### 2 PopOptions( )

This function removes the top-most options record from the options stack.

### 3 ResetOptionsStack( )

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 6.3.1).
DisplayOptionsStack( )

This function prints a human-readable display of the complete options stack.

InfoOptions

This info class can be used to enable messages about options being changed (level 1) or accessed (level 2).

The example below shows simple manipulation of the Options Stack, first using PushOptions and PopOptions and then using the special function calling syntax.

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>
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 described in section 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:

```
LastError() F
```

LastError 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 Directory

When starting GAP it is possible to specify various directories as root directories. In GAP’s view of the world these directories are merged into one meta-directory. This directory is called GAP root directory in the following.

For example, if root1;root2;... is passed as argument to -l when GAP is started and GAP wants to locate a file lib/group.gd in the GAP root directory it will first check if the file exists in root1, if not, it checks root2, and so on.

This layout makes it possible to have one system-wide installation of GAP which is read-only but still allows users to modify individual files. Therefore instead of constructing an absolute path name to a file you should always use DirectoriesLibrary or DirectoriesPackageLibrary together with Filename to construct a filename for a file in the GAP root directory.

Example

Suppose that the system-wide installation lives in /usr/local/lib/gap4 and you want to modify the file lib/files.gd without disturbing the system installation.
In this case create a new directory `/home/myhome/gap` containing a subdirectory `lib` which contains the modified `lib/files.gd`.

The directory/file structure now looks like

```
/usr/local/lib/gap4/
/usr/local/lib/gap4/lib/
/usr/local/lib/gap4/lib/files.gd
/home/myhome/gap/
/home/myhome/gap/lib
/home/myhome/gap/lib/files.gd
```

If you start GAP using (under UNIX)

```
you@unix> gap -l '/home/myhome/gap;/usr/local/lib/gap4'
```

then the file `/home/myhome/gap/lib/files.gd` will be used whenever GAP references the file with filename `lib/files.gd` in the GAP root directory.

This setup also allows one to easily install new GAP packages or bugfixes even if no access to the system GAP installation is possible. Simply unpack the files into “/home/myhome/gap”.

### 9.3 Directories

1. **Directory( string )**
   
   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 `USER_HOME` is defined (this may depend on the operating system) then `Directory` understands a string with a leading ~ character for a path relative to the user’s home directory.

   Paths are otherwise taken relative to the current directory.

2. **DirectoryTemporary( hint )**

   returns a directory object in the category `IsDirectory` 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 either when a garbage collection collects the directory object or upon termination of the GAP job that created the directory. `hint` can be used by `DirectoryTemporary` to construct the name of the directory but `DirectoryTemporary` is free to use only a part of `hint` or even ignore it completely.

   If `DirectoryTemporary` is unable to create a new directory, `fail` is returned. In this case `LastSystemError` can be used to get information about the error.

3. **DirectoryCurrent( )**

   returns the directory object for the current directory.

4. **DirectoriesLibrary( )**

   returns the directory objects for the GAP library `lib` as a list. `lib` must be one of "lib" (the default), "grp", "prim", and so on. The string "" is also legal and with this argument `DirectoriesLibrary` returns the list of GAP root directories; the return value of `DirectoriesLibrary("")` differs from `GAP_ROOT_PATHS` in that the former is a list of directory objects and the latter a list of strings.

   The directory `lib` must exist in at least one of the root directories, otherwise `fail` is returned.
As the files in the GAP root directory (see 9.2) can be distributed into different directories in the file space a list of directories is returned. In order to find an existing file in the GAP root directory you should pass that list to Filename (see 9.4.1) as the first argument. In order to create a filename for a new file inside the 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 instead.

5 ▶ DirectoriesSystemPrograms( )

DirectoriesSystemPrograms returns the directory objects for the list of directories where the system programs reside as a list. Under UNIX this would usually represent $PATH.$

### 9.4 Filename

1 ▶ Filename( dir, name )

1 ▶ Filename( list-of-dirs, name )

If the first argument is a directory object \texttt{dir}, \texttt{Filename} returns the (system dependent) filename as a string for the file with name \texttt{name} in the directory \texttt{dir}. \texttt{Filename} returns the filename regardless of whether the directory contains a file with name \texttt{name} or not.

If the first argument is a list \texttt{list-of-dirs} (possibly of length 1) of directory objects, then \texttt{Filename} searches the directories in order, and returns the filename for the file \texttt{name} in the first directory which contains a file \texttt{name} or \texttt{fail} if no directory contains a file \texttt{name}.

**Examples**

In order to locate the system program \texttt{date} use \texttt{DirectoriesSystemPrograms} together with the second form of \texttt{Filename}.

```gap
gap> path := DirectoriesSystemPrograms();;
gap> date := Filename( path, "date" );
"/bin/date"
```

In order to locate the library file \texttt{files.gd} use \texttt{DirectoriesLibrary} together with the second form of \texttt{Filename}.

```gap
gap> path := DirectoriesLibrary();;
gap> Filename( path, "files.gd" );
"./lib/files.gd"
```

In order to construct filenames for new files in a temporary directory use \texttt{DirectoryTemporary} together with the first form of \texttt{Filename}.

```gap
gap> tmpdir := DirectoryTemporary();;
gap> Filename( [ tmpdir ], "file.new" );
fail
gap> Filename( tmpdir, "file.new" );
"/var/tmp/tmp.0.021738.0001/file.new"
```
Chapter 9. Files and Filenames

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 ctr-'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 \textit{false} one can use \texttt{LastSystemError} (see 9.1.1) to find out the reason (as provided by the operating system).

1) \texttt{IsExistingFile( name-file )}
   returns \textit{true} if a file with the filename \texttt{name-file} exists and can be seen by the GAP process. Otherwise \textit{false} is returned.

2) \texttt{IsReadableFile( name-file )}
   returns \textit{true} if a file with the filename \texttt{name-file} exists \textbf{and} the GAP process has read permissions for the file, or \textit{false} if this is not the case.

3) \texttt{IsWritableFile( name-file )}
   returns \textit{true} if a file with the filename \texttt{name-file} exists \textbf{and} the GAP process has write permissions for the file, or \textit{false} if this is not the case.

4) \texttt{IsExecutableFile( name-file )}
   returns \textit{true} if a file with the filename \texttt{name-file} exists \textbf{and} the GAP process has execute permissions for the file, or \textit{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.

5) \texttt{IsDirectoryPath( name-file )}
   returns \textit{true} if the file with the filename \texttt{name-file} exists \textbf{and} is a directory and \textit{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 \texttt{IsWritableFile} (see 9.6.3), resp. \texttt{IsExecutableFile} (see 9.6.4) to check such permissions).

\underline{Examples}

Note, in particular, how one may use \texttt{LastSystemError} (see 9.1.1) to discover the reason a file access function returns \textit{false}. 
Section 7. File Operations

9.7 File Operations

1  Read( name-file )
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 6.3). 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 ctr-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 (end-of-file) 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 operating system dependent maximum on the number of files that may be open simultaneously. Usually it is 15.

2  ReadAsFunction( name-file )
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

    local a;
    a := 10;
    return a*10;

Reading the file as a function will not affect a global variable a.
Chapter 9. Files and Filenames

```gap
gap> a := 1;
1
gap> ReadAsFunction("/tmp/example.g");

100

gap> a;
1
```

3 ▶ `PrintTo( name-file [, obj1 , ...] )`  
works like `Print` (see 6.2.2), 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` can be used to append to a file (see 9.7.4). There is an operating system dependent maximum on the number of output files that may be open simultaneously, usually this is 14.

4 ▶ `AppendTo( name-file [, obj1 , ...] )`  
works like `PrintTo` (see 9.7.3), except that the output does not overwrite the previous contents of the file, but is appended to the file.

5 ▶ `LogTo( 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` may also be used to log to a stream (see 10.4.5). 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.

6 ▶ `LogTo()`  
In this form `LogTo` stops logging to a file or stream.

7 ▶ `InputLogTo( 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` cannot be used at the same time while `InputLogTo` and `OutputLogTo` can. Note that `InputLogTo` will overwrite the previous contents of this file if it already existed.

8 ▶ `InputLogTo()`  
In this form `InputLogTo` stops logging to a file or stream.

9 ▶ `OutputLogTo( 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` cannot be used at the same time while `InputLogTo` and `OutputLogTo` can. Note that `OutputLogTo` will overwrite the previous contents of this file if it already existed.

10 ▶ `OutputLogTo()`  
In this form `OutputLogTo` stops logging to a file or stream.  
**Note** that one should be careful not to write to a logfile with `PrintTo` or `AppendTo`.

11 ▶ `CrcFile( name-file )`  
computes a checksum value for the file with filename `name-file` and returns this value as an integer. See Section 3.9 for an example. The function returns `fail` if a system error occurred, say, for example, if `name-file` does not exist. In this case the function `LastSystemError` (see 9.1.1) can be used to get information about the error.
Section 7. File Operations

RecallFile( name-file )

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 (see 9.1.1) can be used to get information about the error.

Reread( name-file )

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 (see 3.2.1 in “Programming in GAP”) and NewInfoClass (see 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 (see 2.2.1 in “Programming in GAP”) 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 3.6 and 3.10 for more information.
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 and AppendTo, 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, carriage-return under MacOS, 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.

Note that binary streams are not yet implemented.

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 described in Section 10.2.1. If creating a stream failed then LastSystemError (see 9.1.1) can be used to get information about the failure.

10.1 Categories for Streams and the StreamsFamily

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsStream( obj )</td>
<td>Streams are GAP objects and all open streams, input, output, text and binary, lie in this category.</td>
</tr>
<tr>
<td>IsClosedStream( obj )</td>
<td>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.</td>
</tr>
<tr>
<td>IsInputStream( obj )</td>
<td>All input streams lie in this category, and support input operations such as ReadByte (see 10.3)</td>
</tr>
<tr>
<td>IsInputTextStream( obj )</td>
<td>All text input streams lie in this category. They translate new-line characters read.</td>
</tr>
<tr>
<td>IsInputTextNone( obj )</td>
<td>It is convenient to use a category to distinguish dummy streams (see 10.9) from others. Other distinctions are usually made using representations.</td>
</tr>
</tbody>
</table>
Section 2. Operations applicable to All Streams

6 ▶ IsOutputStream(obj)
All output streams lie in this category and support basic operations such as WriteByte (see 10.4)

7 ▶ IsOutputTextStream(obj)
All text output streams lie in this category and translate new-line characters on output.

8 ▶ IsOutputTextNone(obj)
It is convenient to use a category to distinguish dummy streams (see 10.9) from others. Other distinctions are usually made using representations

9 ▶ StreamsFamily
All streams lie in the StreamsFamily

10.2 Operations applicable to All Streams

1 ▶ CloseStream(stream)
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.

2 ▶ FileDescriptorOfStream(stream)
returns the UNIX file descriptor of the underlying file. This is mainly useful for the UNIXSelect function call (see 10.2.3). This is as of now only available on UNIX-like operating systems and only for streams to local processes and local files.

3 ▶ UNIXSelect(inlist, outlist, exclist, timeoutsec, timeoutusec)
makes the UNIX C-library function select accessible from GAP for streams. The functionality is as described in the man page (see man select). The first three arguments must be lists containing UNIX file descriptors (integers) for streams. They can be obtained via FileDescriptorOfStream (see 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, ReadAsFunction and ReadTest can also be used to read GAP input from a stream. The input is immediately parsed and executed. When reading from a stream \textit{str}, the GAP kernel generates calls to \texttt{ReadLine(str)} to supply text to the parser.

Three further operations: ReadByte, ReadLine and ReadAll, 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.

\begin{itemize}
  \item \texttt{Read( input-text-stream )} \hfill O
    \begin{itemize}
      \item reads the input-text-stream as input until \texttt{end-of-stream} occurs. See 9.7 for details.
    \end{itemize}
  \item \texttt{ReadAsFunction( input-text-stream )} \hfill O
    \begin{itemize}
      \item reads the input-text-stream as function and returns this function. See 9.7 for details.
    \end{itemize}
  \item \texttt{ReadTest( input-text-stream )} \hfill O
    \begin{itemize}
      \item reads the input-text-stream as test input until \texttt{end-of-stream} occurs. See 9.7 for details.
    \end{itemize}
\end{itemize}

\textbf{Example}

\begin{verbatim}
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
\end{verbatim}

\begin{itemize}
  \item \texttt{ReadByte( input-stream )} \hfill O
    \begin{itemize}
      \item \texttt{ReadByte} returns one character (returned as integer) from the input stream \textit{stream-in}. \texttt{ReadByte} returns \texttt{fail} if there is no character available, in particular if it is at the end of a file.
      \item If \textit{stream-in} is the input stream of a input/output process, \texttt{ReadByte} may also return \texttt{fail} if no byte is currently available.
      \item \texttt{ReadByte} is the basic operation for input streams. If a \texttt{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).
      \item \texttt{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.
    \end{itemize}
  \item \texttt{ReadLine( input-stream )} \hfill O
    \begin{itemize}
      \item \texttt{ReadLine} returns one line (returned as string \texttt{with} the newline) from the input stream \textit{input-stream}. \texttt{ReadLine} reads in the input until a newline is read or the end-of-stream is encountered.
    \end{itemize}
\end{itemize}
If \texttt{input-stream} is the input stream of a input/output process, \texttt{ReadLine} may also return \texttt{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 \texttt{ReadLine} which simply calls \texttt{ReadByte} repeatedly. This is only safe for streams that cannot block. The kernel uses calls to \texttt{ReadLine} to supply input to the parser when reading from a stream.

6 ▶ \texttt{ReadAll( input-stream )}

\texttt{ReadAll} returns all characters as string from the input stream \texttt{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, \texttt{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 \texttt{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 \texttt{ReadAll} which simply calls \texttt{ReadLine} repeatedly. This is only really safe for streams which cannot block. Other streams should install a method for \texttt{ReadAll}

Example

gap> i := InputTextString( "1Hallo\n\n\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\n\n\nYou\n1"

7 ▶ \texttt{IsEndOfStream( input-stream )}

\texttt{IsEndOfStream} returns \texttt{true} if the input stream is at \texttt{end-of-stream}, and \texttt{false} otherwise. Note that \texttt{IsEndOfStream} might return \texttt{false} even if the next \texttt{ReadByte} fails.

8 ▶ \texttt{PositionStream( input-stream )}

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 \texttt{PositionStream}, \texttt{SeekPositionStream} and \texttt{RewindStream}. \texttt{PositionStream} returns a non-negative integer denoting the current position in the stream (usually the number of characters \texttt{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 \texttt{fail} is returned.

\begin{verbatim}
9> RewindStream( input-stream )

RewindStream attempts to return an input stream to its starting condition, so that all the same characters can be read again. It returns \texttt{true} if the rewind succeeds and \texttt{fail} otherwise.

A default method implements RewindStream using \texttt{SeekPositionStream}.
\end{verbatim}

\begin{verbatim}
10> SeekPositionStream( input-stream, pos )

SeekPositionStream attempts to rewind or wind forward an input stream to the specified position. This is not possible for all streams. It returns \texttt{true} if the seek is successful and \texttt{fail} otherwise.
\end{verbatim}

\section*{10.4 Operations for Output Streams}

\begin{verbatim}
1> WriteByte( output-stream, byte )
\end{verbatim}

writes the next character (given as integer) to the output stream \texttt{output-stream}. The function returns \texttt{true} if the write succeeds and \texttt{fail} otherwise.

WriteByte is the basic operation for input 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).

\begin{verbatim}
2> WriteLine( output-stream, string )
\end{verbatim}

appends \texttt{string} to \texttt{output-stream}. A final newline is written. The function returns \texttt{true} if the write succeeds and \texttt{fail} otherwise.

A default method is installed which implements WriteLine by repeated calls to WriteByte.

\begin{verbatim}
3> WriteAll( output-stream, string )
\end{verbatim}

appends \texttt{string} to \texttt{output-stream}. No final newline is written. The function returns \texttt{true} if the write succeeds and \texttt{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.

When printing or appending to a stream (using PrintTo, or AppendTo or when logging to a stream), the kernel generates a call to WriteAll for each line output.

Example

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
4> PrintTo( output-stream, arg1, ... )
\end{verbatim}

These functions work like Print, except that the output is appended to the output stream \texttt{output-stream}.
Section 4. Operations for Output Streams

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)
```

5. LogTo( stream )

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 9.7.5).

6. InputLogTo( stream )

causes the subsequent input to be logged to the output stream stream. It works just like it does for files (see 9.7.7).

7. OutputLogTo( stream )

causes the subsequent output to be logged to the output stream stream. It works just like it does for files (see 9.7.9).

When text is being sent to an output text stream via PrintTo, AppendTo, LogTo, etc., it is, by default formatted just as it would be if it was 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 is 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.

8. SetPrintFormattingStatus( stream, newstatus )

sets whether output sent to the output stream stream via PrintTo, AppendTo, etc. (but not WriteByte, WriteLine or WriteAll) 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.

9. PrintFormattingStatus( stream )

returns true if output sent to the output stream stream via PrintTo, AppendTo, etc. (but not WriteByte, WriteLine or WriteAll) will be formatted with line breaks and indentation, and false otherwise (see 10.4.8).

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, 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, 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, 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 ]
```
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 (see 9.1.1) can be used to get information about the error.

1 ▶ InputTextFile( name-file )

InputTextFile( name-file ) returns an input stream in the category IsInputTextStream that delivers the characters from the file name-file.

2 ▶ OutputTextFile( name-file, append )

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 list.

Example

# use a temporary directory
gap> name := Filename( DirectoryTemporary(), "test" );;

# create an output stream, append output, and close again
gap> output := OutputTextFile( name, true );;
gap> AppendTo( output, "Hallo\n", "You\n" );
gap> CloseStream(output);

# create an input, print complete contents of file, and close
gap> input := InputTextFile(name);
gap> Print( ReadAll(input) );
Hallo
You

# append a single line
gap> output := OutputTextFile( name, true );;
gap> AppendTo( output, "AppendLine\n" );

# close output stream to flush the output
gap> CloseStream(output);
# 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 following two commands create streams which accept characters from, or deliver characters to, the user, via the keyboard or the GAP session display.

1. `InputTextUser( )` 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.

2. `OutputTextUser( )` 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 9.7.5).

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.

1. `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.

2. `OutputTextString(list, append)` 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

```plaintext
# read input from a string
gap> input := InputTextString( "Hallo\nYou\n" );;
gap> ReadLine(input);
"Hallo"
gap> ReadLine(input);
"You"

# 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.

IsInputOutputStream( obj )

IsInputOutputStream is the Category of Input-Output Streams; it returns true if the obj is an input-output stream and false otherwise.

At present the only type of Input-Output streams that are implemented provide communication with a local child process, using a pseudo-tty. They are only available on UNIX systems.

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.

InputOutputLocalProcess( dir, executable, args )

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 if closed, the signal SIGTERM is delivered to the child process, which is expected to exit.

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 >
3 ReadAllLine( iostream[, nofail][, IsAllLine] )

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

\[ \text{line} \rightarrow 0 < \text{Length(line)} \text{ and line}[\text{Length(line)}] = \backslash \text{n} \]

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 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.

1 InputTextNone( )
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 (see 11.1.1) when the started program does not read anything.

2 OutputTextNone( )
returns a dummy output stream, which discards all received characters. Its main use is for calls to Process (see 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. This is done via:

1 InstallCharReadHookFunc( stream, mode, func )

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 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.
Handlers can be removed via:

```c
2 UnInstallCharReadHookFunc( stream, func )
```

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.

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.
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 10.8.2).

### 11.1 Process

#### Process

| `Process(dir, prg, stream-in, stream-out, options)` |

**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** (see 9.3.3)—this was the behaviour of GAP 3—or a temporary directory returned by **DirectoryTemporary** (see 9.3.2). 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 **DirectoryPackageLibrary** should be used. The argument of **DirectoryPackageLibrary** 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** 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.

**Examples**

```gap
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
```
Chapter 11. Processes

```gap
gap> # create a temporary directory
gap> tmpdir := DirectoryTemporary();;

gap> Process( tmpdir, ls, stdin, stdout, ["-c"] );

gap> PrintTo( Filename( tmpdir, "emil" ) );

emil

```

`prg` is the filename of the program to launch, for portability it should be the result of `Filename` (see 9.4.1) and should pass `IsExecutableFile`. Note that `Process` does no searching through a list of directories, this is done by `Filename`.

`stream-in` is the input stream that delivers the characters to the process. For portability it should either be `InputTextNone` (if the process reads no characters), `InputTextUser`, the result of a call to `InputTextFile` from which no characters have been read, or the result of a call to `InputTextString`.

`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` (if the process writes no characters), `OutputTextUser`, the result of a call to `OutputTextFile` to which no characters have been written, or the result of a call to `OutputTextString`.

`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`.

**Examples**

In order to find a system program use `DirectoriesSystemPrograms` together with `Filename`.

```gap
gap> path := DirectoriesSystemPrograms();;

gap> date := Filename( path, "date" );

"/bin/date"

```

Now execute `date` with no argument and no input, collect the output into a string stream.

```gap
gap> str := "";; a := OutputTextString(str,true);

gap> Process( DirectoryCurrent(), date, InputTextNone(), a, [] );

0

```

```gap

Fri Jul 11 09:04:23 MET DST 1997

```

### 11.2 Exec

`Exec( cmd, option1, ..., optionN )` F

`Exec` runs a shell in the current directory to execute the command given by the string `cmd` with options `option1, ..., optionN`.

```gap

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.

```gap

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` (see 11.1.1). `Edit` (see 6.9.1) should be used to call an editor from within GAP.
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.

\[ \text{IsObject}( \text{obj} ) \]  

IsObject returns true if the object obj is an object. Obviously it can never return false. It can be used as a filter in \text{InstallMethod} (see 2.2 in “Programming in GAP”) 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.
Chapter 12. Objects and Elements

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 (see 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 the Reference Manual section “Sorted Lists and Sets” 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 30.

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.

\begin{verbatim}
F IsIdenticalObj( obj1, obj2 )
\end{verbatim}

IsIdenticalObj( obj1, obj2 ) 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 value 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.

\begin{verbatim}
gap> IsIdenticalObj( 10^6, 10^6); true
\end{verbatim}

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.)

```gap
gap> l1:= [ 1, 2, 3 ];; l2:= [ 1, 2, 3 ];;
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.

```gap
gap> l1:= [ 1, 2, 3 ];; l2:= l1;;
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.

The library also provides:

```gap
2 ▶ IsNotIdenticalObj( obj1, obj2 )  F
```

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 and Unbind 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.
Chapter 12. Objects and Elements

1. \textbf{IsCopyable( obj )}  

If a mutable form of an object \textit{obj} can be made in GAP, the object is called \textit{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 \textit{ShallowCopy} (see 12.7).

Objects for which only an immutable form exists in GAP are called \textit{constants}. Examples of constants are integers, permutations, and domains. Called with a constant as argument, \textit{Immutable} and \textit{ShallowCopy} return this argument.

2. \textbf{IsMutable( obj )}  

tests whether \textit{obj} is mutable.

If an object is mutable then it is also copyable (see 12.6.1), and a \textit{ShallowCopy} (see 12.7.1) method should be supplied for it. Note that \textit{IsMutable} must not be implied by another filter, since otherwise \textit{Immutable} would be able to create paradoxical objects in the sense that \textit{IsMutable} for such an object is \textit{false} but the filter that implies \textit{IsMutable} is \textit{true}.

In many situations, however, one wants to ensure that objects are \textit{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:

3. \textbf{Immutable( obj )}  

returns an immutable structural copy (see 12.7.2) of \textit{obj} in which the subobjects are immutable \textit{copies} of the subobjects of \textit{obj}. If \textit{obj} is immutable then \textit{Immutable} returns \textit{obj} itself. GAP will complain with an error if one tries to change an immutable object.

4. \textbf{MakeImmutable( obj )}  

One can turn the (mutable or immutable) object \textit{obj} into an immutable one with \textit{MakeImmutable}; note that this also makes all subobjects of \textit{obj} immutable, so one should call \textit{MakeImmutable} only if \textit{obj} and its mutable subobjects are newly created. If one is not sure about this, \textit{Immutable} 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 \textit{Immutable}, 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 \textit{Immutable} 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.

Another interesting example of mutable (and thus copyable) objects is provided by \textit{iterators}, see 28.7. (Of course an immutable form of an iterator is not very useful, but clearly \textit{Immutable} will yield such an object.) Every call of \textit{NextIterator} changes a mutable iterator until it is exhausted, and this is the only way to change an iterator. \textit{ShallowCopy} for an iterator \textit{iter} is defined so as to return a mutable iterator that has no mutable data in common with \textit{iter}, and that behaves equally to \textit{iter} w.r.t. \textit{IsDoneIterator} and (if \textit{iter} is mutable) \textit{NextIterator}. Note that this meaning of the “shallow copy” of an iterator that is returned by \textit{ShallowCopy} is not as obvious as for lists and records, and must be explicitly defined.

Many operations return immutable results, among those in particular attributes (see 13.5). Examples of attributes are \textit{Size}, \textit{Zero}, \textit{AdditiveInverse}, \textit{One}, and \textit{Inverse}. Arithmetic operations, such as the binary infix operations +, -, *, /, \textit{mod}, the unary -, and operations such as \textit{Comm} and \textit{LeftQuotient}, 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 that 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 \(0 \ast \text{obj}\) is equivalent to \text{ZeroSM}(\text{obj})\), \(-\text{obj}\) is equivalent to \text{AdditiveInverseSM}(\text{obj})\), \text{obj}^{-0}\) is equivalent to \text{OneSM}(\text{obj})\), and \text{obj}^{-1}\) is equivalent to \text{InverseSM}(\text{obj})\). The “SM” stands for “same mutability”, and indicates that the result is mutable if and only if the argument is mutable.

The operations \text{ZeroOp}, \text{AdditiveInverseOp}, \text{OneOp}, and \text{InverseOp} return mutable results whenever a mutable version of the result exists, contrary to the attributes \text{Zero}, \text{AdditiveInverse}, \text{One}, and \text{Inverse}.

If one introduces new arithmetic objects then one need not install methods for the attributes \text{One}, \text{Zero}, etc. The methods for the associated operations \text{OneOp} and \text{ZeroOp} 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 \text{TransposedMatMutable} constructs a mutable matrix (contrary to the attribute \text{TransposedMat}), while \text{TriangulizeMat} modified 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

1. \textbf{ShallowCopy}(\textit{obj})

   If GAP supports a mutable form of the object \textit{obj} (see 12.6) then this is obtained by \textbf{ShallowCopy}. Otherwise \textbf{ShallowCopy} returns \textit{obj} itself.

   The subobjects of \textbf{ShallowCopy}(\textit{obj}) are identical to the subobjects of \textit{obj}. Note that if the object returned by \textbf{ShallowCopy} is mutable then it is always a new object. In particular, if the return value is mutable, then it is not identical with the argument \textit{obj}, no matter whether \textit{obj} is mutable or immutable.

   But of course the object returned by \textbf{ShallowCopy} is equal to \textit{obj} w.r.t. the equality operator \(=\).

   Since \textbf{ShallowCopy} is an operation, the concrete meaning of “subobject” depends on the type of \textit{obj}. But for any copyable object \textit{obj}, the definition should reflect the idea of “first level copying”.

   The definition of \textbf{ShallowCopy} for lists (in particular for matrices) can be found in 21.7.

2. \textbf{StructuralCopy}(\textit{obj})

   In a few situations, one wants to make a structural copy \textit{scp} of an object \textit{obj}. This is defined as follows. \textit{scp} and \textit{obj} are identical if \textit{obj} is immutable. Otherwise, \textit{scp} is a mutable copy of \textit{obj} such that each subobject of \textit{scp} is a structural copy of the corresponding subobject of \textit{obj}. Furthermore, if two subobjects of \textit{obj} are identical then also the corresponding subobjects of \textit{scp} are identical.
That both ShallowCopy 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 26.5.1, 6.2.3 and 6.2.4. Others are mainly somewhat technical.

1 ▶ SetName( obj, name )

for a suitable object obj sets that object to have name name (a string).

2 ▶ Name( obj )

returns the name, a string, previously assigned to obj via a call to SetName (see 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.

   gap> g := Group((1,2,3),(1,2));;
   Group([ (1,2,3), (1,2) ])
   gap> SetName(g, "S3");
   gap> g;
   S3
   gap> Name(g);
   "S3"

3 ▶ IsInternallyConsistent( obj )

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.)
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 2 in “Programming in GAP”).

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 “Programming in GAP” in Section 3.

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.

1 ▶ FamilyObj( obj ) returns the family of the object obj.

The family of the object obj is itself an object, its family is the 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

\[ \text{filt1 and filt2} \]

See 20.3.3 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 is defined as the intersection of the category IsMagmaWithInverses and the property IsAssociative.

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 filt has a rank, which is used to define a ranking of the methods installed for an operation, see Section 2.2 in “Programming in GAP”. The rank of a filter can be accessed as

\[ \text{RankFilter( filt )} \]

For simple filters, an incremental rank is defined when the filter is created, see the sections about the creation of filters 3.1, 3.2, 3.3, 3.4; all in “Programming in GAP”. For an arbitrary filter, its rank is given by
the sum of the incremental ranks of the \textit{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 \textit{only} for this purpose, attribute testers are treated as if they would imply the requirements of their attributes.

\begin{enumerate}
\item \texttt{NamesFilter( filt )}
\item \texttt{ShowImpliedFilters( filter )}
\end{enumerate}

\texttt{NamesFilter} returns a list of names of the \textit{implied} simple filters of the filter \textit{filt}, these are all those simple filters \textit{imp} such that every object in \textit{filt} also lies in \textit{imp}. For implications between filters, see 13.2.3 as well as sections 2.7, 3.1, 3.2, 3.3 in “Programming in \texttt{GAP}”.

Displays information about the filters that may be implied by \textit{filter}. They are given by their names. \texttt{ShowImpliedFilters} first displays the names of all filters that are unconditionally implied by \textit{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 \texttt{GAP} will do these).

\begin{verbatim}
gap> ShowImpliedFilters(IsRationalFunction);
Implies:
  IsNearAdditiveElementWithInverse
  IsAdditiveElement
  IsMultiplicativeElementWithInverse

  May imply with:
  +IsZero
  +Tester(IsZero)
  IsConstantRationalFunction
  Tester(IsConstantRationalFunction)
\end{verbatim}

13.3 Categories

The \texttt{categories} of an object are filters (see 13.2) 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 \texttt{IsList} must have methods for \texttt{Length}, \texttt{IsBound}\[\backslash\}\texttt{[}] and the list element access operation \texttt{\[\backslash\].}

An object can lie in several categories. For example, a row vector lies in the categories \texttt{IsList} and \texttt{IsVector}; each list lies in the category \texttt{IsCopyable}, and depending on whether or not it is mutable, it may lie in the category \texttt{IsMutable}. Every domain lies in the category \texttt{IsDomain}.

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 \texttt{IsDenseList}.

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.
Chapter 13. Types of Objects

Every object lies in the category IsObject.

The categories IsExtLElement and IsExtRElement 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 contains objects that can be multiplied from the left and from the right with objects from the same family. IsMultiplicativeElementWithOne contains objects \(obj\) for which a multiplicatively neutral element can be obtained by taking the zeroth power \(obj^0\). IsMultiplicativeElementWithInverse contains objects \(obj\) for which a multiplicative inverse can be obtained by forming \(obj^{-1}\).

Likewise, the categories IsExtAElement, IsAdditiveElement, IsAdditiveElementWithZero, and IsAdditiveElementWithInverse 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 \(-1\).


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 is a subcategory of IsMultiplicativeElementWithInverse. Clearly not every matrix has a multiplicative inverse. But the category IsMatrix makes each matrix an admissible argument of the operation Inverse, 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.

IsObject
IsDomain
IsMagma
IsMagmaWithOne
IsMagmaWithInversesIfNonzero
IsMagmaWithInverses
IsAdditiveMagma
IsAdditiveMagmaWithZero
IsAdditiveMagmaWithInverses
IsExtLSet
IsExtRSet

Of course IsDomain is a subcategory of IsObject. A domain that is closed under multiplication \(*\) is called a magma and it lies in the category IsMagma. If a magma is closed under taking the identity, it lies in IsMagmaWithOne, and if it is also closed under taking inverses, it lies in IsMagmaWithInverses. The category IsMagmaWithInversesIfNonzero 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.

\begin{verbatim}
CategoriesOfObject( object )
\end{verbatim}
returns a list of the names of the categories in which \textit{object} lies.

```gap
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)",
  "CategoryCollections(\text{IS\_PERM})",
  "IsGeneralizedDomain", "IsMagma", "IsMagmaWithOne",
  "IsMagmaWithInversesIfNonzero", "IsMagmaWithInverses" ]
```

**13.4 Representation**

The \textit{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.

\textsc{GAP} distinguishes four essentially different ways to represent objects. First there are the representations \texttt{IsInternalRep} for internal objects such as integers and permutations, and \texttt{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 \textsc{GAP} level.

All other objects are either in the representation \texttt{IsComponentObjectRep} or in the representation \texttt{IsPositionalObjectRep}, see 3.9 and 3.10 in “Programming in \textsc{GAP}”.

An object can belong to several representations in the sense that it lies in several subrepresentations of \texttt{IsComponentObjectRep} or of \texttt{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 \texttt{IsComponentObjectRep} and \texttt{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 \texttt{IsPositionalObjectRep} are \texttt{IsModulusRep}, which is used for residue classes in the ring of integers, and \texttt{IsDenseCoeffVectorRep}, which is used for elements of algebras that are defined by structure constants.

An important subrepresentation of \texttt{IsComponentObjectRep} is \texttt{IsAttributeStoringRep}, which is used for many domains and some other objects. It provides automatic storing of all attribute values (see below).
13. Types of Objects

Chapter 13. Types of Objects

13.5 Attributes

The attributes of an object are filters (see 13.2) that 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.

Properties are a special form of attributes that have the value true or false, see section 13.7.

All attributes in the library are created during initialization, in particular they are not created dynamically at runtime.

Examples of attributes for multiplicative elements are Inverse, One, and Order. Size is an attribute for domains, Centre is an attribute for magmas, and DerivedSubgroup is an attribute for groups.

13.6 Properties

Properties are attributes that may be called dynamically at runtime. They are created dynamically at runtime and may therefore be called before they are initialized.

Examples of properties for multiplicative elements are Inverse, One, and Order. Size is a property for domains, Centre is a property for magmas, and DerivedSubgroup is a property for groups.
13.6 Setter and Tester for Attributes

For every attribute two further operations, the attribute setter and the attribute tester are defined. To check whether an object belongs to an attribute \( \text{attr} \), the tester

\[\text{Tester}( \text{attr} )\]

of the attribute is used; this is a filter (see 13.2) that returns \text{true} or \text{false}, depending on whether or not the value of \( \text{attr} \) for the object is known. For example, \( \text{Tester}( \text{Size } )( \text{obj} ) \) is \text{true} if the size of the object \( \text{obj} \) is known.

To store a value for the attribute \( \text{attr} \) in an object, the setter

\[\text{Setter}( \text{attr} )\]

of the attribute is used. The setter is called automatically when the attribute value has been computed for the first time. One can also call the setter explicitly, for example, \( \text{Setter}( \text{Size } )( \text{obj}, \text{val} ) \) sets \( \text{val} \) as size of the object \( \text{obj} \) if the size was not yet known.

For each attribute \( \text{attr} \) that is declared with \text{DeclareAttribute} resp. \text{DeclareProperty} (see 3.15 in “Programming in GAP”), tester and setter are automatically made accessible by the names \text{Hasattr} and \text{Setattr}, respectively. For example, the tester for \text{Size} is called \text{HasSize}, and the setter is called \text{SetSize}.

\[
\text{gap> g:=Group((1,2,3,4),(1,2));;Size(g);} \\
24 \\
\text{gap> HasSize(g);} \\
\text{true} \\
\text{gap> SetSize(g,99);} \\
\text{gap> Size(g);} \\
24
\]

For two properties \( \text{prop1} \) and \( \text{prop2} \), the intersection \( \text{prop1} \) and \( \text{prop2} \) (see 13.2) is again a property for which a setter and a tester exist. Setting the value of this intersection to \text{true} for a GAP object means to set the values of \( \text{prop1} \) and \( \text{prop2} \) to \text{true} for this object.

\[
\text{gap> prop:= IsFinite and IsCommutative;} \\
\text{<Operation "&&-
filter>>"}> \\
\text{gap> g:= Group( (1,2,3), (4,5) );;} \\
\text{gap> Tester( prop )( g );} \\
\text{false} \\
\text{gap> Setter( prop )( g, true );} \\
\text{gap> Tester( prop )( g ); prop( g );} \\
\text{true} \\
\text{true}
\]

It is not allowed to set the value of such an intersection to \text{false} for an object.

\[
\text{gap> Setter( prop )( Rationals, false );} \\
\text{You cannot set an "and-filter" except to true} \\
\text{not in any function} \\
\text{Entering break read-eval-print loop, you can 'quit;' to quit to outer loop,} \\
\text{or you can return true to set all components true,} \\
\text{but you might really want to reset just one component to continue}
\]

3.\[\text{AttributeValueNotSet}( \text{attr}, \text{obj} )\]

If the value of the attribute \( \text{attr} \) is already stored for \( \text{obj} \), \text{AttributeValueNotSet} simply returns this value. Otherwise the value of \( \text{attr}( \text{obj} ) \) is computed and returned without storing it in \( \text{obj} \). This can be useful
when “large” attribute values (such as element lists) are needed only once and shall not be stored in the object.

```
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.

4► `InfoAttributes` V

This info class (together with `InfoWarning`; see 7.4.6) 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.

5► `DisableAttributeValueStoring( attr )` F

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.

6► `EnableAttributeValueStoring( attr )` F

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.

```
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);
```
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, IsCommutative, IsAnticommutevative, IsLDistributive, and IsRDistributive, which mean that the multiplication of elements in the domain satisfies 

\[(a * b) * c = a * (b * c), a * b = b * a, a * b = -(b * a), a * (b + c) = a * b + a * c, (a + b) * c = a * c + b * c,\]

respectively, for all a, b, c in the domain.

1. KnownPropertiesOfObject(object) returns a list of the names of the properties whose values are known for object.

2. KnownTruePropertiesOfObject(object) returns a list of the names of the properties known to be true for object.

```gap
gap> g:=Group((1,2),(1,2,3));;
gap> KnownPropertiesOfObject(g);
[ "IsFinite", "CanEasilyCompareElements", "CanEasilySortElements", 
  "IsDuplicateFree", "IsGeneratorsOfMagmaWithInverses", "IsAssociative", 
  "IsFinitelyGeneratedGroup", "IsSubsetLocallyFiniteGroup", 
  "KnowsHowToDecompose", "IsChainTypeGroup", "IsStabChainViaChainSubgroup" ]
gap> Size(g);
6
```

```gap
gap> KnownPropertiesOfObject(g);
[ "IsEmpty", "IsTrivial", "IsNonTrivial", "IsFinite", 
  "CanEasilyCompareElements", "CanEasilySortElements", "IsDuplicateFree", 
  "IsGeneratorsOfMagmaWithInverses", "IsAssociative", 
  "IsFinitelyGeneratedGroup", "IsSubsetLocallyFiniteGroup", 
  "KnowsHowToDecompose", "IsPerfectGroup", "IsSolvableGroup", 
  "IsPolycyclicGroup", "IsChainTypeGroup", "IsStabChainViaChainSubgroup" ]
gap> KnownTruePropertiesOfObject(g);
[ "IsNonTrivial", "IsFinite", "CanEasilyCompareElements", 
  "CanEasilySortElements", "IsDuplicateFree", 
  "IsGeneratorsOfMagmaWithInverses", "IsAssociative", 
  "IsFinitelyGeneratedGroup", "IsSubsetLocallyFiniteGroup", 
  "KnowsHowToDecompose", "IsSolvableGroup", "IsPolycyclicGroup" ]
```
Chapter 13. Types of Objects

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 and CanEasilyComputePcgs 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 2.7 in “Programming in GAP”) from other filters.

For example, a true value of CanEasilyComputePcgs for a group means that certain methods are applicable that use a pcgs (see 43.1) 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 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.

1 ▶ TypeObj( obj ) F

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.

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 as

2 ▶ DataType( type ) F
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.

\begin{verbatim}
gap> -1234;
-1234


\end{verbatim}

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 17.

\begin{verbatim}
1 Integers V
  PositiveIntegers V
  NonnegativeIntegers V

These global variables represent the ring of integers and the semirings of positive and nonnegative integers, respectively.

\begin{verbatim}
gap> Size( Integers ); 2 in Integers;
infinity true

2 IsIntegers( obj ) C
  IsPositiveIntegers( obj ) C
  IsNonnegativeIntegers( obj ) C

are the defining categories for the domains Integers, PositiveIntegers, and NonnegativeIntegers.

\begin{verbatim}
gap> IsIntegers( Integers ); IsIntegers( Rationals ); IsIntegers( 7 );
true false false

Integers is a subset of Rationals, which is a subset of Cyclotomics. See Chapter 18 for arithmetic operations and comparison of integers.
\end{verbatim}
\end{verbatim}
14.1 Elementary Operations for Integers

1 ▶ IsInt( obj )
   Every rational integer lies in the category IsInt, which is a subcategory of IsRat (see 16).

2 ▶ IsPosInt( obj )
   Every positive integer lies in the category IsPosInt.

3 ▶ Int( elm )
   Int returns an integer int whose meaning depends on the type of elm.
   If elm is a rational number (see 16) then int is the integer part of the quotient of numerator and denominator of elm (see 14.2.1).
   If elm is an element of a finite prime field (see Chapter 57) then int is the smallest nonnegative integer such that elm = int * One( elm ).
   If elm is a string (see Chapter 26) consisting of digits '0', '1', ..., '9' and '-' (at the first position) then int is the integer described by this string. The operation String (see 26.5.1) can be used to compute a string for rational integers, in fact for all cyclotomics.

   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

4 ▶ IsEvenInt( n )
   tests if the integer n is divisible by 2.

5 ▶ IsOddInt( n )
   tests if the integer n is not divisible by 2.

6 ▶ AbsInt( n )
   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 see 54.6.2).
   See also 18.1.6.

   gap> AbsInt( 33 );
   33
   gap> AbsInt( -214378 );
   214378
   gap> AbsInt( 0 );
   0

7 ▶ SignInt( n )
   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.

   gap> SignInt( 33 );
1

```gap
gap> SignInt( -214378 );
-1
gap> SignInt( 0 );
0
```

8 ▶ \texttt{LogInt}( \textit{n}, \textit{base} )

\texttt{LogInt} returns the integer part of the logarithm of the positive integer \textit{n} with respect to the positive integer \textit{base}, i.e., the largest positive integer \textit{exp} such that \textit{base}^{\textit{exp}} \leq \textit{n}. \texttt{LogInt} will signal an error if either \textit{n} or \textit{base} is not positive.

For \textit{base} 2 this is very efficient because the internal binary representation of the integer is used.

```gap
gap> LogInt( 1030, 2 );
10 # 2^{10} = 1024
gap> LogInt( 1, 10 );
0
```

9 ▶ \texttt{RootInt}( \textit{n} )

\texttt{RootInt} returns the integer part of the \textit{k}th root of the integer \textit{n}. If the optional integer argument \textit{k} is not given it defaults to 2, i.e., \texttt{RootInt} returns the integer part of the square root in this case.

If \textit{n} is positive, \texttt{RootInt} returns the largest positive integer \textit{r} such that \textit{r}^{\textit{k}} \leq \textit{n}. If \textit{n} is negative and \textit{k} is odd \texttt{RootInt} returns -\texttt{RootInt}( -\textit{n}, \textit{k} ). If \textit{n} is negative and \textit{k} is even \texttt{RootInt} will cause an error. \texttt{RootInt} will also cause an error if \textit{k} is 0 or negative.

```gap
gap> RootInt( 361 );
19
gap> RootInt( 2 * 10^12 );
1414213
gap> RootInt( 17000, 5 );
7 # 7^5 = 16807
```

10 ▶ \texttt{SmallestRootInt}( \textit{n} )

\texttt{SmallestRootInt} returns the smallest root of the integer \textit{n}.

The smallest root of an integer \textit{n} is the integer \textit{r} of smallest absolute value for which a positive integer \textit{k} exists such that \textit{n} = \textit{r}^{\textit{k}}.

```gap
gap> SmallestRootInt( 2^30 );
2
gap> SmallestRootInt( -2^30 );
-4 # note that $(-2)^{30} = +(2^{30})$
gap> SmallestRootInt( 279936 );
6
gap> LogInt( 279936, 6 );
7
gap> SmallestRootInt( 1001 );
1001
```

11 ▶ \texttt{Random( Integers )}

\texttt{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 construct 'Random( [ \textit{low} .. \textit{high} ] )'. (Also see 28.6.1.)
Chapter 14. Integers

1. ▶ QuoInt(n, m)

QuoInt returns the integer part of the quotient of its integer operands.

If \( n \) and \( m \) are positive \( \text{QuoInt}(n, m) \) is the largest positive integer \( q \) such that \( q \times m \leq n \). 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 (see 54.6.3).

\[
\text{gap> QuoInt}(5,3); \text{QuoInt}(-5,3); \text{QuoInt}(5,-3); \text{QuoInt}(-5,-3);
\]

\[1\ -1\ -1\ 1\]

2. ▶ BestQuoInt(n, m)

BestQuoInt returns the best quotient \( q \) of the integers \( n \) and \( m \). This is the quotient such that \( n - q \times m \) 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.

\[
\text{gap> BestQuoInt}(5,3); \text{BestQuoInt}(-5,3);
\]

\[2\ -2\]

3. ▶ RemInt(n, m)

RemInt returns the remainder of its two integer operands.

If \( m \) is not equal to zero \( \text{RemInt}(n, m) = n - m \times \text{QuoInt}(n, m) \). Note that the rules given for QuoInt imply that \( \text{RemInt}(n, m) \) has the same sign as \( n \) and its absolute value is strictly less than the absolute value of \( m \). Note also that \( \text{RemInt}(n, m) = n \mod m \) when both \( n \) and \( m \) are nonnegative.

Dividing by 0 signals an error.

RemInt is used in a method for the general operation EuclideanRemainder (see 54.6.4).

\[
\text{gap> RemInt}(5,3); \text{RemInt}(-5,3); \text{RemInt}(5,-3); \text{RemInt}(-5,-3);
\]

\[2\ -2\ 2\ -2\]

4. ▶ GcdInt(m, n)

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 \( \text{GcdInt}(m, 0) = \text{GcdInt}(0, m) = \text{AbsInt}(m) \) and \( \text{GcdInt}(0, 0) = 0 \).

GcdInt is a method used by the general function Gcd (see 54.7.1).

\[
\text{gap> GcdInt}(123, 66);
\]

\[3\]

5. ▶ Gcdex(m, n)

returns a record \( g \) describing the extended greatest common divisor of \( m \) and \( n \). The component \( \text{gcd} \) is this gcd, the components \( \text{coeff1} \) and \( \text{coeff2} \) are integer cofactors such that \( g.\text{gcd} = g.\text{coeff1} \times m +\)
Section 2. Quotients and Remainders

If $m$ and $n$ both are nonzero, $\text{AbsInt}(g.\text{coeff1})$ is less than or equal to $\text{AbsInt}(n) / (2 \times g.\gcd)$ and $\text{AbsInt}(g.\text{coeff2})$ is less than or equal to $\text{AbsInt}(m) / (2 \times g.\gcd)$.

If $m$ or $n$ or both are zero, $\text{coeff3}$ is $-n / g.\gcd$ and $\text{coeff4}$ is $m / g.\gcd$.

The coefficients always form a unimodular matrix, i.e., the determinant $g.\text{coeff1} \times g.\text{coeff4} - g.\text{coeff3} \times g.\text{coeff2}$ is 1 or $-1$.

\begin{verbatim}
gap> Gcdex( 123, 66 );  
# 3 = 7\times123 - 13\times66, 0 = -22\times123 + 41\times66  
gap> Gcdex( 0, -3 );  
rec( gcd := 3, coeff1 := 0, coeff2 := -1, coeff3 := 1, coeff4 := 0 )  
gap> Gcdex( 0, 0 );  
rec( gcd := 0, coeff1 := 1, coeff2 := 0, coeff3 := 0, coeff4 := 1 )
\end{verbatim}

returns the least common multiple of the integers $m$ and $n$. $\text{LcmInt}$ is a method used by the general function $\text{Lcm}$.

\begin{verbatim}
gap> LcmInt( 123, 66 );  
2706
\end{verbatim}

returns the $q$-adic representation of the integer $i$ as a list $l$ of coefficients where $i = \sum_{j=0}^{\infty} q^j \cdot l[j+1]$.

returns the multiadic expansion of the integer $int$ modulo the integers given in $\text{ints}$ (in ascending order). It returns a list of coefficients in the reverse order to that in $\text{ints}$.

returns the combination of the $\text{residues}$ modulo the $\text{moduli}$, i.e., the unique integer $c$ from $[0..\text{Lcm}(\text{moduli})-1]$ such that $c = \text{residues}[i]$ modulo $\text{moduli}[i]$ for all $i$, if it exists. If no such combination exists $\text{ChineseRem}$ signals an error.

Such a combination does exist if and only if $\text{residues}[i] = \text{residues}[k]$ mod $\gcd(\text{moduli}[i], \text{moduli}[k])$ 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.

\begin{verbatim}
gap> ChineseRem( [ 2, 3, 5, 7 ], [ 1, 2, 3, 4 ] );  
53

gap> ChineseRem( [ 6, 10, 14 ], [ 1, 3, 5 ] );  
103

gap> ChineseRem( [ 6, 10, 14 ], [ 1, 2, 3 ] );  
Error, the residues must be equal modulo 2 called from  
... lines omitted here ...
\end{verbatim}

returns $r^e \mod m$ for integers $r, e$ and $m$ ($e \geq 0$). Note that using $r^-e \mod m$ will generally be slower, because it can not reduce intermediate results the way $\text{PowerModInt}$ does but would compute $r^-e$ first and then reduce the result afterwards.

$\text{PowerModInt}$ is a method for the general operation $\text{PowerMod}$. 

---

Note: The provided content is a transcription of the document, including formatting and mathematical notations. It does not include diagrammatic representations or special symbols that may be present in the original document.
14.3 Prime Integers and Factorization

Primes

Primes is a strictly sorted list of the 168 primes less than 1000.

This is used in IsPrimeInt and FactorsInt to cast out small primes quickly.

\[ \text{gap} > \text{Primes}[1] \]
\[ \Rightarrow 2 \]
\[ \text{gap} > \text{Primes}[100] \]
\[ \Rightarrow 541 \]

IsPrimeInt

IsPrimeInt returns false if it can prove that \( n \) 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 \( n < 10^{13} \) and for all composite \( n \) that have a factor \( p < 1000 \). So for integers \( n < 10^{13} \), IsPrimeInt is a proper primality test. It is conceivable that IsPrimeInt may return true for some composite \( n > 10^{13} \), but no such \( n \) is currently known. So for integers \( n > 10^{13} \), IsPrimeInt is a probable-primality test. Therefore IsPrimeInt will issue a warning when called with an argument > 10^{13}. (The function IsProbablyPrimeInt will do the same calculations but not issue a warning.)

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 integer.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.

\[ \text{gap} > \text{IsPrimeInt}(2^{31} - 1) \]
\[ \Rightarrow \text{true} \]
\[ \text{gap} > \text{IsPrimeInt}(10^{42} + 1) \]
\[ \Rightarrow \text{false} \]

IsPrimePowerInt

IsPrimePowerInt returns true if the integer \( n \) is a prime power and false otherwise.

\( n \) 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 \( p^i = n \). 1 and -1 are not prime powers.

Note that IsPrimePowerInt uses SmallestRootInt (see 14.1.10) and a probable-primality test (see 14.3.2).
Section 3. Prime Integers and Factorization

gap> IsPrimePowerInt( 31^5 );
true
gap> IsPrimePowerInt( 2^31-1 );
true # $2^{31}-1$ is actually a prime
gap> IsPrimePowerInt( 2^63-1 );
false

gap> Filtered( [-10..10], IsPrimePowerInt );
[ -8, -7, -5, -3, -2, 2, 3, 4, 5, 7, 8, 9 ]

NextPrimeInt( n )

NextPrimeInt returns the smallest prime which is strictly larger than the integer n.

Note that NextPrimeInt uses a probable-primality test (see 14.3.2).

gap> NextPrimeInt( 541 ); NextPrimeInt( -1 );
547
2

PrevPrimeInt( n )

PrevPrimeInt returns the largest prime which is strictly smaller than the integer n.

Note that PrevPrimeInt uses a probable-primality test (see 14.3.2).

gap> PrevPrimeInt( 541 ); PrevPrimeInt( 1 );
523
-2

FactorsInt( n )

FactorsInt calls FactorsRho with a limit of trials on the number of trials is performs. The default is 8192.

FactorsInt returns a list of prime factors of the integer n.

If the $i$th power of a prime divides $n$ this prime appears $i$ times. The list is sorted, that is the smallest prime factors come first. The first element has the same sign as $n$, the others are positive. For any integer $n$ it holds that $\text{Product}( \text{FactorsInt}( n ) ) = n$.

Note that FactorsInt uses a probable-primality test (see 14.3.2). Thus FactorsInt might return a list which contains composite integers.

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.

In the second form, FactorsInt calls FactorsRho with a limit of trials on the number of trials is performs. The default is 8192.

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 );
#I beyond the guaranteed bound of the probabilistic primality test
prints the prime factorization of the integer \( n \) in human-readable form.

```gap
gap> PrintFactorsInt( Factorial( 7 ) ); Print( "\n" );
2^4*3^2*5*7
```

returns the prime factorization of the integer \( n \) as a list \([p_1, e_1, \ldots, p_n, e_n]\) with \( n = \prod_{i=1}^{n} p_i^{e_i} \).

```gap
gap> PrimePowersInt( Factorial( 7 ) );
[ 2, 4, 3, 2, 5, 1, 7, 1 ]
```

\( \text{DivisorsInt} \) 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 \( \text{Divisors}( -n ) = \text{Divisors}( n ) \).

Since the set of divisors of 0 is infinite calling \( \text{DivisorsInt}( 0 ) \) causes an error.

\( \text{DivisorsInt} \) may call \( \text{FactorsInt} \) to obtain the prime factors. \( \text{Sigma} \) and \( \text{Tau} \) (see 15.4.1 and 15.4.2) compute the sum and the number of positive divisors, respectively.

```gap
gap> DivisorsInt( 1 ); DivisorsInt( 20 ); DivisorsInt( 541 );
[ 1 ]
[ 1, 2, 4, 5, 10, 20 ]
[ 1, 541 ]
```

**14.4 Residue Class Rings**

If \( r, s \) and \( n \) are integers, \( r / s \mod n \) as a reduced fraction is \( p / q \), and \( q \) 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 4.12 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.

\( \text{ZmodnZ}( n ) \) corresponds to \( \text{Integers mod } n \).

\( \text{ZmodpZ}( p ) \) does the same if the argument \( p \) is a prime integer, additionally the result is a field. \( \text{ZmodpZNC}( p ) \) 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.
Section 4. Residue Class Rings

3. \texttt{ZmodnZObj( Fam, r )} \\
\texttt{ZmodnZObj( r, n )}

If the first argument is a residue class family \textit{Fam} then \texttt{ZmodnZObj} returns the element in \textit{Fam} whose coset is represented by the integer \textit{r}. If the two arguments are an integer \textit{r} and a positive integer \textit{n} then \texttt{ZmodnZObj} returns the element in \texttt{ZmodnZ( n )} (see 14.4.2) whose coset is represented by the integer \textit{r}.

\begin{verbatim}
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
\end{verbatim}

4. \texttt{IsZmodnZObj( obj )} \\
\texttt{IsZmodnZObjNonprime( obj )} \\
\texttt{IsZmodpObj( obj )} \\
\texttt{IsZmodpObjSmall( obj )} \\
\texttt{IsZmodpObjLarge( obj )}

The elements in the rings $\mathbb{Z}/n\mathbb{Z}$ are in the category \texttt{IsZmodnZObj}. If \textit{n} is a prime then the elements are of course also in the category \texttt{IsFFE} (see 57.1.1), otherwise they are in \texttt{IsZmodnZObjNonprime}. \texttt{IsZmodpObj} is an abbreviation of \texttt{IsZmodnZObj} and \texttt{IsFFE}. This category is the disjoint union of \texttt{IsZmodpObjSmall} and \texttt{IsZmodpObjLarge}, the former containing all elements with \textit{n} at most \texttt{MAXSIZE_GF_INTERNAL}.

The reasons to distinguish the prime case from the nonprime case are

- that objects in \texttt{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 \texttt{IsZmodnZObjNonprimeFamily} (note that for prime \textit{n}, the family must be an \texttt{IsFFEFamily}).

The reasons to distinguish the small and the large case are that for small \textit{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 \textit{n}.

Note that we cannot claim that every finite field element of degree 1 is in \texttt{IsZmodnZObj}, since finite field elements in internal representation may not know that they lie in the prime field.

The residue class rings are rings, thus all operations for rings (see Chapter 54) apply. See also Chapters 57 and 15.
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**. \( \phi(m) \) (see 15.1.2) is the order of this group, \( \lambda(m) \) (see 15.1.3) the exponent. If and only if \( m \) is 2, 4, an odd prime power \( p^e \), or twice an odd prime power \( 2p^e \), this group is cyclic. In this case the generators of the group, i.e., elements of order \( \phi(m) \), are called **primitive roots** (see 15.2.3, 15.2.4).

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.

1 ▶ **InfoNumtheor**

**InfoNumtheor** is the info class (see 7.4) for the functions in the number theory chapter.

### 15.1 Prime Residues

1 ▶ **PrimeResidues( m )**

**PrimeResidues** returns the set of integers from the range \( 0..\text{Abs}(m)-1 \) that are coprime to the integer \( m \). \( \text{Abs}(m) \) must be less than \( 2^{28} \), otherwise the set would probably be too large anyhow.

```gap
gap> PrimeResidues( 0 ); PrimeResidues( 1 ); PrimeResidues( 20 );
[ ]
[ 0 ]
[ 1, 3, 7, 9, 11, 13, 17, 19 ]
```

2 ▶ **Phi( m )**

**Phi** returns the number \( \phi(m) \) of positive integers less than the positive integer \( m \) that are coprime to \( m \).

Suppose that \( m = p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k} \). Then \( \phi(m) = p_1^{e_1-1}(p_1 - 1)p_2^{e_2-1}(p_2 - 1)\ldots p_k^{e_k-1}(p_k - 1) \).

```gap
gap> Phi( 12 ); Phi( 2^13-1 ); Phi( 2^15-1 );
4
8190 # which proves that 2^(13)-1 is a prime
27000
```

3 ▶ **Lambda( m )**

**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 15.1.2).

Carmichael’s theorem states that \( \lambda \) can be computed as follows: \( \lambda(2) = 1 \), \( \lambda(4) = 2 \) and \( \lambda(p^e) = 2^{e-2} \) if \( 3 \leq e \), \( \lambda(p^e) = (p-1)p^{e-1} \) (i.e. \( \phi(p) \)) if \( p \) is an odd prime and \( \lambda(n \cdot m) = \text{Lcm}(\lambda(n), \lambda(m)) \) if \( n, m \) 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.

\begin{verbatim}
gap> Lambda( 10 );
 4
gap> Lambda( 30 );
 4
gap> Lambda( 561 );
 80 # 561 is the smallest Carmichael number
\end{verbatim}

\section{Primitive Roots and Discrete Logarithms}

Let $n$ be a positive integer. \texttt{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.

\begin{verbatim}
gap> GeneratorsPrimeResidues( 1 );
  rec( primes := [ ], exponents := [ ], generators := [ ] )
gap> GeneratorsPrimeResidues( 4*3 );
  rec( primes := [ 2, 3 ], exponents := [ 2, 1 ], generators := [ 7, 5 ] )
gap> GeneratorsPrimeResidues( 8*9*5 );
  rec( primes := [ 2, 3, 5 ], exponents := [ 3, 2, 1 ],
       generators := [ [ 271, 181 ], 281, 217 ] )
\end{verbatim}

\subsection{OrderMod ($n, m$)}

\texttt{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 \texttt{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 15.2.4).

\texttt{OrderMod} usually spends most of its time factoring $m$ and $\phi(m)$ (see 14.3.6).

\begin{verbatim}
gap> OrderMod( 2, 7 ); OrderMod( 3, 7 );
  3
  6 # 3 is a primitive root modulo 7
\end{verbatim}

\subsection{LogMod ($n, r, m$)}

\texttt{LogMod} 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 \texttt{fail} is returned.
At the moment only a very naive method has been implemented.

\[
\text{gap> } l := \text{LogMod}(2, 5, 7); 5^l \mod 7 = 2; \\
4 \text{ true}
\]

\[
\text{gap> } \text{LogMod}(1, 3, 3); \text{LogMod}(2, 3, 3); \\
0 \text{ fail}
\]

\[\text{gap> } \text{PrimitiveRootMod}(m[, \text{start}])\]

**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***.

\[
\text{gap> } \text{PrimitiveRootMod}(409); \text{PrimitiveRootMod}(541, 2); \\
21 \# \text{largest primitive root for a prime less than 2000} \\
10
\]

\[
\text{gap> } \text{PrimitiveRootMod}(337, 327); \text{PrimitiveRootMod}(30); \\
\text{fail} \# \text{327 is the largest primitive root mod 337} \\
\text{fail} \# \text{there exists no primitive root modulo 30}
\]

\[\text{gap> } \text{IsPrimitiveRootMod}(r, m)\]

**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.

\[
\text{gap> } \text{IsPrimitiveRootMod}(2, 541); \text{IsPrimitiveRootMod}(-539, 541); \\
\text{true} \# \text{same computation as above} \\
\text{gap> } \text{IsPrimitiveRootMod}(4, 541); \\
\text{false} \\
\text{gap> } \text{ForAny}([1..29], r \rightarrow \text{IsPrimitiveRootMod}(r, 30)); \\
\text{false} \# \text{there does not exist a primitive root modulo 30}
\]

### 15.3 Roots Modulo Integers

\[\text{gap> } \text{Jacobi}(n, m)\]

**Jacobi** returns the value of the **Jacobi symbol** of the integer ***n*** modulo the integer ***m***.

Suppose that \(m = p_1 p_2 \cdots p_k\) is a product of primes, not necessarily distinct. Then for \(n\) coprime to \(m\) the Jacobi symbol is defined by \(J(n/m) = L(n/p_1)L(n/p_2)\cdots L(n/p_k)\), where \(L(n/p)\) is the Legendre symbol (see 15.3.2). By convention \(J(n/1) = 1\). If the gcd of \(n\) and \(m\) is larger than 1 we define \(J(n/m) = 0\). If \(n\) is a quadratic residue modulo \(m\), i.e., if there exists an \(r\) such that \(r^2 \equiv n \pmod m\) then \(J(n/m) = 1\). However, \(J(n/m) = 1\) implies the existence of such an \(r\) only if \(m\) is a prime.

**Jacobi** is very efficient, even for large values of \(n\) and \(m\), it is about as fast as the Euclidean algorithm (see 54.7.1).

\[
\text{gap> } \text{Jacobi}(11, 35); \\
1 \# 9^2 = 11 \mod 35 \\
\text{gap> } \text{Jacobi}(6, 35); \\
-1 \# \text{thus there is no } r \text{ such that } r^2 = 6 \mod 35 \\
\text{gap> } \text{Jacobi}(3, 35); \\
1 \# \text{even though there is no } r \text{ with } r^2 = 3 \mod 35
\]

\[\text{gap> } \text{Legendre}(n, m)\]

**Legendre** returns the value of the **Legendre symbol** of the integer ***n*** modulo the positive integer ***m***.
Section 3. Roots Modulo Integers

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 $\text{RootMod}$ (see 15.3.3).

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 15.3.1) 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 14.3.6).

A description of the Jacobi symbol, the Legendre symbol, and related topics can be found in [Bak84].

```gap
gap> Legendre( 5, 11 );
gap> Legendre( 6, 11 );
gap> Legendre( 3, 35 );
```

$\text{RootMod}$ 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 $\text{RootMod}$ returns $\text{fail}$. If only the arguments $n$ and $m$ are given, the default value for $k$ is 2.

In the current implementation $k$ must be a prime.

A square root of $n$ exists only if $\text{Legendre}(n,m) = 1$ (see 15.3.2). If $m$ has $r$ different prime factors then there are $2^r$ different roots of $n \pmod{m}$. It is unspecified which one $\text{RootMod}$ returns. You can, however, use $\text{RootsMod}$ (see 15.3.4) to compute the full set of roots.

$\text{RootMod}$ is efficient even for large values of $m$, in fact the most time is usually spent factoring $m$ (see 14.3.6).

```gap
gap> RootMod( 64, 1009 );
gap> RootMod( 64, 3, 1009 );
gap> RootMod( 64, 5, 1009 );
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ), x -> x mod 1009 );
```

$\text{RootsMod}$ computes the set of $k$th roots of the integer $n$ modulo the positive integer $m$, i.e., a $r$ such that $r^k \equiv n \pmod{m}$. If only the arguments $n$ and $m$ are given, the default value for $k$ is 2.

In the current implementation $k$ must be a prime.

```gap
gap> RootsMod( 1, 7*31 );
gap> RootsMod( 7, 7*31 );
gap> RootsMod( 5, 7*31 );
gap> RootsMod( 1, 5, 7*31 );
```

$\text{RootsUnityMod}$ returns the set of $k$-th roots of unity modulo the positive integer $m$, i.e., the list of all solutions $r$ of $r^k \equiv 1 \pmod{m}$. 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.

\[
\text{gap> RootsUnityMod( 7*31 ); RootsUnityMod( 3, 7*31 );}
\]
\[
[ 1, 92, 125, 216 ]
\]
\[
[ 1, 25, 32, 36, 67, 149, 156, 191, 211 ]
\]
\[
\text{gap> RootsUnityMod( 5, 7*31 );}
\]
\[
[ 1, 8, 64, 78, 190 ]
\]
\[
\text{gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),}
\]
\[
> x \rightarrow x \mod{1009} );
\]
\[
[ 1001, 8 ]
\]

15.4 Multiplicative Arithmetic Functions

\[\text{Sigma( n )}\]

**Sigma** returns the sum of the positive divisors of the integer $n$.

**Sigma** is a multiplicative arithmetic function, i.e., if $n$ and $m$ are relatively prime we have $\sigma(nm) = \sigma(n)\sigma(m)$.

Together with the formula $\sigma(p^e) = (p^{e+1} - 1)/(p - 1)$ 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, the known ones 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, and 859433. 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 14.3.6).

\[
\text{gap> Sigma( 0 );}
\]

*Error, Sigma: <n> must not be 0 called from ...
\]

4 lines omitted here ...

\[
\text{brk> quit;}
\]

\[
\text{gap> Sigma( 1 ); Sigma( 1009 ); Sigma( 8128 ) = 2*8128;}
\]

1

1010 # thus 1009 is a prime

true # thus 8128 is a perfect number

\[\text{ Tau( n )}\]

**Tau** returns the number of the positive divisors of the integer $n$.

**Tau** is a multiplicative arithmetic function, i.e., if $n$ and $m$ are relatively prime we have $\tau(nm) = \tau(n)\tau(m)$.

Together with the formula $\tau(p^e) = e + 1$ this allows us to compute $\tau(n)$.

**Tau** usually spends most of its time factoring $n$ (see 14.3.6).
Section 5. Miscellaneous

gap> Tau( 0 );
Error, Tau: <n> must not be 0 called from ...
... 4 lines omitted here ...
brk> quit;
gap> Tau( 1 ); Tau( 1013 ); Tau( 8128 ); Tau( 36 );
1
# thus 1013 is a prime
14
9
# the result is odd if and only if the argument is a perfect square

MoebiusMu( n )

MoebiusMu computes the value of Moebius inversion function for the 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 an even number and \(-1\) if \( n \) has an odd number of prime factors.

The importance of \( \mu \) stems from the so called inversion formula. Suppose \( f(n) \) 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 15.1.2).

MoebiusMu usually spends all of its time factoring \( n \) (see 14.3.6).

gap> MoebiusMu( 60 ); MoebiusMu( 61 ); MoebiusMu( 62 );
0
-1
1

15.5 Miscellaneous

TwoSquares( n )

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 \( ab \). 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 \( n/(ab) \), which must exist, and applies the Euclidean algorithm to \( r \) and \( n \). The first residues in the sequence that are smaller than \( \sqrt{n/(ab)} \) times \( c \) are a possible pair \( x \) and \( y \).

Better descriptions of the algorithm and related topics can be found in [Wag90] and [Zag90].

gap> TwoSquares( 5 ); TwoSquares( 11 );
[ 1, 2 ]
fail  # no representation exists
gap> TwoSquares( 16 ); TwoSquares( 45 );
[ 0, 4 ]
[ 3, 6 ]  # 3 is the minimal possible gcd because 9 divides 45
gap> TwoSquares( 125 ); TwoSquares( 13*17 );
[ 2, 11 ]  # not [ 5, 10 ] because this has not minimal gcd
[ 5, 14 ]  # [10,11] would be the other possible representation
gap> TwoSquares( 848654483879497562821 );
#I beyond the guaranteed bound of the probabilistic primality test
[ 6305894639, 28440994650 ]  # 848654483879497562821 is prime
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 58).

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.

```
gap> 2/3;
2/3
gap> 66/123;
22/41 # numerator and denominator are made relatively prime
gap> 17/-13;
-17/13 # the numerator carries the sign
gap> 121/11;
11 # rationals with denominator 1 (after cancelling) are integers
```

**Rationals**

```
gap> Size( Rationals ); 2/3 in Rationals;
infinity true
```

**IsRationals( obj )**

*Rationals* is a subset of **Cyclotomics**, so see chapter 18 for arithmetic operations and comparison of rationals. Functions for the field *Rationals* can be found in chapters 56 and 58.

**16.1 Elementary Operations for Rationals**

**IsRat( obj )**

Every rational number lies in the category **IsRat**, which is a subcategory of **IsCyc** (see 18).

```
gap> IsRat( 2/3 );
true
gap> IsRat( 17/-13 );
true
gap> IsRat( 11 );
true
gap> IsRat( IsRat );
false # 'IsRat' is a function, not a rational
```

**IsPosRat( obj )**

Every positive rational number lies in the category **IsPosRat**.
Every negative rational number lies in the category \texttt{IsNegRat}.

\texttt{NumeratorRat} returns the numerator of the rational \texttt{rat}. Because the numerator holds the sign of the rational it may be any integer. Integers are rationals with denominator 1, thus \texttt{NumeratorRat} is the identity function for integers.

\begin{verbatim}
gap> NumeratorRat( 2/3 ); 2
gap> NumeratorRat( 66/123 ); 22  # numerator and denominator are made relatively prime
gap> NumeratorRat( 17/-13 ); -17  # the numerator holds the sign of the rational
gap> NumeratorRat( 11 ); 11  # integers are rationals with denominator 1
\end{verbatim}

\texttt{DenominatorRat} returns the denominator of the rational \texttt{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 \texttt{DenominatorRat} returns 1 for integers.

\begin{verbatim}
gap> DenominatorRat( 2/3 ); 3
gap> DenominatorRat( 66/123 ); 41  # numerator and denominator are made relatively prime
gap> DenominatorRat( 17/-13 ); 13  # the denominator holds the sign of the rational
gap> DenominatorRat( 11 ); 1  # integers are rationals with denominator 1
\end{verbatim}

\texttt{Rat} returns a rational number \texttt{rat} whose meaning depends on the type of \texttt{elm}.

If \texttt{elm} is a string consisting of digits '0', '1', ..., '9' and '-' (at the first position), '/' and the decimal dot '.' then \texttt{rat} is the rational described by this string. The operation \texttt{String} (see 26.5.1) can be used to compute a string for rational numbers, in fact for all cyclotomics.

\begin{verbatim}
gap> Rat( "1/2" ); Rat( "35/14" ); Rat( "35/-27" ); Rat( "3.14159" );
1/2 5/2 -35/27 314159/100000
\end{verbatim}

\texttt{Random} for rationals returns pseudo random rationals which are the quotient of two random integers. See the description of \texttt{Random} for integers (14.1.11) for details. (Also see 28.6.1.)
This chapter describes the 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.

## 17.1 Combinatorial Numbers

### Factorial

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 $e^x$, which is the generating function for factorial.

```
gap> List( [0..10], Factorial );
[ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800 ]
gap> Factorial( 30 );
265252859812191058636308480000000
```

### Binomial

The **binomial coefficient** $\binom{n}{k}$ of integers $n$ and $k$, which is defined as $n! / (k!(n-k)!)$ (see 17.1.1).

We define $\binom{n}{0} = 1$, $\binom{n}{k} = 0$ 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 $\binom{n}{k}$, hence the name.

```
gap> List( [0..4], k->Binomial( 4, k ) );
[ 1, 4, 6, 4, 1 ] # Knuth calls this the trademark of Binomial

gap> PrintArray( last );
[ [ 1, 0, 0, 0, 0, 0, 0 ], # the lower triangle is
  [ 1, 1, 0, 0, 0, 0, 0 ], # called Pascal's triangle
  [ 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

$\text{NrCombinations}$ (see 17.2.1) is the generalization of $\text{Binomial}$ for multisets. $\text{Combinations}$ (see 17.2.1) computes the set of all combinations of a multiset.
Chapter 17. Combinatorics

3 \(\text{Bell}(n)\)

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 17.2.13). This implies of course that \(B(n) = \sum_{k=0}^{n} S_2(n,k)\) (see 17.1.6). \(B(n)/n!\) is the coefficient of \(x^n\) in the formal series \(e^{e^x - 1}\), which is the generating function for \(B(n)\).

\[\text{gap} \text{ } \text{List}([0..6], n \rightarrow \text{Bell}(n));\]
\[\text{1, 1, 2, 5, 15, 52, 203}\]
\[\text{gap} \text{ } \text{Bell}(14);\]
\[190899322\]

4 \(\text{Bernoulli}(n)\)

returns the \(n\)-th Bernoulli number \(B_n\), which is defined by \(B_0 = 1\) and \(B_n = -\sum_{k=0}^{n-1} \frac{(n+1)}{k} B_k/(n+1)\).

\(B_n/n!\) is the coefficient of \(x^n\) in the power series of \(x/e^x - 1\). Except for \(B_1 = -1/2\) the Bernoulli numbers for odd indices are zero.

\[\text{gap} \text{ } \text{Bernoulli}(4);\]
\[-1/30\]
\[\text{gap} \text{ } \text{Bernoulli}(10);\]
\[5/66\]
\[\text{gap} \text{ } \text{Bernoulli}(12);\]
\[-691/2730 \text{ } \# \text{ there is no simple pattern in Bernoulli numbers}\]
\[\text{gap} \text{ } \text{Bernoulli}(50);\]
\[495057205241079648212477525/66 \text{ } \# \text{ and they grow fairly fast}\]

5 \(\text{Stirling1}(n,k)\)

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 \(n!/x^n = \sum_{k=0}^{n} S_1(n,k)x^k\) which is the generating function for Stirling numbers of the first kind. Note the similarity to \(x^n = \sum_{k=0}^{n} S_2(n,k)k!/x^k\) (see 17.1.6). 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.

\[\text{gap} \text{ } \text{List}([0..4], k \rightarrow \text{Stirling1}(4,k));\]
\[\text{[0, 6, 11, 6, 1]} \text{ } \# \text{ Knuth calls this the trademark of S_1}\]
\[\text{gap} \text{ } \text{List}([0..6], n \rightarrow \text{List}([0..6], k \rightarrow \text{Stirling1}(n,k))));\]
\[\text{gap} \text{ } \text{PrintArray}(\text{last});\]
\[\text{[ [ [0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 1, 1, 0, 0, 0], [0, 2, 3, 1, 0, 0], [0, 6, 11, 6, 1, 0], [0, 24, 50, 35, 10, 1], [0, 120, 274, 225, 85, 15], [0, 1056, 2916, 2940, 1260, 210, 0] ] ]}\]
\[\text{gap} \text{ } \text{Stirling1}(50,10);\]
\[101623020926367490059043797119309944043405505380503665627365376\]
6. Stirling2( n, k )

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

\[
S_2(n,0) = 1, \quad S_2(0,0) = S_2(0,k) = 0 \quad \text{if} \quad n,k \neq 0.
\]

The recurrence relation is:

\[
S_2(n,k) = kS_2(n-1,k) + S_2(n-1,k-1).
\]

\( S_2(n,k) \) is the number of ways to partition a set of \( n \) elements into \( k \) pairwise disjoint nonempty subsets (see 17.2.13). Stirling numbers of the second kind appear as coefficients in the expansion of

\[
x^n = \sum_{k=0}^{n} \frac{S_2(n,k)}{k!} x^k.
\]

Note the similarity to

\[
\frac{n!}{x^n} = \sum_{k=0}^{n} \frac{S_1(n,k)}{k!} x^k
\]

(see 17.1.5). 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.

\[
gap> \text{List( [0..4], k->Stirling2( 4, k ) );}
\]

\[
[ 0, 1, 7, 6, 1 ] \quad \# \text{Knuth calls this the trademark of } S_2
\]

\[
gap> \text{List( [0..6], n->List( [0..6], k->Stirling2( n, k ) ) );}
\]

\[
\text{gap> PrintArray( last );}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 3 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 7 & 6 & 1 & 0 & 0 & 0 \\
0 & 1 & 15 & 25 & 10 & 1 & 0 & 0 \\
0 & 1 & 31 & 90 & 65 & 15 & 1 & 0
\end{array}
\]

\[
gap> \text{Stirling2( 50, 10 );}
\]

\[
26157615862881292012777396577993781727011
\]

17.2 Combinations, Arrangements and Tuples

1. Combinations( mset [, k ] )

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 \( \binom{|mset|}{k} \) combinations with \( k \) elements, and the set of all combinations is just the powerset of \( mset \), which contains all subsets of \( mset \) and has cardinality \( 2^{|mset|} \).

\[
gap> \text{Combinations( [1,2,2,3] );}
\]

\[
\begin{array}{cccccccc}
[ , ] , [ 1 ] , [ 1, 2 ] , [ 1, 2, 2 ] , [ 1, 2, 2, 3 ] , [ 1, 2, 3 ] , [ 1, 3 ] ,
\end{array}
\]

\[
gap> \text{NrCombinations( [1..52], 5 );}
\]

\[
2592800 \quad \# \text{number of different hands in a game of poker}
\]

The function Arrangements (see 17.2.3) computes ordered selections without repetitions, UnorderedTuples (see 17.2.5) computes unordered selections with repetitions and Tuples (see 17.2.7) computes ordered selections with repetitions.

2. NrCombinations( mset [, k ] )

returns the number of Combinations(\( mset, k \)).

\[
gap> \text{NrCombinations( [1..52], 5 );}
\]

\[
2592800 \quad \# \text{number of different hands in a game of poker}
\]

The function Arrangements (see 17.2.3) computes ordered selections without repetitions, UnorderedTuples (see 17.2.5) computes unordered selections with repetitions and Tuples (see 17.2.7) computes ordered selections with repetitions.

3. Arrangements( mset [, k ] )

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 17.1.1) arrangements with \( k \) elements.

4\begin{verbatim}
NrArrangements( mset [, k] )
\end{verbatim}
returns the number of \texttt{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 \texttt{settle} and you have to make a four letter word. Then the possibilities are given by

\begin{verbatim}
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" ],
# 96 more possibilities
[ "t", "t", "s", "e" ], [ "t", "t", "s", "l" ] ]
\end{verbatim}

Can you find the five proper English words, where \texttt{lets} does not count? Note that the fact that the list returned by \texttt{Arrangements} is a proper set means in this example that the possibilities are listed in the same order as they appear in the dictionary.

\begin{verbatim}
gap> NrArrangements( ["s","e","t","t","l","e"] );
523
\end{verbatim}

The function \texttt{Combinations} (see 17.2.1) computes unordered selections without repetitions, \texttt{UnorderedTuples} (see 17.2.5) computes unordered selections with repetitions and \texttt{Tuples} (see 17.2.7) computes ordered selections with repetitions.

5\begin{verbatim}
UnorderedTuples( set, k )
\end{verbatim}
returns the set of all unordered tuples of length \( k \) of the set \( set \).

An unordered tuple of length \( k \) of \( set \) is a 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 17.1.2) such unordered tuples.

Note that the fact that \texttt{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.

6\begin{verbatim}
NrUnorderedTuples( set, k )
\end{verbatim}
returns the number of \texttt{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..6]\]

\begin{verbatim}
gap> NrUnorderedTuples( [1..6], 5 );
252
\end{verbatim}

\begin{verbatim}
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 ],
# 99 more tuples
[ 1, 3, 4, 5, 6 ], [ 1, 3, 4, 6, 6 ], [ 1, 3, 5, 5, 5 ],
# 99 more tuples
[ 3, 3, 4, 4, 5 ], [ 3, 3, 4, 4, 6 ], [ 3, 3, 4, 5, 5 ],
# 39 more tuples
[ 5, 5, 6, 6, 6 ], [ 5, 6, 6, 6, 6 ], [ 6, 6, 6, 6, 6 ]]
\end{verbatim}
The function Combinations (see 17.2.1) computes unordered selections without repetitions, Arrangements (see 17.2.3) computes ordered selections without repetitions and Tuples (see 17.2.7) computes ordered selections with repetitions.

7 ▶ Tuples( set, k )
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.

8 ▶ NrTuples( set, k )
returns the number of Tuples(set, k).

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 21.20.14).
The function Combinations (see 17.2.1) computes unordered selections without repetitions, Arrangements (see 17.2.3) computes ordered selections without repetitions, and finally the function UnorderedTuples (see 17.2.5) computes unordered selections with repetitions.

9 ▶ PermutationsList( mset )
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 17.1.1) 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 $|mset|!/k_1!k_2!...$, which is sometimes called multinomial coefficient.

10 ▶ NrPermutationsList( mset )
returns the number of PermutationsList(mset).

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( [1,2,2,3,3,4,4,4,4] );
12600

The function Arrangements (see 17.2.3) is the generalization of PermutationsList that allows you to specify the size of the permutations. Derangements (see 17.2.11) computes permutations that have no fixpoints.

11 ▶ Derangements( list )
returns the set of all derangements of the list list.
A **derangement** is a fixpoint-free permutation of \( \text{list} \) and is represented by a list that contains exactly the same elements as \( \text{list} \), but in such an order that the derangement has at no position the same element as \( \text{list} \). If the list \( \text{list} \) contains no element twice there are exactly \( |\text{list}|!(1/2! - 1/3! + 1/4! - \cdots + (-1)^n/n!) \) derangements.

Note that the ratio \( \frac{\text{NrPermutationsList}([1..n])}{\text{NrDerangements}([1..n])} \), 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.

\[
\begin{align*}
\text{NrDerangements}(\text{list}) & \quad \text{returns the number of Derangements(list).} \\
\text{As an example of derangements suppose that you have to send four different letters to four different people. Then a derangement corresponds to a way to send those letters such that no letter reaches the intended person.}
\end{align*}
\]

\[
gap> \text{Derangements}([1,2,3,4]); \\
\begin{array}{l}
[ [ 2, 1, 4, 3 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 3, 1, 4, 2 ], \\
[ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 3, 1, 2 ], \\
[ 4, 3, 2, 1 ] ]
\end{array}
\]

\[
gap> \text{NrDerangements}([1..10]); \\
1334961
\]

\[
gap> \text{Int}(10^-7*\text{NrPermutationsList}([1..10])/\text{last}); \\
\begin{array}{r}
27182816
\end{array}
\]

The function **PermutationsList** (see 17.2.9) computes all permutations of a list.

\[
\begin{align*}
\text{PartitionsSet}(\text{set}, k) & \quad \text{returns the set of all unordered partitions of the set set into k pairwise disjoint nonempty sets. If k is not given it returns all unordered partitions of set for all k.} \\
\text{An unorderd partition of set is a set of pairwise disjoint nonempty sets with union set and is represented by a sorted list of such sets. There are } B(|\text{set}|) \text{ partitions of the set set and } S_2(|\text{set}|, k) \text{ partitions with k elements.}
\end{align*}
\]

\[
\begin{align*}
\text{NrPartitionsSet}(\text{set}, k) & \quad \text{returns the number of PartitionsSet(set,k).}
\end{align*}
\]

\[
\begin{align*}
\text{PartitionsSet}([1,2,3]); \\
\begin{array}{l}
[ [ 1 ], [ 2 ], [ 3 ] ], [ [ 1 ], [ 2, 3 ] ], [ [ 1, 2 ], [ 3 ] ], \\
[ [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 2 ] ]
\end{array}
\]

\[
\begin{align*}
\text{PartitionsSet}([1,2,3,4], 2); \\
\begin{array}{l}
[ [ 1 ], [ 2, 3, 4 ] ], [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 2, 3 ], [ 4 ] ], \\
[ [ 1, 2, 4 ], [ 3 ] ], [ [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 3, 4 ], [ 2 ] ], \\
[ [ 1, 4 ], [ 2, 3 ] ]
\end{array}
\end{align*}
\]

\[
\text{NrPartitionsSet([1..6])}; \\
\begin{array}{r}
203
\end{array}
\]

\[
\text{NrPartitionsSet([1..10], 3)};
\]

\[
\begin{align*}
\text{NrPartitionsSet}([1..10], 3); \\
\end{align*}
\]
Note that PartitionsSet does currently not support multisets and that there is currently no ordered counterpart.

\[ \text{Partitions}(n[,k]) \]
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 \( n \) 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 \( e^{\sqrt{2/\pi}} \sqrt{\frac{3}{n}} \) such partitions.

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 \( p(n) := \text{NrPartitions}(n) \) is the number of conjugacy classes of the symmetric group on \( n \) points.

Ramanujan found the identities \( p(5i + 4) = 0 \mod 5 \), \( p(7i + 5) = 0 \mod 7 \) and \( p(11i + 6) = 0 \mod 11 \) and many other fascinating things about the number of partitions.

Do not call Partitions with an \( n \) much larger than 40, in which case there are 37338 partitions, since the list will simply become too large.

\[ \text{NrPartitions}(n[,k]) \]
returns the number of Partitions\((set,k)\).

\[ \text{Partitions}(7) \]
\[ [ [ 1, 1, 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1, 1, 1 ], [ 2, 2, 1, 1, 1, 1 ], [ 3, 1, 1, 1, 1 ], [ 3, 2, 1, 1, 1 ], [ 3, 3, 1, 1 ], [ 4, 1, 1, 1 ], [ 4, 2, 1 ], [ 4, 3 ], [ 5, 1, 1 ], [ 5, 2 ], [ 6, 1 ], [ 7 ] ] \]

\[ \text{Partitions}(8,3) \]
\[ [ [ 3, 3, 2 ], [ 4, 2, 2 ], [ 4, 3, 1 ], [ 5, 2, 1 ], [ 6, 1, 1 ] ] \]

The function OrderedPartitions (see 17.2.17) is the ordered counterpart of Partitions.

\[ \text{OrderedPartitions}(n[,k]) \]
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 \( n \) for all \( k \).

An ordered partition is an ordered sum \( n = p_1 + p_2 + \cdots + p_k \) 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 17.1.2) ordered partitions with \( k \) summands.

Do not call OrderedPartitions with an \( n \) much larger than 15, the list will simply become too large.
18 ▶ NrOrderedPartitions( n [, k] )
returns the number of OrderedPartitions(set,k).

```
gap> NrOrderedPartitions( 5 );
524288
```

The function Partitions (see 17.2.15) is the unordered counterpart of OrderedPartitions.

19 ▶ PartitionsGreatestLE( n, m )
returns the set of all (unordered) partitions of the integer n having parts less or equal to the integer m.

```
gap> PartitionsGreatestLE( 6, 3 );
[ [ 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 ] ]
```

20 ▶ PartitionsGreatestEQ( n, m )
returns the set of all (unordered) partitions of the integer n having greatest part equal to the integer m.

```
gap> PartitionsGreatestEQ( 6, 3 );
```

21 ▶ RestrictedPartitions( n, set [, k] )
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 17.2.15) 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 \([1..n]\).

```
gap> RestrictedPartitions( 8, [1,3,5,7] );
[ [ 1, 1, 1, 1, 1, 1, 1 ], [ 3, 1, 1, 1, 1, 1 ], [ 5, 1, 1, 1 ], [ 7, 1 ] ]
```

22 ▶ NrRestrictedPartitions( n, set [, k] )
returns the number of RestrictedPartitions(n,set,k).

```
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.

23 ▶ SignPartition( pi )
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 40.3.1). Partitions of sign 1 are called even partitions while partitions of sign -1 are called odd.

```
gap> SignPartition([6,5,4,3,2,1]);
-1
```

24 ▶ AssociatedPartition( pi )
AssociatedPartition returns the associated partition of the partition \( pi \) which is obtained by transposing the corresponding Young diagram.
Section 3. Fibonacci and Lucas Sequences

\begin{verbatim}
gap> AssociatedPartition([4,2,1]);
[ 3, 2, 1, 1 ]
gap> AssociatedPartition([6]);
[ 1, 1, 1, 1, 1, 1 ]
\end{verbatim}

\begin{verbatim}
25\textbf{PowerPartition}( p\imath, k )
\end{verbatim}

\begin{verbatim}
\texttt{PowerPartition} \textit{returns the partition corresponding to the } k \textit{-th power of a permutation with cycle structure } p\imath. 
Each part \( l \) of \( p\imath \) is replaced by \( d = \gcd(l, k) \) parts \( l/d \). So if \( p\imath \) is a partition of \( n \) then \( p\imath^k \) also is a partition of \( n \). \texttt{PowerPartition} describes the powermap of symmetric groups.
\end{verbatim}

\begin{verbatim}
gap> PowerPartition([6,5,4,3,2,1], 3);
[ 5, 4, 2, 2, 2, 2, 1, 1, 1, 1 ]
\end{verbatim}

\begin{verbatim}
26\textbf{PartitionTuples}( n, r )
\end{verbatim}

\begin{verbatim}
\texttt{PartitionTuples} \textit{returns the list of all } r \textit{-tuples of partitions which together form a partition of } n. 
\texttt{r-}tuples of partitions describe the classes and the characters of wreath products of groups with \( r \) conjugacy classes with the symmetric group \( S_n \).
\end{verbatim}

\begin{verbatim}
gap> PartitionTuples(3, 2);
[ [ [ 1, 1, 1 ], [ ] ], [ [ 1, 1 ], [ 1 ] ], [ [ 1 ], [ 1, 1 ] ],
[ [ ], [ 1, 1, 1 ] ], [ [ 2, 1 ], [ ] ], [ [ 1 ], [ 2 ] ],
\end{verbatim}

17.3 Fibonacci and Lucas Sequences

\begin{verbatim}
1\textbf{Fibonacci}( n )
\end{verbatim}

\begin{verbatim}
\texttt{Fibonacci} \textit{returns the } n \textit{th number of the } \textbf{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 \( \phi = (\sqrt{5} + 1)/2 \), 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 54.7.1) than any other pair of integers of the same size. \texttt{Fibonacci}(k) is the special case \texttt{Lucas}(1,-1,k)[1] (see 17.3.2).
\end{verbatim}

\begin{verbatim}
gap> Fibonacci( 10 );
55
\end{verbatim}

\begin{verbatim}
2\textbf{Lucas}( P, Q, k )
\end{verbatim}

\begin{verbatim}
\texttt{Lucas} \textit{returns the } k \textit{-th values of the } \textbf{Lucas sequence} with parameters \( P \) and \( Q \), which must be integers, as a list of three integers.

Let \( \alpha, \beta \) be the two roots of \( x^2 - Px + Q \) then we define \( \texttt{Lucas}(P, Q, k)[1] = U_k = (\alpha^k - \beta^k)/(\alpha - \beta) \) and \( \texttt{Lucas}(P, Q, k)[2] = V_k = (\alpha^k + \beta^k) \) and as a convenience \( \texttt{Lucas}(P, Q, k)[3] = Q^k \).
\end{verbatim}

\begin{verbatim}
gap> Lucas( 1, -1, 10 );
9227465

\gap> Lucas( 1, -1, -10 );
-55
\end{verbatim}
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 $V_{2k} = V_k^2 - 2Q^k, V_{2k+1} = ((P^2 - 4Q)V_{2k} + PV_{2k})/2$.

Fibonacci$(k)$ (see 17.3.1) 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.

\[
\begin{align*}
\text{gap> } & \text{List( } [0..10], i->\text{Lucas}(1,-2,i)[1] \text{ ); } \\
& [ 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341 ] \quad \# 2^k - (-1)^k/3 \\
\text{gap> } & \text{List( } [0..10], i->\text{Lucas}(1,-2,i)[2] \text{ ); } \\
& [ 2, 1, 5, 7, 17, 31, 65, 127, 257, 511, 1025 ] \quad \# 2^k + (-1)^k \\
\text{gap> } & \text{List( } 0..10, i->\text{Lucas}(1,-1,i)[1] \text{ ); } \\
& [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ] \quad \# \text{Fibonacci sequence} \\
\text{gap> } & \text{List( } 0..10, i->\text{Lucas}(2,1,i)[1] \text{ ); } \\
& [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ] \quad \# \text{the roots are equal}
\end{align*}
\]

17.4 Permanent of a Matrix

\[
\text{\textbf{Permanent}}( \text{ mat })
\]

returns the permanent of the matrix mat. The permanent is defined by $\sum_{p \in \text{Symm}(n)} \prod_{i=1}^n \text{mat}[i][i^p]$. Note the similarity of the definition of the permanent to the definition of the determinant (see 24.3.4). 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.

\[
\begin{align*}
\text{gap> } & \text{Permanent( } [[0,1,1,1], \\
& > [1,0,1,1], \\
& > [1,1,0,1], \\
& > [1,1,1,0]] \text{ ); } \\
9 & \quad \# \text{inefficient way to compute } \text{'NrDerangements([1..4])'} \\
\text{gap> } & \text{Permanent( } [[1,1,0,1,0,0,0], \\
& > [0,1,1,0,1,0,0], \\
& > [0,0,1,1,0,1,0], \\
& > [0,0,0,1,1,0,1], \\
& > [1,0,0,0,1,1,0], \\
& > [0,1,0,0,0,1,1], \\
& > [1,0,1,0,0,0,1]] \text{ ); } \\
24 & \quad \# 24 \text{ permutations fit the projective plane of order 2}
\end{align*}
\]
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 \( e_n = e^{2\pi i/n} \) is a primitive complex \( n \)-th root of unity. The elements of these fields are called cyclotomics.

Information concerning operations for fields of cyclotomics, for example certain integral bases, can be found in Chapter 58. For more general operations that take a field extension as a —possibly optional— argument, e.g., Trace or Coefficients, see Chapter 56.

### 18.1 Operations for Cyclotomics

\( \text{E( } n \text{ )} \)

\( \text{E} \) returns the primitive \( n \)-th root of unity \( e_n = e^{2\pi i/n} \). Cyclotomics are usually entered as sums of roots of unity, with rational coefficients, and irrational cyclotomics are displayed in the same way. (For special cyclotomics, see 18.4.)

\[
\text{gap}\> \text{E(9); E(9)^3; E(6); E(12) / 3;}
\]

\[
-E(9)^4-E(9)^7
\]

\[
E(3)
\]

\[
-E(3)^2
\]

\[
-1/3\cdot E(12)^7
\]

A particular basis is used to express cyclotomics, see 58.1; note that \( \text{E(9)} \) is not a basis element, as the above example shows.

\( \text{Cyclotomics} \)

\( \text{V} \) is the field of all cyclotomics (in GAP).

\[
\text{gap}\> \text{E(9) in Cyclotomics; 37 in Cyclotomics; true in Cyclotomics;}
\]

\[
\text{true}
\]

\[
\text{true}
\]

\[
\text{false}
\]

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 cyclotomic is a Galois automorphism if and only if \( k \) is coprime to the conductor of the cyclotomic.
Chapter 18. Cyclotomic Numbers

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

Every object in the family CyclotomicsFamily lies in the category IsCyclotomic. This covers integers, rationals, proper cyclotomics, the object infinity (see 18.2.1), and unknowns (see Chapter 19). All these objects except infinity and unknowns lie also in the category IsCyc, infinity lies in (and can be detected from) the category IsInfinity, and unknowns lie in IsUnknown.

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

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 58.1), 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.

gap> r:= ER( 5 ); # The square root of 5 is a cyclotomic integer.
E(5)-E(5)^2-E(5)^3+E(5)^4
gap> IsIntegralCyclotomic( r ); # It has integral coefficients.
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

The operation Int can be used to find a cyclotomic integer near to an arbitrary cyclotomic. For rationals, Int returns the largest integer smaller or equal to the argument.
Section 1. Operations for Cyclotomics

gap> Int( E(5)+1/2*E(5)^2 ); Int( 2/3*E(7)+3/2*E(4) );
E(5)
E(4)

The operation \texttt{String} returns for a cyclotomic a string corresponding to the way the cyclotomic is printed by \texttt{ViewObj} and \texttt{PrintObj}.

\begin{verbatim}
gap> String( E(5)+1/2*E(5)^2 ); String( 17/3 );
"E(5)+1/2*E(5)^2"
"17/3"
\end{verbatim}

\textbf{Conductor( cyc )}

\textbf{Conductor( C )}

For an element \textit{cyc} of a cyclotomic field, \textbf{Conductor} returns the smallest integer \textit{n} such that \textit{cyc} is contained in the \textit{n}-th cyclotomic field. For a collection \textit{C} of cyclotomics (for example a dense list of cyclotomics or a field of cyclotomics), \textbf{Conductor} returns the smallest integer \textit{n} such that all elements of \textit{C} are contained in the \textit{n}-th cyclotomic field.

\begin{verbatim}
gap> Conductor( 0 ); Conductor( E(10) ); Conductor( E(12) );
1
5
12
\end{verbatim}

\textbf{AbsoluteValue( cyc )}

returns the absolute value of a cyclotomic number \textit{cyc}. At the moment only methods for rational numbers exist.

\begin{verbatim}
gap> AbsoluteValue(-3);
3
\end{verbatim}

\textbf{RoundCyc( cyc )}

is a cyclotomic integer \textit{z} (see 18.1.4) near to the cyclotomic \textit{cyc} in the sense that the \textit{i}-th coefficient in the external representation (see 18.1.8) of \textit{z} is \texttt{Int( c+1/2 )} where \textit{c} is the \textit{i}-th coefficient in the external representation of \textit{cyc}. Expressed in terms of the Zumbroich basis (see 58.1), the coefficients of \textit{cyc} w.r.t. this basis are rounded.

\begin{verbatim}
gap> RoundCyc( E(5)+1/2*E(5)^2 ); RoundCyc( 2/3*E(7)+3/2*E(4) );
E(5)+E(5)^2
\end{verbatim}

\textbf{CoeffsCyc( cyc, N )}

Let \textit{cyc} be a cyclotomic with conductor \textit{n}. If \textit{N} is not a multiple of \textit{n} then \textbf{CoeffsCyc} returns \texttt{fail} because \textit{cyc} cannot be expressed in terms of \textit{N}-th roots of unity. Otherwise \textbf{CoeffsCyc} returns a list of length \textit{N} with entry at position \textit{j} equal to the coefficient of \texttt{e^{2\pi i (j-1)/N}} if this root belongs to the \textit{N}-th Zumbroich basis (see 58.1), and equal to zero otherwise. So we have \textit{cyc} = \textbf{CoeffsCyc}(\textit{cyc}, \textit{N}) \cdot \textbf{List}([1..\textit{N}], \textit{j} \rightarrow E(\textit{N})^{(j-1)}) .

\begin{verbatim}
gap> cyc:= E(5)+E(5)^2;
E(5)+E(5)^2
\end{verbatim}

\begin{verbatim}
gap> CoeffsCyc( cyc, 5 ); CoeffsCyc( cyc, 15 ); CoeffsCyc( cyc, 7 );
[ 0, 1, 1, 0, 0 ]
\end{verbatim}

\begin{verbatim}
[ 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, -1, 0, -1, 0 ]
\end{verbatim}

\texttt{fail}
Chapter 18. Cyclotomic Numbers

For a cyclotomic number \( \text{cyc} \) (see 18.1.3), this function returns the smallest positive integer \( n \) such that \( n \ast \text{cyc} \) is a cyclotomic integer (see 18.1.4). For rational numbers \( \text{cyc} \), the result is the same as that of \text{DenominatorRat} \) (see 16.1.5).

\[
\begin{align*}
\text{gap}> \text{ExtRep} \text{OfObj}( \text{E}(5) ); \text{CoeffsCyc}( \text{E}(5), 15 ); \\
\text{[ 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1, 0 ]} \\
\text{fail}
\end{align*}
\]

Given a cyclotomic \( \text{root} \) that is known to be a root of unity (this is not checked), \text{DescriptionOfRootOfUnity} \) returns a list \([n, e]\) of coprime positive integers such that \( \text{root} = \text{E}(n)^e \) holds.

\[
\begin{align*}
\text{gap}> \text{E}(9); \text{DescriptionOfRootOfUnity}( \text{E}(9) ); \\
-\text{E}(9)^4 - \text{E}(9)^7 \\
\text{[ 9, 1 ]} \\
\text{gap}> \text{DescriptionOfRootOfUnity}( -\text{E}(3) ); \\
\text{[ 6, 5 ]}
\end{align*}
\]

\text{IsGaussInt} \) returns \text{true} if the object \( x \) is a Gaussian integer and \text{false} otherwise. Gaussian integers are of the form \( a + b \ast \text{E}(4) \), where \( a \) and \( b \) are integers.

\[
\begin{align*}
\text{gap}> \text{Field}( \text{E}(5) + \text{E}(5)^4 ); \text{DefaultField}( \text{E}(5) + \text{E}(5)^4 ); \\
\text{NF}(5, [ 1, 4 ]); \text{CF}(5)
\end{align*}
\]

18.2 Infinity

\text{IsInfinity} \) is a special \text{GAP} object that lies in \text{CyclotomicsFamily}. It is larger than all other objects in this family. \text{Infinity} is mainly used as return value of operations such as \text{Size} and \text{Dimension} for infinite resp. infinite dimensional domains.

Note that no arithmetic operations are provided for \text{infinity}, in particular there is no problem to define what \( 0 \ast \text{infinity} \) or \( \text{infinity} - \text{infinity} \) means.

Often it is useful to distinguish \text{infinity} from “proper” cyclotomics. For that, \text{infinity} lies in the category \text{IsInfinity} but not in \text{IsCyc}, and the other cyclotomics lie in the category \text{IsCyc} but not in \text{IsInfinity}.
Section 4. ATLAS Irrationalities

\begin{verbatim}
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 ]
\end{verbatim}

18.3 Comparisons of Cyclotomics

To compare cyclotomics, the operators $<$, $<=$, $=$, $>=$, $>$ and $<$ can be used, the result will be \texttt{true} if the first operand is smaller, smaller or equal, equal, larger or equal, larger, or unequal, respectively, and \texttt{false} otherwise.

Cyclotomics are ordered as follows: The relation between rationals is the natural one, rationals are smaller than irrational cyclotomics, and \texttt{infinity} is the largest cyclotomic. For two irrational cyclotomics with different conductors, the one with smaller conductor is regarded as smaller. Two irrational cyclotomics with same conductor are compared via their external representation.

For comparisons of cyclotomics and other \texttt{GAP} objects, see Section 4.11.

\begin{verbatim}
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
\end{verbatim}

18.4 ATLAS Irrationalities

\begin{verbatim}
1 ▶ EB( n )
▶ EC( n )
▶ ED( n )
▶ EE( n )
▶ EF( n )
▶ EG( n )
▶ EH( n )
\end{verbatim}

For $N$ a positive integer, let $z = E(N) = \exp(2\pi i / N)$. The following so-called atomic irrationalities (see Chapter 7, Section 10 of [CCN+85]) can be entered using functions. (Note that the values are not necessary irrational.)
We define

From a theorem of Gauss we know that is installed as method for the operation \( n \in \mathbb{Z} \). For given embedding of cyclotomic fields into the complex numbers, \( \operatorname{ER}(N) \) can be computed from \( b_N \). By the chosen embedding of cyclotomic fields into the complex numbers, \( \operatorname{ER} \) returns the positive square root if \( N \) is positive, and if \( N \) is negative then \( \operatorname{ER}(N) = \sqrt{-N} \). In any case, \( \operatorname{EI}(N) = \sqrt{N} \).

\[ b_N = \begin{cases} \frac{1}{2}(-1 + \sqrt{N}) & \text{if } N \equiv 1 \pmod{4} \\ \frac{1}{2}(-1 + i\sqrt{N}) & \text{if } N \equiv -1 \pmod{4} \end{cases} \]

So \( \sqrt{N} \) can be computed from \( b_N \) (see 18.4.1).

For given \( N \), let \( n_k = n_k(N) \) be 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

\[ \begin{align*}
\operatorname{EY}(N) &= y_n = z + z^n & (n = n_2) \\
\operatorname{EX}(N) &= x_n = z + z^n + z^{n^2} & (n = n_3) \\
\operatorname{EW}(N) &= u_n = z + z^n + z^{n^2} + z^{n^3} & (n = n_4) \\
\operatorname{EV}(N) &= v_n = z + z^n + z^{n^2} + z^{n^3} + z^{n^4} & (n = n_5) \\
\operatorname{EU}(N) &= u_n = z + z^n + z^{n^2} + \ldots + z^{n^k} & (n = n_6) \\
\operatorname{ET}(N) &= t_n = z + z^n + z^{n^2} + \ldots + z^{n^k} & (n = n_7) \\
\operatorname{ES}(N) &= s_n = z + z^n + z^{n^2} + \ldots + z^{n^k} & (n = n_8)
\end{align*} \]

\[ \begin{align*}
\operatorname{EM}(N) &= m_n = z - z^n & (n = n_2) \\
\operatorname{EL}(N) &= l_n = z - z^n + z^{n^2} - z^{n^3} & (n = n_4) \\
\operatorname{EK}(N) &= k_n = z - z^n + \ldots - z^{n^k} & (n = n_6) \\
\operatorname{EJ}(N) &= j_n = z - z^n + \ldots - z^{n^k} & (n = n_8)
\end{align*} \]
Let $n_k(d) = n_k^d(N)$ be the $d + 1$-th integer with multiplicative order exactly $k$ modulo $N$, chosen in the order of preference defined above; we write $n_k = n_k^0, n_k' = n_k^{(1)}, n_k'' = n_k^{(2)}$ and so on. These values can be computed as $\text{NK}(N,k,d) = n_k^d(N)$; if there is no integer with the required multiplicative order, $\text{NK}$ returns fail.

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 above definitions by $n'_k, n''_k, \ldots$; they can be entered as $\text{EY}(N,d) = y_N^{(d)}$ and $\text{EX}(N,d) = x_N^{(d)}$.

To explain the “ampersand” syntax in general we remark that “&k” is interpreted as $q^*k_N$, 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 $*k$.

```plaintext
gap> EW(16,3); EW(17,2); ER(3); EI(3); EY(5); EB(9);
0
E(17)+E(17)^4+E(17)^13+E(17)^16
-E(12)^7+E(12)^11
E(3)-E(3)^2
E(5)+E(5)^4
1
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*13-8&5" );
-3*E(24)-E(24)^11+E(24)^17+3*E(24)^19
gap> AtlasIrrationality("3y'''24*13-4&8&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

For a cyclotomic \( \text{cyc} \) and an integer \( k \), \text{GaloisCyc} returns the cyclotomic obtained by raising the roots of unity in the Zumbroich basis representation of \( \text{cyc} \) to the \( k \)-th power. If \( k \) is coprime to the integer \( n \), \text{GaloisCyc} acts as a Galois automorphism of the \( n \)-th cyclotomic field (see 58.3); to get the Galois automorphisms themselves, use \text{GaloisGroup} (see 56.3.1).

The complex conjugate of \( \text{cyc} \) is \( \text{GaloisCyc}( \text{cyc}, -1) \), which can also be computed using \text{ComplexConjugate} (see 58.2.4).

For a list or matrix \( \text{list} \) of cyclotomics, \text{GaloisCyc} returns the list obtained by applying \text{GaloisCyc} to the entries of \( \text{list} \).

\[
\text{gap} \text{GaloisCyc}( E(5) + E(5)^4, 2 ); \\
\text{gap} \text{GaloisCyc}( E(5), -1 ); \quad \# \text{the complex conjugate}
\]

\[
\text{gap} \text{GaloisCyc}( E(5) + E(5)^4, -1 ); \quad \# \text{this value is real}
\]

\[
\text{gap} \text{GaloisCyc}( \text{E}(15) + \text{E}(15)^4, 3 ); \\
\text{gap} \text{ComplexConjugate}( \text{E}(7) );
\]

\[
\text{gap} \text{GaloisCyc}( \text{E}(5) + \text{E}(5)^4, 2 ); \\
\text{gap} \text{GaloisCyc}( \text{E}(5), -1 ); \quad \# \text{the complex conjugate}
\]

2▶ \text{StarCyc}( \text{cyc} )

If the cyclotomic \( \text{cyc} \) is an irrational element of a quadratic extension of the rationals then \text{StarCyc} returns the unique Galois conjugate of \( \text{cyc} \) that is different from \( \text{cyc} \), otherwise \text{fail} is returned. In the first case, the return value is often called \( \text{cyc}^* \) (see 69.11).

\[
\text{gap} \text{StarCyc}( \text{EB}(5) ); \text{StarCyc}( \text{E}(5) );
\]

\[
\text{gap} \text{StarCyc}( \text{E}(5) ); \text{StarCyc}( \text{E}(5) ); \\
\text{fail}
\]

3▶ \text{Quadratic}( \text{cyc} )

Let \( \text{cyc} \) be a cyclotomic integer that lies in a quadratic extension field of the rationals. Then we have \( \text{cyc} = (a + b\sqrt{n})/d \) for integers \( a, b, n, d \), such that \( d \) is either 1 or 2. In this case, \text{Quadratic} returns a record with the components \( a, b, \text{root}, d, \text{ATLAS}, \text{display} \); the values of the first four are \( a, b, n, \text{and} d, \text{the ATLAS value is a (not necessarily shortest) representation of \( \text{cyc} \) in terms of the ATLAS irrationalities \( b_{|n|}, h_{|n|}, \text{display} \), and the display value is a string that expresses \( \text{cyc} \) in GAP notation, corresponding to the value of the ATLAS component.

If \( \text{cyc} \) is not a cyclotomic integer or does not lie in a quadratic extension field of the rationals then \text{fail} is returned.

If the denominator \( d \) is 2 then necessarily \( n \) is congruent to 1 modulo 4, and \( r_a, i_a \) are not possible; we have \( \text{cyc} = x + y \cdot \text{EB( root )} \) with \( y = b, x = (a + b) / 2 \).

If \( d = 1 \), we have the possibilities \( i_{|n|} \) for \( n < -1 \), \( a + b \cdot i \) for \( n = -1 \), \( a + b \cdot r_a \) for \( n > 0 \). Furthermore if \( n \) is congruent to 1 modulo 4, also \( \text{cyc} = (a + b) + 2 \cdot b \cdot h_{|n|} \) is possible; the shortest string of these is taken as the value for the component \text{ATLAS}. 

```
Section 5. Galois Conjugacy of Cyclotomics

Let \( mat \) be a matrix of cyclotomics. \texttt{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.

\texttt{GaloisMat} returns a record with the components \( mat \), \texttt{galoisfams}, and \texttt{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.

\( \texttt{galoisfams} \):

a list that has the same length as the \( \texttt{mat} \) component, its entries are either 1, 0, -1, or lists. \( \texttt{galoisfams}[i] = 1 \) means that \( \texttt{mat}[i] \) consists of rationals, i.e. \([ \texttt{mat}[i] \]) forms an orbit; \( \texttt{galoisfams}[i] = -1 \) means that \( \texttt{mat}[i] \) contains unknowns (see Chapter 19); in this case \( \texttt{mat}[i] \) is regarded as an orbit, too, even if \( \texttt{mat}[i] \) contains irrational entries; if \( \texttt{galoisfams}[i] = [l_1, l_2] \) is a list then \( \texttt{mat}[i] \) is the first element of its orbit in \( \texttt{mat} \), \( \texttt{l_1} \) is the list of positions of rows that form the orbit, and \( \texttt{l_2} \) is the list of corresponding Galois automorphisms (as exponents, not as functions), so we have \( \texttt{mat[l_1][j]}[k] = \texttt{GaloisCyc(mat[i][k], l_2[j])} \); \( \texttt{galoisfams}[i] = 0 \) means that \( \texttt{mat}[i] \) is an element of a nontrivial orbit but not the first element of it.

\( \texttt{generators} \):

a list of permutations generating the permutation group corresponding to the action of the Galois group on the rows of \( \texttt{mat} \).

In the following example we temporarily increase the line length limit from its default value 80 to 84 in order to get a nicer output format.

```plaintext
gap> Quadratic( EB(5) ); Quadratic( EB(27) );
rec( a := -1, b := 1, root := 5, d := 2, ATLAS := "b5",
display := "(-1+ER(5))/2" )
rec( a := -1, b := 3, root := -3, d := 2, ATLAS := "1+3b3",
display := "(-1+3*ER(-3))/2" )
gap> Quadratic(0); Quadratic( E(5) );
rec( a := 0, b := 0, root := 1, d := 1, ATLAS := "0", display := "0" )
fail
```

\( \texttt{GaloisMat} \) A

returns the list of rationalized rows of \( \texttt{mat} \), 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 \( \texttt{mat} \) (see 18.5.4), so the operation may be viewed as a kind of trace on the rows.
Note that no two rows of \textit{mat} should be equal.

\begin{verbatim}
gap> mat:=List(Irr(CharacterTable( "A5" )),ValuesOfClassFunction);
[ [ 1, 1, 1, 1, 1 ], [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
gap> RationalizedMat( mat );
[ [ 1, 1, 1, 1, 1 ], [ 6, -2, 0, 1, 1 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
\end{verbatim}

18.6 Internally Represented Cyclotomics

The implementation of an \textit{internally represented cyclotomic} is based on a list of length equal to its conductor. This means that the internal representation of a cyclotomic does \textbf{not} refer to the smallest number field but the smallest \textit{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 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 69).

For the representation of cyclotomics one has to recall that the $n$-th cyclotomic field $\mathbb{Q}(\zeta_n)$ is a vector space of dimension $\varphi(n)$ over the rationals where $\varphi$ denotes Euler’s phi-function (see 15.1.2).

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 \texttt{GAP}, see 58.1 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 \textbf{not} the multiplication of the group ring $\mathbb{Q}[\zeta_n]$; given a $\mathbb{Q}$-basis of $\mathbb{Q}(\zeta_n)$ the result of the multiplication (computed as multiplication of polynomials in $\zeta_n$, using $(\zeta_n)^n = 1$) will be converted to the basis.

\begin{verbatim}
gap> E(5) * E(5)^2; ( E(5) + E(5)^4 ) * E(5)^2;
E(5)^3
E(5)+E(5)^3

E(5)^3
E(5)+E(5)^3

gap> ( E(5) + E(5)^4 ) * E(5); -E(5)-E(5)^3-E(5)^4
\end{verbatim}

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 \texttt{ExtRepOfObj}.

Since the conductor of internally represented cyclotomics must be in the category \texttt{IsSmallIntRep}, the biggest possible (though not very useful) conductor is 65535. So the maximal cyclotomic field implemented in \texttt{GAP} is not really the field $\mathbb{Q}_{ab}$. 
19

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 a character of a group sill be induced to a supergroup (see 70.9.3) but the class fusion is only a parametrized map (see Chapter 71), 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 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.

In the first form Unknown returns a new unknown value, i.e., the first one that is larger than all unknowns which exist in the current GAP session.

In the second form Unknown returns the n-th unknown; if it did not exist already, it is created.

LargestUnknown is the largest n that is used in any Unknown(n) in the current GAP session. This is used in Unknown which increments this value when asked to make a new unknown.

IsUnknown is the category of unknowns in GAP.

```
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
Unkn0wns can be compared via = and < with all cyclotomics and with certain other GAP objects (see 4.11). We have $\text{Unknown}(n) \geq \text{Unknown}(m)$ if and only if $n \geq m$ holds; unknowns are larger than all cyclotomics that are not unknowns.

```
gap> IsUnknown( Unknown );  IsUnknown( Unknown() );
false
true
```

The usual arithmetic operations $+$, $-$, $\ast$ 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. \textbf{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 24.1.2).
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 and IsBool. 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 20.1.1) is regarded as a boolean.

```c
IsBool( obj )
```

tests whether obj is true, false or fail.

```gap
gap> IsBool( true );  IsBool( false );  IsBool( 17 );
true
true
false
```

## 20.1 Fail

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 (see 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 20.3 below), and fail cannot appear in boolean lists (see Chapter 22).

## 20.2 Comparisons of Booleans

```gap
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 and 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.
The ordering of boolean values is defined by \texttt{true} < \texttt{false} < \texttt{fail}. For the comparison of booleans with other GAP objects, see Section 4.11.

\begin{verbatim}
gap> true < false; fail
true

\end{verbatim}

\section*{20.3 Operations for Booleans}

The following boolean operations are only applicable to \texttt{true} and \texttt{false}.

\subsection*{1\quad \texttt{bool1} or \texttt{bool2}}

The logical operator \texttt{or} evaluates to \texttt{true} if at least one of the two boolean operands \texttt{bool1} and \texttt{bool2} is \texttt{true} and to \texttt{false} otherwise.

\texttt{or} first evaluates \texttt{bool1}. If the value is neither \texttt{true} nor \texttt{false} an error is signalled. If the value is \texttt{true}, then \texttt{or} returns \texttt{true} without evaluating \texttt{bool2}. If the value is \texttt{false}, then \texttt{or} evaluates \texttt{bool2}. Again, if the value is neither \texttt{true} nor \texttt{false} an error is signalled. Otherwise \texttt{or} returns the value of \texttt{bool2}. This \texttt{short-circuited} evaluation is important if the value of \texttt{bool1} is \texttt{true} and evaluation of \texttt{bool2} would take much time or cause an error.

\texttt{or} is associative, i.e., it is allowed to write \texttt{b1 or b2 or b3}, which is interpreted as \((\texttt{b1 or b2}) or b3\). \texttt{or} has the lowest precedence of the logical operators. All logical operators have lower precedence than the comparison operators \(=, <, in\), etc.

\begin{verbatim}
gap> true or false;
true

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
\end{verbatim}

\subsection*{2\quad \texttt{bool1} and \texttt{bool2}}

The logical operator \texttt{and} evaluates to \texttt{true} if both boolean operands \texttt{bool1} and \texttt{bool2} are \texttt{true} and to \texttt{false} otherwise.

\texttt{and} first evaluates \texttt{bool1}. If the value is neither \texttt{true} nor \texttt{false} an error is signalled. If the value is \texttt{false}, then \texttt{and} returns \texttt{false} without evaluating \texttt{bool2}. If the value is \texttt{true}, then \texttt{and} evaluates \texttt{bool2}. Again, if the value is neither \texttt{true} nor \texttt{false} an error is signalled. Otherwise \texttt{and} returns the value of \texttt{bool2}. This \texttt{short-circuited} evaluation is important if the value of \texttt{bool1} is \texttt{false} and evaluation of \texttt{bool2} would take much time or cause an error.

\texttt{and} is associative, i.e., it is allowed to write \texttt{b1 and b2 and b3}, which is interpreted as \((\texttt{b1 and b2}) and b3\). \texttt{and} has higher precedence than the logical \texttt{or} operator, but lower than the unary logical \texttt{not} operator. All logical operators have lower precedence than the comparison operators \(=, <, in\), etc.
Section 3. Operations for Booleans

\begin{verbatim}
gap> true and false;
false
gap> true and true;
true
gap> false and 17; # this does not cause an error, because '17' is never looked at
false

3\textbf{\large and} \textbf{fil1} \textbf{\large and} \textbf{fil2}

and can also be applied to filters. It returns a filter that when applied to some argument \(x\), tests \(\text{fil1}(x)\) and \(\text{fil2}(x)\).

\begin{verbatim}
gap> andfilt:= IsPosRat and IsInt;;
gap> andfilt( 17 ); andfilt( 1/2 );
true
false
\end{verbatim}

4\textbf{\large not} \textbf{bool}

The logical operator \textbf{not} returns \textbf{true} if the boolean value \textit{bool} is \textbf{false} and \textbf{true} otherwise. An error is signalled if \textit{bool} does not evaluate to \textbf{true} or \textbf{false}.

\textbf{not} has higher precedence than the other logical operators, \textbf{or} and \textbf{and}. All logical operators have lower precedence than the comparison operators \(=\), \(<\), \(\in\), etc.

\begin{verbatim}
gap> true and false;
false
gap> not true;
false
gap> not false;
true
\end{verbatim}
\end{verbatim}
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 28 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 26), 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 and Random, will be described in Chapter 28, in particular see 28.2.

### 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 [].

\[
\text{gap> } [1, 2, 3]; \\
\text{[1, 2, 3]} \quad \text{# a list with three elements} \\
\text{gap> } [[], [1], [1, 2]]; \\
\text{[[], [1], [1, 2]]} \quad \text{# a list may contain other lists}
\]

Each list constructed this way is mutable (see 12.6).

1. **IsList** (obj) C
   
   tests whether obj is a list.

   \[
   \text{gap> } \text{IsList( [1, 3, 5, 7] )}; \quad \text{IsList( 1 )}; \\
   \text{true} \\
   \text{false}
   \]

2. **IsDenseList** (obj) C
   
   A list is dense if it has no holes, i.e., contains an element at every position. 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.
Section 1. List Categories

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 6.3). After assigning a value and typing return;, GAP is finally able to comply with our request (by responding with 16).

3 ▶ IsHomogeneousList( obj )

returns true if obj is a list and it is homogeneous, or 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 28), a nonempty homogeneous list is also a collection.

gap> IsHomogeneousList( [ 1, 2, 3 ] ); IsHomogeneousList( [] );
true
true
gap> IsHomogeneousList( [ 1, false, () ] );
false

4 ▶ IsTable( obj )

A table is a nonempty list of homogeneous lists which lie in the same family. Typical examples of tables are matrices (see 24).

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

5 ▶ IsConstantTimeAccessList( list )

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 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 \texttt{list} in the category \texttt{IsList} (see 21.1.1) is regarded as a list, and if methods for the basic list operations are installed for \texttt{list} then \texttt{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 21.17.6).

\begin{verbatim}
1 ▶ \( \texttt{list[pos]} \) O
2 ▶ \texttt{IsBound\( \texttt{list[pos]} \) O}
3 ▶ \( \texttt{list[pos] := val} \) O
4 ▶ \texttt{Unbind\( \texttt{list[pos]} \) O}
\end{verbatim}

These operations implement element access, test for element boundedness, list element assignment, and removal of the element at position \texttt{pos}. In all cases, the index \texttt{pos} must be a positive integer.

Note that the special characters \texttt{[}, \texttt{]}, \texttt{:,} and \texttt{=} must be escaped with a backslash \texttt{\(} (see 4.3); so \texttt{\[\]} denotes the operation for element access in a list, whereas \texttt{[]} denotes an empty list. (Maybe the variable names involving special characters look strange, but nevertheless they are quite suggestive.)

\( \texttt{\[\]} \) is equivalent to \texttt{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, \texttt{IsBound\[\]} is used explicitly mainly in method installations. In other situations, one can simply call \texttt{IsBound}, which then delegates to \texttt{IsBound\[\]} if the first argument is a list, and to \texttt{IsBound\}. if the first argument is a record.

Analogous statements hold for \texttt{\[\]} and \texttt{Unbind\[\].}

21.3 List Elements

\begin{verbatim}
1 ▶ \texttt{list[pos]}
\end{verbatim}

The above construct evaluates to the \texttt{pos}-th element of the list \texttt{list}. \texttt{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.

\begin{verbatim}
gap> l := [ 2, 3, 5, 7, 11, 13 ];; l[1]; l[2]; l[6];
gap> l := [ 2, 3, 5, 7, 11, 13 ];; l[1]; l[2]; l[6];
gap> l[1]; l[2]; l[6];
2
3
13
\end{verbatim}

If \texttt{list} is not a list, or \texttt{pos} does not evaluate to a positive integer, or \texttt{list[pos]} is unbound an error is signalled.

\begin{verbatim}
2 ▶ \texttt{list\(\texttt{pos} \)}
\end{verbatim}

The above construct evaluates to a new list \texttt{new} whose first element is \texttt{list[pos[1]]}, whose second element is \texttt{list[pos[2]]}, and so on. \texttt{pos} must be a dense list of positive integers. However, it does not need to be sorted and may contain duplicate elements. If for any \texttt{i}, \texttt{list[pos[i]]} is unbound, an error is signalled.
Section 3. List Elements

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 following example.

\begin{verbatim}
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 ] ]
\end{verbatim}

Note the difference between the two examples. The latter extracts elements 1, 2, and 3 from \textit{m} and then extracts the elements 3 and 2 from this list. The former extracts elements 1, 2, and 3 from \textit{m} and then extracts the elements 3 and 2 from each of those element lists.

To be precise: With each selector \([\text{pos}]\) or \(\{\text{poss}\}\) we associate a level that is defined as the number of selectors of the form \(\{\text{poss}\}\) to its left in the same expression. For example

\begin{verbatim}
level 0 0 1 1 1 2
\end{verbatim}

Then a selector \textit{list}[\text{pos}] of level level is computed as \texttt{ListElement(list,pos,level)}, where \texttt{ListElement} is defined as follows. (Note that \texttt{ListElement} is not a GAP function.)

\begin{verbatim}
ListElement := function ( list, pos, level )
  if level = 0 then
    return list[pos];
  else
    return List( list, elm -> ListElement(elm,pos,level-1) );
  fi;
end;
\end{verbatim}

and a selector \textit{list}\{\text{pos}\} of level level is computed as \texttt{ListElements(list,poss,level)}, where \texttt{ListElements} is defined as follows. (Note that \texttt{ListElements} is not a GAP function.)

\begin{verbatim}
ListElements := function ( list, poss, level )
  if level = 0 then
    return list[poss];
  else
    return List( list, elm -> ListElements(elm,poss,level-1) );
  fi;
end;
\end{verbatim}

This operation implements sublist access. For any list, the default method is to loop over the entries in the list \textit{poss}, and to delegate to the element access operation. (For the somewhat strange variable name, cf. 21.2.)
### 21.4 List Assignment

1. \texttt{list[ pos ] := object;}

The list element assignment assigns the object \texttt{object}, which can be of any type, to the list entry at the position \texttt{pos}, which must be a positive integer, in the mutable (see 12.6) list \texttt{list}. That means that accessing the \texttt{pos}-th element of the list \texttt{list} will return \texttt{object} after this assignment.

```gap
gap> 1 := [ 1, 2, 3 ];;
[ 3, 2, 3 ]
gap> 1[1] := 3;; 1;
# assign a new object
[ 3, 2, 3 ]
gap> 1[2] := [ 4, 5, 6 ];; 1;
# <object> may be of any type
[ 3, [ 4, 5, 6 ], 3 ]
gap> 1[ 1[1] ] := 10;; 1;
# <index> may be an expression
[ 3, [ 4, 5, 6 ], 10 ]
```

If the index \texttt{pos} is larger than the length of the list \texttt{list} (see 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.

```gap
gap> l[4] := "another entry";; 1; # <list> is enlarged
[ 3, [ 4, 5, 6 ], 10, "another entry" ]
gap> l[ 10 ] := 1;; 1;
# now <list> has a hole
[ 3, [ 4, 5, 6 ], 10, "another entry",,, 1 ]
```

The function \texttt{Add} (see 21.4.4) 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 \texttt{list} does not evaluate to a list, \texttt{pos} does not evaluate to a positive integer or \texttt{object} is a call to a function which does not return a value (for example \texttt{Print}) an error is signalled.

2. \texttt{list{ poss } := objects;}

The sublist assignment assigns the object \texttt{objects[1]}, which can be of any type, to the list \texttt{list} at the position \texttt{poss[1]}, the object \texttt{objects[2]} to \texttt{list[poss[2]]}, and so on. \texttt{poss} must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. \texttt{objects} must be a dense list and must have the same length as \texttt{poss}.

```gap
gap> 1 := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> 1{[1..4]} := [10..13];; 1;
[ 10, 11, 12, 13, 11, 13, 17, 19 ]
gap> 1{[1,7,1,10]} := [ 1, 2, 3, 4 ];; 1;
[ 3, 11, 12, 13, 11, 13, 2, 19,, 4 ]
```

It is possible to nest such sublist assignments, as can be seen in the following example.

```gap
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 \texttt{[pos]} or \texttt{[poss]} we associate a level that is defined as the number of selectors of the form \texttt{[poss]} to its left in the same expression. For example
Then a list assignment \( \text{list}[\text{pos}] := \text{vals} \) of level \( \text{level} \) is computed as \( \text{ListAssignment}( \text{list}, \text{pos}, \text{vals}, \text{level} ) \), where \( \text{ListAssignment} \) is defined as follows. (Note that \( \text{ListAssignment} \) is not a GAP function.)

\[
\text{ListAssignment} := \text{function}( \text{list}, \text{pos}, \text{vals}, \text{level} )
\]

\[
\text{local } i;
\]

\[
\text{if } \text{level} = 0 \text{ then}
\]

\[
\text{list}[\text{pos}] := \text{vals};
\]

\[
\text{else}
\]

\[
\text{for } i \text{ in } [1..\text{Length}(\text{list})] \text{ do}
\]

\[
\text{ListAssignment}( \text{list}[i], \text{pos}, \text{vals}[i], \text{level}-1 );
\]

\[
\text{od};
\]

\[
\text{fi};
\]

\[
\text{end};
\]

and a list assignment \( \text{list}\{\text{poss}\} := \text{vals} \) of level \( \text{level} \) is computed as \( \text{ListAssignments}( \text{list}, \text{poss}, \text{vals}, \text{level} ) \), where \( \text{ListAssignments} \) is defined as follows. (Note that \( \text{ListAssignments} \) is not a GAP function.)

\[
\text{ListAssignments} := \text{function}( \text{list}, \text{poss}, \text{vals}, \text{level} )
\]

\[
\text{local } i;
\]

\[
\text{if } \text{level} = 0 \text{ then}
\]

\[
\text{list}\{\text{poss}\} := \text{vals};
\]

\[
\text{else}
\]

\[
\text{for } i \text{ in } [1..\text{Length}(\text{list})] \text{ do}
\]

\[
\text{ListAssignments}( \text{list}[i], \text{poss}, \text{vals}[i], \text{level}-1 );
\]

\[
\text{od};
\]

\[
\text{fi};
\]

\[
\text{end};
\]

This operation implements sublist assignment. For any list, the default method is to loop over the entries in the list \( \text{poss} \), and to delegate to the element assignment operation. (For the somewhat strange variable name, cf. 21.2.)

\[
\text{Add}( \text{list}, \text{obj} )
\]

adds the element \( \text{obj} \) to the end of the mutable list \( \text{list} \), i.e., it is equivalent to the assignment \( \text{list}[\text{Length}(\text{list}) + 1] := \text{obj} \), see 21.4.1. Nothing is returned by \( \text{Add} \), the function is only called for its side effect.

\[
\text{gap> } 1 := [ 2, 3, 5 ];; \text{Add}( 1, 7 ); 1;
\]

\[
[ 2, 3, 5, 7 ]
\]

\[
\text{Append}( \text{list1}, \text{list2} )
\]

adds the elements of the list \( \text{list2} \) to the end of the mutable list \( \text{list1} \), see 21.4.2. \( \text{list2} \) may contain holes, in which case the corresponding entries in \( \text{list1} \) will be left unbound. \( \text{Append} \) returns nothing, it is only called for its side effect.

Note that \( \text{Append} \) changes its first argument, while \( \text{Concatenation} \) (see 21.20.1) creates a new list and leaves its arguments unchanged.

\[
\text{gap> } 1 := [ 2, 3, 5 ];; \text{Append}( 1, [ 7, 11, 13 ] ); 1;
\]

\[
[ 2, 3, 5, 7, 11, 13 ]
\]

\[
\text{gap> } \text{Append}( 1, [ 17,, 23 ] ); 1;
\]

\[
[ 2, 3, 5, 7, 11, 13, 17,, 23 ]
\]
Chapter 21. Lists

21.5 IsBound and Unbind for Lists

\[ \text{IsBound( list } [ n ] \text{ )} \]

\textbf{IsBound} returns \textbf{true} if the list \textit{list} has a element at the position \( n \), and \textbf{false} otherwise. \textit{List} must evaluate to a list, otherwise an error is signalled.

\begin{verbatim}
gap> 1 := [ , 2, 3, , 5, , 7, , , 11 ];;
gap> IsBound( 1[7] );
true
gap> IsBound( 1[4] );
false
gap> IsBound( 1[101] );
false
\end{verbatim}

\[ \text{Unbind( list } [ n ] \text{ )} \]

\textbf{Unbind} deletes the element at the position \( n \) in the mutable list \textit{list}. That is, after execution of \textbf{Unbind}, \textit{list} no longer has an assigned value at the position \( n \). Thus \textbf{Unbind} can be used to produce holes in a list. Note that it is not an error to unbind a nonexisting list element. \textit{List} must evaluate to a list, otherwise an error is signalled.

\begin{verbatim}
gap> 1 := [ , 2, 3, 5, , 7, , , 11 ];;
gap> Unbind( 1[3] ); 1;
[ , 2, , 5, , 7, , , 11 ]
gap> Unbind( 1[4] ); 1;
[ , 2,, , 7,, , 11 ]
\end{verbatim}

Note that \textbf{IsBound} and \textbf{Unbind} are special in that they do not evaluate their argument, otherwise \textbf{IsBound} would always signal an error when it is supposed to return \textbf{false} and there would be no way to tell \textbf{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.

\begin{verbatim}
i := 3;
i := i + 1;
\end{verbatim}

But in this example it is not the \textbf{integer} 3 which is changed, by adding one to it. Instead the \textbf{variable} \textit{i} is changed by assigning the value of \textit{i}+1, which happens to be 4, to \textit{i}. The same thing happens in the following example

\begin{verbatim}
l := [ 1, 2 ];
l := [ 1, 2, 3 ];
\end{verbatim}

The second assignment does not change the first list, instead it assigns a new list to the variable \textit{l}. On the other hand, in the following example the list \textit{l} is changed by the second assignment.

\begin{verbatim}
l := [ 1, 2 ];
l[3] := 3;
\end{verbatim}

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 \textit{var}:=\textit{list}; means in this interpretation that \textit{var}
is a name for the object list. At the end of the following example 12 still has the value [ 1, 2 ] as this list has not been changed and nothing else has been assigned to it.

11 := [ 1, 2 ];
12 := 11;
11 := [ 1, 2, 3 ];

But after the following example the list for which 12 is a name has been changed and thus the value of 12 is now [ 1, 2, 3 ].

11 := [ 1, 2 ];
12 := 11;
11[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, see 12.5.

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 11 and 12 are not identical, though they are equal of course.

11 := [ 1, 2 ];
12 := [ 1, 2 ];

Also in the following example, no lists in the list l are identical.

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 11 and 12 are identical lists.

11 := [ 1, 2 ];
12 := 11;

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 11 and 12 are identical lists:

11 := [ 1, 2 ];
f := function ( l ) return l; end;
12 := f( 11 );

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 11 and 12 are changed, and are still identical.

11 := [ 1, 2 ];
12 := 11;
11[1] := 2;
21.7 Duplication of Lists

Here we describe the meaning of \texttt{ShallowCopy} and \texttt{StructuralCopy} for lists. For the general definition of these functions, see 12.7.

The subobjects (see 12.7.1) of a list are exactly its elements. This means that for any list \texttt{list}, \texttt{ShallowCopy} returns a mutable \new{} \texttt{new} list \texttt{new} that is not identical to any other list (see 21.6), and whose elements are identical to the elements of \texttt{list}.

Analogously, for a \texttt{mutable} list \texttt{list}, \texttt{StructuralCopy} returns a mutable \new{} \texttt{new} list \texttt{scp} that is not identical to any other list, and whose elements are structural copies (defined recursively) of the elements of \texttt{list}; an element of \texttt{scp} is mutable (and then a \new{} \texttt{new}) if and only if the corresponding element of \texttt{list} is mutable. In both cases, modifying the copy \texttt{new} resp. \texttt{scp} by assignments (see 21.4) does not modify the original object \texttt{list}.

\texttt{ShallowCopy} basically executes the following code for lists.

\begin{verbatim}
new := []; for i in [ 1 .. Length( list ) ] do if IsBound( list[i] ) then new[i] := list[i]; fi; od;
\end{verbatim}

\begin{verbatim}
gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];; list2 := ShallowCopy( list1 );; gap> IsIdenticalObj( list1, list2 );
false gap> IsIdenticalObj( list1[1], list2[1] );
true gap> list2[1][1] := 0;; list1; list2;
[ [ 1, 2 ], [ 3, 4 ] ]
[ 0, [ 3, 4 ] ]
\end{verbatim}

\texttt{StructuralCopy} basically executes the following code for lists.

\begin{verbatim}
new := []; for i in [ 1 .. Length( list ) ] do if IsBound( list[i] ) then new[i] := StructuralCopy( list[i] ); fi; od;
\end{verbatim}

\begin{verbatim}
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 ] ]
\end{verbatim}

The above code is not entirely correct. If the object \texttt{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 \texttt{new} and the object \texttt{list} have exactly the same structure when viewed as a general graph.
Section 9. Enlarging Internally Represented Lists

21.8 Membership Test for Lists

1&gt; obj in list

tests whether there is a positive integer index such that list[index] = obj.

If the list list knows that it is strictly sorted (see 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.

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;
false # uses binary search and only 4 comparisons

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 (see 21.17.5), is the logical length.

Suppose that Length 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, and other functions that need to know the logical length, more efficient, the length of a list is stored along with the list.
21.10 Comparisons of Lists

\[ \text{list1} = \text{list2} \]
\[ \text{list1} \not<\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not<\not\not\not\not\not\not\not\not\not\not\not\not\not<\not\not\not\not\not\not\not\not\not\not\not\not\not<\not\not\not\not\not\not\not\not\not\not\not\not\not< \text{list2} \]

Two lists \text{list1} and \text{list2} are equal if and only if for every index \( i \), either both entries \( \text{list1}[i] \) and \( \text{list2}[i] \) are unbound, or both are bound and are equal, i.e., \( \text{list1}[i] = \text{list2}[i] \) is true.

\[
\text{gap> } [1, 2, 3] = [1, 2, 3]; \\
\text{true} \\
\text{gap> } [,,2,3] = [1,2,]; \\
\text{false} \\
\text{gap> } [1,2,3] = [3,2,1]; \\
\text{false}
\]

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.

\[ \text{list1} < \text{list2} \]
\[ \text{list1} \leq\leq \text{list2} \]

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 \( \text{list1} \) and \( \text{list2} \) at position \( i \) differ, i.e., either exactly one of \( \text{list1}[i] \), \( \text{list2}[i] \) is bound or both entries are bound and differ. Then \text{list1} is less than \text{list2} if either \( \text{list1}[i] \) is unbound (and \( \text{list2}[i] \) is not) or both are bound and \( \text{list1}[i] < \text{list2}[i] \) is true.

\[
\text{gap> } [1,2,3,4] < [1,2,4,8]; \\
\text{true} \quad \# <\text{list1}>[3] < <\text{list2}>[3] \\
\text{gap> } [1,2,3] < [1,2,3,4]; \\
\text{true} \quad \# <\text{list1}>[4] \text{ is unbound and therefore very small} \\
\text{gap> } [1,,3,4] < [1,2,3]; \\
\text{true} \quad \# <\text{list1}>[2] \text{ is unbound and therefore very small}
\]

Note that for comparing two lists with \(<\) or \(\leq\), 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, see 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 24.2) and row vectors (see 23.1) 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 \text{Length} (see 21.17.5) returning the number of rows etc.

For the categories and attributes that control the arithmetic behaviour of lists, see 21.12.
Section 12. Filters Controlling the Arithmetic Behaviour of Lists

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 23.1 and in 24.2, the compression status is dealt with in 23.2 and 24.13.

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.

1 ▶ IsGeneralizedRowVector( list )

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 (see 21.12.4) will return a nonzero value when called with list.

2 ▶ IsMultiplicativeGeneralizedRowVector( list )

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 (see 21.12.5) will return a nonzero value when called with list.

Note that these filters do not enable default methods for addition or multiplication (cf. 21.12.3).

```gap
gap> IsList( "abc" ); IsGeneralizedRowVector( "abc" );
true
false
gap> liemat:= LieObject( [[ 1, 2 ], [ 3, 4 ] ] );
LieObject( [[ 1, 2 ], [ 3, 4 ] ] )
gap> IsGeneralizedRowVector( liemat );
true
gap> IsMultiplicativeGeneralizedRowVector( liemat );
false
gap> bas:= CanonicalBasis( FullRowSpace( Rationals, 3 ) );
CanonicalBasis( ( Rationals^3 ) )
gap> IsMultiplicativeGeneralizedRowVector( bas );
true
```

3 ▶ IsListDefault( list )

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.


All internally represented lists are in this category, and also all lists in the representations IsGF2VectorRep, Is8BitVectorRep, IsGF2MatrixRep, and Is8BitMatrixRep (see 23.2 and 24.13). 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.
Chapter 21. Lists

NestingDepthA( obj )

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, as the integer 1 if obj is an empty list in IsGeneralizedRowVector, and as 1 plus the additive nesting depth of the first bound entry in obj otherwise.

NestingDepthM( obj )

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, as the integer 1 if obj is an empty list in IsMultiplicativeGeneralizedRowVector, and as 1 plus the multiplicative nesting depth of the first bound entry in obj otherwise.

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; for binary operations (addition and subtraction), at least one argument must be in IsGeneralizedRowVector, and the other either is not a list or also in IsGeneralizedRowVector.

(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, it shall be left to this argument whether the result in question is defined and what it is.)

Zero

The zero (see 30.10.3) of a list x in IsGeneralizedRowVector 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.
Additive Arithmetic for Lists

Section 13. Additive Arithmetic for Lists

181

Section 13. Additive Arithmetic for Lists

AdditiveInverse

The additive inverse (see 30.10.9) of a list \( x \) in \texttt{IsGeneralizedRowVector} 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.

\[
gap> \text{AdditiveInverse}([1, 2, 3]); \quad \text{AdditiveInverse}([[1, 2], 3]); \\
\text{[ -1, -2, -3 ]}
\]

Addition

If \( x \) and \( y \) are in \texttt{IsGeneralizedRowVector} and have the same additive nesting depth (see 21.12.4), the sum \( x + y \) is defined \texttt{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 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 \texttt{IsGeneralizedRowVector} and \( y \) is in \texttt{IsGeneralizedRowVector} 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.

\[
gap> 1 + [1, 2, 3]; \quad [1, 2, 3] + [0, 2, 4]; \quad [1, 2] + [Z(2)]; \\
\text{[ 2, 3, 4 ]} \quad \text{[ 1, 4, 7 ]} \quad \text{[ 0*Z(2), 2 ]}
\]

Subtraction

For two \texttt{GAP} objects \( x \) and \( y \) of which one is in \texttt{IsGeneralizedRowVector} and the other is also in \texttt{IsGeneralizedRowVector} or is neither a list nor a domain, \( x - y \) is defined as \( x + (-y) \).
Chapter 21. Lists

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 \textit{IsMultiplicativeGeneralizedRowVector}; for binary operations (multiplication and division), at least one argument must be in \textit{IsMultiplicativeGeneralizedRowVector}, and the other either not a list or also in \textit{IsMultiplicativeGeneralizedRowVector}.

(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 \textit{IsMultiplicativeGeneralizedRowVector}, it shall be left to this argument whether the result in question is defined and what it is.)

One

The one (see 30.10.2) of a dense list \(x\) in \textit{IsMultiplicativeGeneralizedRowVector} 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 \(\text{One}(x[1][1])\) and off-diagonal entries \(\text{Zero}(x[1][1])\).

\begin{verbatim}
gap> One( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ 1, 0 ], [ 0, 1 ] ]
\end{verbatim}

Inverse

The inverse (see 30.10.8) of an invertible square table \(x\) in \textit{IsMultiplicativeGeneralizedRowVector} 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 \(\text{One}(x)\).

\begin{verbatim}
gap> Inverse( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ -2, 1 ], [ 3/2, -1/2 ] ]
\end{verbatim}

Multiplication

There are three possible computations that might be triggered by a multiplication involving a list in \textit{IsMultiplicativeGeneralizedRowVector}. Namely, \(x \ast y\) might be

(I) the inner product \(x[1] \ast y[1] + x[2] \ast y[2] + \cdots + x[n] \ast y[n]\), 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 \(x \ast y[i]\) 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 \(x[i] \ast y\) 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.

\[
\begin{array}{ccc}
\text{scl} & \text{vec} & \text{mat} \\
\hline
\text{scl} & (\text{L}) & (\text{L}) \\
\text{vec} & (\text{R}) & (\text{I}) \\
\text{mat} & (\text{R}) & (\text{R}) \\
\end{array}
\]

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` 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`, 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.

\[
gap> [ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ] * (1,4); \\
gap> [ 1, 2, , 4 ] * 2; \\
gap> [ 1, 2, 3 ] * [ 1, 3, 5, 7 ]; \\
gap> m:= [ [ 1, 2 ], 3 ];; m * m; \\
gap> m * n = n[1] * m, m[2] * m; \\
gap> true \\
gap> true
\]

**Division**

For two GAP objects \(x\) and \(y\) of which one is in `IsMultiplicativeGeneralizedRowVector` and the other is also in `IsMultiplicativeGeneralizedRowVector` or is neither a list nor a domain, \(x/y\) is defined as \(x * y^{-1}\).
Chapter 21. Lists

gap> [ 1, 2, 3 ] / 2; [ 1, 2 ] / [ [ 1, 2 ], [ 3, 4 ] ];
[ 1/2, 1/2, 3/2 ]
[ 1, 0 ]

mod

If $x$ and $y$ are in IsMultiplicativeGeneralizedRowVector and have the same multiplicative nesting depth (see 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 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 and $y$ is in IsMultiplicativeGeneralizedRowVector 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 $x[i] \mod y[i]$ 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.

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 ]

Left Quotient

For two GAP objects $x$ and $y$ of which one is in IsMultiplicativeGeneralizedRowVector and the other is also in IsMultiplicativeGeneralizedRowVector or is neither a list nor a domain, $\text{LeftQuotient}(x, y)$ is defined as $x^{-1} \ast y$.

gap> \text{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, the sequence $x = x_1, x_2, \ldots x_n$ where $x_{n+1}$ 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. 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 (\texttt{Zero}, \texttt{AdditiveInverse}, \texttt{One}, \texttt{Inverse}), an operation that returns a result that is mutable (\texttt{ZeroOp}, \texttt{AdditiveInverseOp}, \texttt{OneOp}, \texttt{InverseOp}), and an operation whose result has the same immutability level as the argument (\texttt{ZeroSM}, \texttt{AdditiveInverseSM}, \texttt{OneSM}, \texttt{InverseSM}). The last kind of operations is equivalent to the corresponding infix operations \texttt{0 * list}, \texttt{- list}, \texttt{list^0}, and \texttt{list^{-1}}. (This holds not only for lists, see 12.6.)

\begin{verbatim}
  gap> IsMutable( l1 ); IsMutable( 2 * Immutable( [ 1, 2, 3 ] ) );
  true false
  gap> IsMutable( l2 ); IsMutable( l3 );
  true true
\end{verbatim}

An example motivating the mutability rule is the use of syntactic constructs such as \texttt{obj * list} and \texttt{- 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 \texttt{0 * [ 1 .. n ]}. The latter can be done also using \texttt{ListWithIdenticalEntries}.

\begin{verbatim}
1 ▶ ListWithIdenticalEntries( n, obj )
  is a list \texttt{list} of length \( n \) that has the object \texttt{obj} stored at each of the positions from 1 to \( n \). Note that all elements of \texttt{lists} are identical, see 21.6.
  gap> ListWithIdenticalEntries( 10, 0 );
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
\end{verbatim}

\section{Finding Positions in Lists}

\begin{verbatim}
1 ▶ Position( list, obj [, from] )
  returns the position of the first occurrence \texttt{obj} in \texttt{list}, or \texttt{fail} if \texttt{obj} is not contained in \texttt{list}. If a starting index \texttt{from} is given, it returns the position of the first occurrence starting the search \textit{after} position \texttt{from}.
  Each call to the two argument version is translated into a call of the three argument version, with third argument the integer zero \texttt{0}. (Methods for the two argument version must be installed as methods for the version with three arguments, the third being described by \texttt{IsZeroCyc}.)
  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
\end{verbatim}

\begin{verbatim}
2 ▶ PositionCanonical( list, obj )
  returns the position of the canonical associate of \texttt{obj} in \texttt{list}. The definition of this associate depends on \texttt{list}. For internally represented lists it is defined as the element itself (and \texttt{PositionCanonical} thus defaults to
\end{verbatim}
Position, see 21.16.1), but for example for certain enumerators (see 21.23) other canonical associates can be defined.

For example RightTransversal defines the canonical associate to be the element in the transversal defining the same coset of a subgroup in a group.

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
gap> PositionCanonical(rt, (1,2));
2
\end{verbatim}

\begin{verbatim}
space
3\textbf{\texttt{PositionNthOccurrence}}( list, obj, n ) O
\end{verbatim}
returns the position of the \texttt{n}-th occurrence of \texttt{obj} in \texttt{list} and returns \texttt{fail} if \texttt{obj} does not occur \texttt{n} times.

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
space
4\textbf{\texttt{PositionSorted}}( list, elm ) O
\end{verbatim}
\begin{verbatim}
4\textbf{\texttt{PositionSorted}}( list, elm, func ) O
\end{verbatim}

In the first form \texttt{PositionSorted} returns the position of the element \texttt{elm} in the sorted list \texttt{list}.

In the second form \texttt{PositionSorted} returns the position of the element \texttt{elm} in the list \texttt{list}, which must be sorted with respect to \texttt{func}. \texttt{func} must be a function of two arguments that returns \texttt{true} if the first argument is less than the second argument and \texttt{false} otherwise.

\texttt{PositionSorted} returns \texttt{pos} such that \texttt{list[pos - 1] \textless elm and elm \leq list[pos]}. That means, if \texttt{elm} appears once in \texttt{list}, its position is returned. If \texttt{elm} appears several times in \texttt{list}, the position of the first occurrence is returned. If \texttt{elm} is not an element of \texttt{list}, the index where \texttt{elm} must be inserted to keep the list sorted is returned.

\texttt{PositionSorted} uses binary search, whereas \texttt{Position} 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 21.17.3 and 21.17.4.

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
gap> PositionSorted( [1,4,5,5,6,7], 5 );
3
\end{verbatim}

\begin{verbatim}
gap> PositionSorted( [1,4,5,5,6,7], 8 );
7
\end{verbatim}
Section 16. Finding Positions in Lists

5 ▶ PositionSet( list, obj )
▶ PositionSet( list, obj, func )

PositionSet is a slight variation of PositionSorted. The only difference to PositionSorted is that PositionSet returns fail if obj is not in list.

```
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
```

6 ▶ PositionProperty( list, func )

returns the first position of an element in the list list for which the property tester function func returns true.

```
gap> PositionProperty( [10^7..10^8], IsPrime );
20

gap> PositionProperty( [10^5..10^6],
>    n -> not IsPrime(n) and IsPrimePowerInt(n) );
490
```

First (see 21.20.19) allows you to extract the first element of a list that satisfies a certain property.

7 ▶ PositionBound( list )

returns the first index for which an element is bound in the list list. For the empty list it returns fail.

```
gap> PositionBound([1,2,3]);
1

gap> PositionBound([,1,2,3]);
2
```

8 ▶ PositionNot( list, val[, from-minus-one] )

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 nonnegative integer is given as optional argument from-minus-one then the first position larger than from-minus-one with this property is returned.

```
PositionNonZero( vec )
```

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.

PositionNonZero implements a special case of PositionNot (see 21.16.8). 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.
Chapter 21. Lists

\begin{verbatim}
gap> l := [ 1, 1, 2, 3, 2 ];
3
gap> PositionNot( l, 1 );
5
6
gap> PositionNonZero( l );
1
2
\end{verbatim}

\texttt{PositionSublist( list, sub )}
\texttt{PositionSublist( list, sub, from )}

returns the smallest index in the list \texttt{list} at which a sublist equal to \texttt{sub} starts. If \texttt{sub} does not occur the operation returns \texttt{fail}. The second version starts searching \texttt{after} position \texttt{from}.

To determine whether \texttt{sub} matches \texttt{list} at a particular position, use \texttt{IsMatchingSublist} instead (see 21.17.1).

\section*{21.17 Properties and Attributes for Lists}

\begin{verbatim}
\texttt{IsMatchingSublist( list, sub )}
\texttt{IsMatchingSublist( list, sub, at )}
\end{verbatim}

returns \texttt{true} if \texttt{sub} matches a sublist of \texttt{list} from position 1 (or position \texttt{at}, in the case of the second version), or \texttt{false}, otherwise. If \texttt{sub} is empty \texttt{true} is returned. If \texttt{list} is empty but \texttt{sub} is non-empty \texttt{false} is returned.

If you actually want to know whether there is an \texttt{at} for which \texttt{IsMatchingSublist( list, sub, at )} is true, use a construction like \texttt{PositionSublist( list, sub ) fail} instead (see 21.16.10); it’s more efficient.

\textbf{Note:} A list that contains mutable objects (like lists or records) \texttt{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.

\begin{verbatim}
gap> l := [[1],[2]];
\end{verbatim}

\begin{verbatim}
[ [ 1 ], [ 2 ] ]
gap> IsSSortedList( l );
true
gap> l[1][1]:=3;
3
gap> IsSSortedList( l );
false
\end{verbatim}

For such lists these property values must be computed anew each time the property is asked for. For example, if \texttt{list} is a list of mutable row vectors then the call of \texttt{Position} (see 21.16.1) with \texttt{list} as first argument cannot take advantage of the fact that \texttt{list} is in fact sorted. One solution is to call explicitly \texttt{PositionSorted} (see 21.16.4) in such a situation, another solution is to replace \texttt{list} by an immutable copy using \texttt{Immutable} (see 12.6).

\begin{verbatim}
\texttt{IsDuplicateFree( obj )}
\texttt{IsDuplicateFreeList( obj )}
\end{verbatim}

\texttt{IsDuplicateFree(obj)}; returns \texttt{true} if \texttt{obj} is both a list or collection, and it is duplicate free; otherwise it returns \texttt{false}. \texttt{IsDuplicateFreeList} is a synonym for \texttt{IsDuplicateFree} and \texttt{IsEmpty}.

A list is \texttt{duplicate free} if it is dense and does not contain equal entries in different positions. Every domain (see 12.4) is duplicate free.
Section 18. Sorting Lists

\section*{18. Sorting Lists}

\subsection*{18.1 IsSortedList( obj )}
\begin{itemize}
\item Returns \texttt{true} if \texttt{obj} is a list and it is sorted, or \texttt{false} otherwise.
\end{itemize}

A list \texttt{list} is \texttt{sorted} if it is dense (see 21.1.2) and satisfies the relation \(\text{list}[i] \leq \text{list}[j]\) whenever \(i < j\). Note that a sorted list is not necessarily duplicate free (see 21.17.2 and 21.17.4).

Many sorted lists are in fact homogeneous (see 21.1.3), but also non-homogeneous lists may be sorted (see 30.11).

\subsection*{18.2 IsSSortedList( obj )}
\begin{itemize}
\item Returns \texttt{true} if \texttt{obj} is a list and it is strictly sorted, or \texttt{false} otherwise. \texttt{IsSSortedList} is short for “is strictly sorted list”; \texttt{IsSet} is just a synonym for \texttt{IsSSortedList}.
\end{itemize}

A list \texttt{list} is \texttt{strictly sorted} if it is sorted (see 21.17.3) and satisfies the relation \(\text{list}[i] \triangleleft \text{list}[j]\) whenever \(i < j\). In particular, such lists are duplicate free (see 21.17.2).

In sorted lists, membership test and computing of positions can be done by binary search, see 21.19.

(Currently there is no 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.)

\subsection*{18.3 Length( list )}
\begin{itemize}
\item Returns the length of the list \texttt{list}, which is defined to be the index of the last bound entry in \texttt{list}.
\end{itemize}

\subsection*{18.4 ConstantTimeAccessList( list )}
\begin{itemize}
\item Returns an immutable list containing the same elements as the list \texttt{list} (which may have holes) in the same order. If \texttt{list} is already a constant time access list, \texttt{ConstantTimeAccessList} returns an immutable copy of \texttt{list} directly. Otherwise it puts all elements and holes of \texttt{list} into a new list and makes that list immutable.
\end{itemize}

\section*{21.18 Sorting Lists}

\subsection*{21.18.1 Sort( list )}
\begin{itemize}
\item Sorts the list \texttt{list} in increasing order. In the first form \texttt{Sort} uses the operator \(<\) to compare the elements. (If the list is not homogeneous it is the user's responsibility to ensure that \(<\) is defined for all element pairs, see 30.11) In the second form \texttt{Sort} uses the function \texttt{func} to compare elements. \texttt{func} must be a function taking two arguments that returns \texttt{true} if the first is regarded as strictly smaller than the second, and \texttt{false} otherwise.
\end{itemize}

\texttt{Sort} does not return anything, it just changes the argument \texttt{list}. Use \texttt{ShallowCopy} (see 12.7.1) if you want to keep \texttt{list}. Use \texttt{Reversed} (see 21.20.7) if you want to get a new list 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., \texttt{Sort} is not stable.
# note the random order of the elements with equal first component

2► SortParallel( list, list2 )
   O

   ▶ SortParallel( list, list2, func )
   O

   sorts the list list1 in increasing order just as Sort (see 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.

   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, 6, 7, 7 ]
gap> list2;
[ 7, 3, 2, 9, 5, 8 ] # [ 7, 3, 9, 2, 5, 8 ] is also possible

3► Sortex( list )
   O

   sorts the list list via the operator< and returns the permutation that must be applied to list to obtain the sorted list. (If the list is not homogeneous it is the users responsibility to ensure that < is defined for all element pairs, see 30.11)

   Permuted (see 21.20.15) allows you to rearrange a list according to a given permutation.

   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, 6, 7, 7 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]

4► SortingPerm( list )
   F

   SortingPerm returns the same as Sortex( list ) but does not change the argument.

   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 ]

Currently GAP uses shellsort.
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, see 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 or RemoveSet (see 21.19.6, 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 17), 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 and Union, are described in Chapter 28. In particular, for the construction of proper sets, see 28.2.5 and 28.2.8. For finding positions in sorted lists, see 21.16.4.

1► obj in list

The element test for strictly sorted lists uses binary search.

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 (see 28.2.5) is silently applied to it first (see 28.2.5).

2► IsEqualSet( list1, list2 )

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 (see 28.2.5) 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 28.2.5), but the former is more efficient.

    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

3► IsSubsetSet( list1, list2 )

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 (see 28.2.5) is applied to it first.

4► AddSet( set, obj )

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 \( \text{obj} \) must be in the same family as each element of \( \text{set} \).

```gap
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 ]
```

5 ▶ RemoveSet( \( \text{set}, \text{obj} \) )

removes the element \( \text{obj} \) from the proper set \( \text{set} \). If \( \text{obj} \) is not contained in \( \text{set} \) then \( \text{set} \) is not changed. If \( \text{obj} \) is an element of \( \text{set} \) it is removed and all the following elements in the list are moved one position forward.

```gap
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 ]
```

6 ▶ UniteSet( \( \text{set}, \text{list} \) )

unites the proper set \( \text{set} \) with \( \text{list} \). This is equivalent to adding all elements of \( \text{list} \) to \( \text{set} \) (see 21.19.4).

```gap
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 ]
```

7 ▶ IntersectSet( \( \text{set}, \text{list} \) )

intersects the proper set \( \text{set} \) with \( \text{list} \). This is equivalent to removing from \( \text{set} \) all elements of \( \text{set} \) that are not contained in \( \text{list} \).

```gap
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 ]
```

8 ▶ SubtractSet( \( \text{set}, \text{list} \) )

subtracts \( \text{list} \) from the proper set \( \text{set} \). This is equivalent to removing from \( \text{set} \) all elements of \( \text{list} \).

```gap
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 ]
```

There are nondestructive counterparts of the functions UniteSet, IntersectSet, and SubtractSet available for proper sets. These are UnionSet, IntersectionSet, and Difference. The former two are methods for the more general operations Union and Intersection (see 28.4.3, 28.4.2), the latter is itself an operation (see 28.4.4).

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 \( \text{set} \). If \( \text{set} \) is not a proper set it is not specified to which of a number of equal elements in \( \text{set} \) the element in the result is identical (see 21.6).
21.20 Operations for Lists

Several of the following functions expect the first argument to be either a list or a collection (see 28), with possibly slightly different meaning for lists and non-list collections. For these functions, the list case is indicated by an argument named \texttt{list}, and the collection case by one named \texttt{C}.

\begin{verbatim}
1 \textbf{Concatenation( list1, list2, \ldots )} \textbf{F}
\textbf{Concatenation( list )} \textbf{F}
\end{verbatim}

In the first form \texttt{Concatenation} returns the concatenation of the lists \texttt{list1}, \texttt{list2}, etc. The \texttt{concatenation} is the list that begins with the elements of \texttt{list1}, followed by the elements of \texttt{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 \texttt{list} must be a dense list of lists \texttt{list1}, \texttt{list2}, etc., and \texttt{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 \texttt{list1}, \texttt{list2}, etc. (see 21.6).

Note that \texttt{Concatenation} creates a new list and leaves its arguments unchanged, while \texttt{Append} (see 21.4.5) changes its first argument. For computing the union of proper sets, \texttt{Union} can be used, see 28.4.3 and 21.19.

\begin{verbatim}
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 ]
\end{verbatim}

\begin{verbatim}
2 \textbf{Compacted( list )} \textbf{O}
\end{verbatim}

returns a new mutable list that contains the elements of \texttt{list} in the same order but omitting the holes.

\begin{verbatim}
gap> 1:=[1,,,3,,,4,[5,,,6],7];; Compacted( 1 );
[ 1, 3, 4, [ 5,,, 6 ], 7 ]
\end{verbatim}

\begin{verbatim}
3 \textbf{Collected( list )} \textbf{O}
\end{verbatim}

returns a new list \texttt{new} that contains for each element \texttt{elm} of the list \texttt{list} a list of length two, the first element of this is \texttt{elm} itself and the second element is the number of times \texttt{elm} appears in \texttt{list}. The order of those pairs in \texttt{new} corresponds to the ordering of the elements \texttt{elm}, so that the result is sorted.

For all pairs of elements in \texttt{list} the comparison via \texttt{<} must be defined.

\begin{verbatim}
gap> Factors( Factorial( 10 ) );
[ 2, 2, 2, 2, 2, 2, 2, 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 ] ]
\end{verbatim}

\begin{verbatim}
4 \textbf{DuplicateFreeList( list )} \textbf{O}
\textbf{Unique( list )} \textbf{O}
\end{verbatim}

returns a new mutable list whose entries are the elements of the list \texttt{list} with duplicates removed. \texttt{DuplicateFreeList} only uses the = comparison and will not sort the result. Therefore \texttt{DuplicateFreeList} can be used even if the elements of \texttt{list} do not lie in the same family. \texttt{Unique} is an alias for \texttt{DuplicateFreeList}.
AsDuplicateFreeList( list )
returns the same result as DuplicateFreeList (see 21.20.4), except that the result is immutable.

Flat( list )
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.

(To reconstruct a matrix from a Flattened list, the sublist operator can be used:

`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 ] ]`
)

Reversed( list )
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).

`gap> reversed( [ 1, 4, 9, 5, 6, 7 ] );`  
`[ 7, 6, 5, 9, 4, 1 ]`

IsLexicographicallyLess( list1, list2 )
Let list1 and list2 be two dense lists, but not necessarily homogeneous (see 21.1.2, 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.

Apply( list, func )
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.

`gap> l:= [ 1, 2, 3 ];`  
`Apply( l, i -> i^2 );`  
`[ 1, 4, 9 ]`

PermListList( list1, list2 )
returns a permutation p of [ 1 .. Length( list1 ) ] such that list1[i^p] = list2[i]. It returns fail if there is no such permutation.
Section 20. Operations for Lists

```gap
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 ]
```

**Maximum**

In the first form `Maximum` returns the maximum of its arguments, i.e., one argument `obj` for which `obj ≥ obj1`, `obj ≥ obj2` etc. In the second form `Maximum` takes a homogeneous list `list` and returns the maximum of the elements in this list.

```gap
gap> Maximum( -123, 700, 123, 0, -1000 );
    700
gap> Maximum( [ -123, 700, 123, 0, -1000 ] );
    700
gap> Maximum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
    [ 2, -11 ]  # lists are compared elementwise
```

**Minimum**

In the first form `Minimum` returns the minimum of its arguments, i.e., one argument `obj` for which `obj ≤ obj1`, `obj ≤ obj2` etc. In the second form `Minimum` takes a homogeneous list `list` and returns the minimum of the elements in this list.

```gap
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 ]
```

**MaximumList**

```gap
return the maximum resp. the minimum of the elements in the list `list`. They are the operations called by `Maximum` resp. `Minimum`. 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).
```

**Cartesian**

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 \( \text{tup1} \) of \( \text{cart} \) contains the first element from \( \text{list1} \), from \( \text{list2} \) and so on. The second element \( \text{tup2} \) of \( \text{cart} \) contains the first element from \( \text{list1} \), the first from \( \text{list2} \), an so on, but the last element of \( \text{tup2} \) is the second element of the last argument list. This implies that \( \text{cart} \) is a proper set if and only if all argument lists are proper sets (see 21.19).

The function \( \text{Tuples} \) (see 17.2.7) computes the \( k \)-fold cartesian product of a list.

\[
\begin{align*}
gap> & \text{Cartesian( [1,2], [3,4], [5,6] );} \\
& \quad \begin{pmatrix} 1, 3, 5 \end{pmatrix}, \begin{pmatrix} 1, 4, 5 \end{pmatrix}, \begin{pmatrix} 1, 4, 6 \end{pmatrix}, \begin{pmatrix} 2, 3, 5 \end{pmatrix}, \\
& \quad \begin{pmatrix} 2, 3, 6 \end{pmatrix}, \begin{pmatrix} 2, 4, 5 \end{pmatrix}, \begin{pmatrix} 2, 4, 6 \end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
gap> & \text{Cartesian( [1,2,2], [1,1,2] );} \\
& \quad \begin{pmatrix} 1, 1 \end{pmatrix}, \begin{pmatrix} 1, 1 \end{pmatrix}, \begin{pmatrix} 1, 2 \end{pmatrix}, \begin{pmatrix} 2, 1 \end{pmatrix}, \begin{pmatrix} 2, 1 \end{pmatrix}, \begin{pmatrix} 2, 2 \end{pmatrix}
\end{align*}
\]

\begin{align*}
\text{Permutated( list, perm )} & \\
\text{List( list )} & \\
\text{List( C )} & \\
\text{List( list, func )}
\end{align*}

returns a new list \( \text{new} \) that contains the elements of the list \( \text{list} \) permuted according to the permutation \( \text{perm} \). That is \( \text{new}[i \mod \text{perm}] = \text{list}[i] \).

\( \text{Sortex} \) (see 21.18.3) allows you to compute a permutation that must be applied to a list in order to get the sorted list.

\[
\begin{align*}
gap> & \text{Permutated( [ 5, 4, 6, 1, 7, 5 ], (1,3,5,6,4) );} \\
& \quad [ 1, 4, 5, 5, 6, 7 ]
\end{align*}
\]

\begin{align*}
\text{List( list )} & \\
\text{List( C )} & \\
\text{List( list, func )}
\end{align*}

In the first form, where \( \text{list} \) is a list (not necessarily dense or homogeneous), \( \text{List} \) returns a new mutable list \( \text{new} \) that contains the elements (and the holes) of \( \text{list} \) in the same order; thus \( \text{List} \) does the same as \( \text{ShallowCopy} \) (see 12.7.1) in this case.

In the second form, where \( \text{C} \) is a collection (see 28) that is not a list, \( \text{List} \) returns a new mutable list \( \text{new} \) such that \( \text{Length( new )} \) is the number of different elements of \( \text{C} \), and \( \text{new} \) contains the different elements of \( \text{C} \) in an unspecified order which may change for repeated calls. \( \text{new}[\text{pos}] \) executes in constant time (see 21.1.5), and the size of \( \text{new} \) is proportional to its length. The generic method for this case is \( \text{ShallowCopy( Enumerator( C ) ).} \)

In the third form, for a dense list \( \text{list} \) and a function \( \text{func} \), which must take exactly one argument, \( \text{List} \) returns a new mutable list \( \text{new} \) given by \( \text{new}[\text{pos}] = \text{func( list[ pos ] )} \).

\[
\begin{align*}
gap> & \text{List( [1,2,3], i -> i^2 );} \\
& \quad [ 1, 4, 9 ]
\end{align*}
\]

\begin{align*}
\text{Filtered( list, func )} & \\
\text{Filtered( C, func )}
\end{align*}

returns a new list that contains those elements of the list \( \text{list} \) or collection \( \text{C} \) (see 28), respectively, for which the unary function \( \text{func} \) returns \( \text{true} \).

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 \( \text{list} \). If an element for which \( \text{func} \) returns \( \text{true} \) appears several times in \( \text{list} \) it will also appear the same number of times in the result. \( \text{list} \) may contain holes, they are ignored by \( \text{Filtered} \).

For each element of \( \text{list} \) resp. \( \text{C} \), \( \text{func} \) must return either \( \text{true} \) or \( \text{false} \), 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 (see 21.4) can be used to extract elements of a list according to indices given in another list.

```gap
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) ]
```

In the first form, \texttt{Number} returns the number of bound entries in the list \texttt{list}. For dense lists \texttt{Number}, \texttt{Length} (see 21.17.5), and \texttt{Size} (see 28.3.6) return the same value; for lists with holes \texttt{Number} returns the number of bound entries, \texttt{Length} returns the largest index of a bound entry, and \texttt{Size} signals an error.

In the last two forms, \texttt{Number} returns the number of elements of the list \texttt{list} resp. the collection \texttt{C} for which the unary function \texttt{func} returns \texttt{true}. If an element for which \texttt{func} returns \texttt{true} appears several times in \texttt{list} it will also be counted the same number of times.

For each element of \texttt{list} resp. \texttt{C}, \texttt{func} must return either \texttt{true} or \texttt{false}, otherwise an error is signalled.

\texttt{Filtered} (see 21.20.17) allows you to extract the elements of a list that have a certain property.

```gap
gap> Number( [ 2, 3, 5, 7 ] );
4
gap> Number( [ , 2, , 5,, 7,,, 11 ] );
5
gap> Number( [1..20], IsPrime );
8
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
4
> n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
2
gap> Number( Group( (1,2), (1,2,3) ), x -> Order( x ) = 2 );
3
```

\texttt{First} returns the first element of the list \texttt{list} for which the unary function \texttt{func} returns \texttt{true}. \texttt{list} may contain holes. \texttt{func} must return either \texttt{true} or \texttt{false} for each element of \texttt{list}, otherwise an error is signalled. If \texttt{func} returns \texttt{false} for all elements of \texttt{list} then \texttt{First} returns \texttt{fail}.

\texttt{PositionProperty} (see 21.16.6) allows you to find the position of the first element in a list that satisfies a certain property.
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

20\textgreater{}\texttt{ForAll( list, func )} \hspace{1cm} F
\textgreater{}\texttt{ForAll( C, func )} \hspace{1cm} F

tests whether the unary function \textit{func} returns \texttt{true} for all elements in the list \textit{list} resp. the collection \textit{C}.

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\textgreater{}\texttt{ForAny( list, func )} \hspace{1cm} F
\textgreater{}\texttt{ForAny( C, func )} \hspace{1cm} F

tests whether the unary function \textit{func} returns \texttt{true} for at least one element in the list \textit{list} resp. the collection \textit{C}.

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..4], j -> IsPrime(i+j) ) );
true

22\textgreater{}\texttt{Product( list[, init] )} \hspace{1cm} F
\textgreater{}\texttt{Product( C[, init] )} \hspace{1cm} F
\textgreater{}\texttt{Product( list, func[, init] )} \hspace{1cm} F
\textgreater{}\texttt{Product( C, func[, init] )} \hspace{1cm} F

In the first two forms \texttt{Product} returns the product of the elements of the dense list \textit{list} resp. the collection \textit{C} (see 28). In the last two forms \texttt{Product} applies the function \textit{func}, which must be a function taking one argument, to the elements of the dense list \textit{list} resp. the collection \textit{C}, and returns the product of the results. In either case \texttt{Product} returns \texttt{1} if the first argument is empty.

If an additional initial value \textit{init} is given, \texttt{Product} returns the product of \textit{init} and the elements of the first argument resp. of their images under the function \textit{func}. This is useful for example if the first argument is empty and a different identity than \texttt{1} is desired, in which case \texttt{init} is returned.
Section 20. Operations for Lists

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)

23 ▶ Sum( list[, init] )  
▶ Sum( C[, init] )  
▶ Sum( list, func[, init] )  
▶ Sum( C, func[, init] )  

In the first two forms Sum returns the sum of the elements of the dense list list resp. the collection C (see 28). In the last two forms Sum applies the function func, which must be a function taking one argument, to the elements of the dense list list resp. the collection C, and returns the sum of the results. In either case Sum returns 0 if the first argument is empty.

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.

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)

24 ▶ Iterated( list, func )  

returns the result of the iterated application of the function func, which must take two arguments, to the elements of the list list. More precisely Iterated returns the result of the following application, \( f(\cdots f(f(list[1], list[2]), list[3]), \ldots, list[n]) \).

gap> Iterated( [ 126, 66, 105 ], Gcd );
3

25 ▶ ListN( list1, list2, ..., listn, f )  

Applies the n-argument function func to the lists. That is, ListN returns the list whose ith entry is \( f(list1[i], list2[i], \ldots, listn[i]) \).

gap> ListN( [1,2], [3,4], \+ );
[ 4, 6 ]
Chapter 21. Lists

21.21 Advanced List Manipulations

The following functions are generalizations of List (see 21.20.16), Set (see 28.2.5), Sum (see 21.20.23), and Product (see 21.20.22).

\[
\text{ListX}( \text{arg1, arg2, \ldots, argn, func})
\]

ListX returns a new list constructed from the arguments.

Each of the arguments \(\text{arg1, arg2, \ldots, argn}\) 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 \(\text{true}\) or \(\text{false}\)
  this introduces a new if-statement in the sequence of nested for-loops and if-statements.

The last argument \(\text{func}\) must be a function, it is applied to the values of the loop-variables and the results are collected.

Thus \(\text{ListX}(\text{list, func})\) is the same as \(\text{List}(\text{list, func})\), and \(\text{ListX}(\text{list, func, x \to x})\) is the same as \(\text{Filtered}(\text{list, func})\).

As a more elaborate example, assume \(\text{arg1}\) is a list or collection, \(\text{arg2}\) is a function returning \(\text{true}\) or \(\text{false}\), \(\text{arg3}\) is a function returning a list or collection, and \(\text{arg4}\) is another function returning \(\text{true}\) or \(\text{false}\), then

\[
\text{result} := \text{ListX}(\text{arg1, arg2, arg3, arg4, func})
\]

is equivalent to

\[
\text{result} := [];
\text{for } \text{v1 in arg1} \text{ do }
  \text{if arg2( v1 ) then }
    \text{for v2 in arg3( v1 ) do }
      \text{if arg4( v1, v2 ) then }
        \text{Add( result, func( v1, v2 ) );}
      \text{fi;}
    \text{od;}
  \text{fi;}
\text{od;}
\]
The following example shows how \texttt{ListX} can be used to compute all pairs and all strictly sorted pairs of elements in a list.

\begin{verbatim}
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 ] ]
\end{verbatim}

In the following example, \texttt{<} is the comparison operation:

\begin{verbatim}
gap> ListX( l, l, <, pair );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
\end{verbatim}

\textbf{21.22 Ranges}

A \textbf{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 \texttt{[ first, second .. last ]} or, if the difference between consecutive elements is 1, as \texttt{[ first .. last ]}.

If \texttt{first > last}, \texttt{[first..last]} is the empty list, which by definition is also a range; also, if \texttt{second > first > last} or \texttt{second < first < last}, then \texttt{[first,second..last]} is the empty list. If \texttt{first = last}, \texttt{[first,second..last]} is a singleton list, which is a range, too. Note that \texttt{last - first} must be divisible by the increment \texttt{second - first}, otherwise an error is signalled.

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 \texttt{last + second-first} to the entry \texttt{range[Length(range)+1]}, the resulting list will no longer be a range.

\begin{verbatim}
gap> r := [10..20];
[ 10 .. 20 ]
gap> Length( r );
11

gap> r[3];
12

gap> 17 in r;
true

gap> r[12] := 25;; r;
[ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25 ] # is no longer a range

gap> r := [1,3..17];
[ 1, 3 .. 17 ]
\end{verbatim}
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 ];
Range: <last>-<first> (31) must be divisible by <inc> (3)

Most often ranges are used in connection with the for-loop (see 4.19). 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.

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.

1 ▶ IsRange( obj ) C

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”).

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

2 ▶ ConvertToRangeRep( list ) F

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` or `Append`, 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.

```gap
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:

```bash
1 ▶ IsQuickPositionList( list )
```

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` in `list` and a bit list than ordered lists to represent subsets of `list`.

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 41.5).

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 28.7).

For constructing enumerators, see 28.2.2 and 28.2.3.
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 blist for brevity.

A boolean list (“blist”) is a list that has no holes and contains only true and false. If a list is known to be a boolean list by a test with IsBlist it is stored in a compact form. See 22.4.

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.

### 22.1 Boolean Lists Representing Subsets

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.

```gap
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 22.3.2.
returns the sublist \( \text{sub} \) of the list \( \text{list} \), which must have no holes, represented by the boolean list \( \text{blist} \), which must have the same length as \( \text{list} \). \( \text{sub} \) contains the element \( \text{list}[i] \) if \( \text{blist}[i] \) is \text{true} and does not contain the element if \( \text{blist}[i] \) is \text{false}. The order of the elements in \( \text{sub} \) is the same as the order of the corresponding elements in \( \text{list} \).

\[
gap> \text{ListBlist}([1..8],[\text{false, true, true, true, false, true, true}]);
\text{[ 2, 3, 4, 5, 7, 8 ]}
\]

returns the number of entries of the boolean list \( \text{blist} \) that are \text{true}. This is the size of the subset represented by the boolean list \( \text{blist} \).

\[
gap> \text{SizeBlist}( [ \text{false, true, false, true, false} ] );
2
\]

returns \text{true} if the boolean list \( \text{blist2} \) is a subset of the boolean list \( \text{list1} \), which must have equal length, and \text{false} otherwise. \( \text{blist2} \) is a subset of \( \text{blist1} \) if \( \text{blist1}[i] = \text{blist1}[i] \) or \( \text{blist2}[i] \) for all \( i \).

\[
gap> \text{blist1} := [ \text{true, true, false, false} ];
gap> \text{blist2} := [ \text{true, false, true, false} ];
gap> \text{IsSubsetBlist}( \text{blist1}, \text{blist2} );
\text{false}
gap> \text{blist2} := [ \text{true, false, false, false} ];
gap> \text{IsSubsetBlist}( \text{blist1}, \text{blist2} );
\text{true}
\]

returns the union of the boolean lists \( \text{blist1}, \text{blist2}, \ldots \), which must have equal length. The union is a new boolean list such that \( \text{union}[i] = \text{blist1}[i] \) or \( \text{blist2}[i] \) or \( \ldots \).

The second form takes the union of all blists (which as for the first form must have equal length) in the list \( \text{list} \).

\[
gap> \text{UnionBlist}( \text{blist1}, \text{blist2}[,...] );
\text{[ 2, 3, 4, 5, 7, 8 ]}
\]

returns the intersection of the boolean lists \( \text{blist1}, \text{blist2}, \ldots \), which must have equal length. The intersection is a new boolean list such that \( \text{inter}[i] = \text{blist1}[i] \) and \( \text{blist2}[i] \) and \( \ldots \).

In the second form \( \text{list} \) must be a list of boolean lists \( \text{blist1}, \text{blist2}, \ldots \), which must have equal length, and \text{IntersectionBlist} returns the intersection of those boolean lists.

\[
gap> \text{IntersectionBlist}( \text{blist1}, \text{blist2}[,...] );
\text{[ 2, 3, 4, 5, 7, 8 ]}
\]

returns the asymmetric set difference (exclusive or) of the two boolean lists \( \text{blist1} \) and \( \text{blist2} \), which must have equal length. The asymmetric set difference is a new boolean list such that \( \text{union}[i] = \text{blist1}[i] \) and not \( \text{blist2}[i] \).

\[
gap> \text{DifferenceBlist}( \text{blist1}, \text{blist2} );
\text{[ 2, 3, 4, 5, 7, 8 ]}
\]
Chapter 22. Boolean Lists

```
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 ]
```

22.3 Function that Modify Boolean Lists

1 ▶ UniteBlist( blist1, blist2 ) F

UniteBlist unites the boolean list blist1 with the boolean list blist2, which must have the same length. This is equivalent to assigning \( \text{blist1}[i] := \text{blist1}[i] \text{ or blist2}[i] \) for all \( i \). UniteBlist returns nothing, it is only called to change blist1.

```
gap> blist1 := [ true, true, false, false ];
gap> blist2 := [ true, false, true, false ];
gap> UniteBlist( blist1, blist2 );

[ true, true, true, false ]
```

2 ▶ UniteBlistList( list, blist, sub ) F

works like UniteBlist(blist,BlistList(list,sub)). As no intermediate blist is created, the performance is better than the separate function calls.

The function UnionBlist (see 22.2.1) is the nondestructive counterpart to the procedure UniteBlist.

3 ▶ IntersectBlist( blist1, blist2 ) F

intersects the boolean list blist1 with the boolean list blist2, which must have the same length. This is equivalent to assigning \( \text{blist1}[i] := \text{blist1}[i] \text{ and blist2}[i] \) for all \( i \). IntersectBlist returns nothing, it is only called to change blist1.

```
gap> blist1 := [ true, true, false, false ];
gap> blist2 := [ true, false, true, false ];
gap> IntersectBlist( blist1, blist2 );

[ true, false, false, false ]
```

The function IntersectionBlist (see 22.2.2) is the nondestructive counterpart to the procedure IntersectBlist.

4 ▶ SubtractBlist( blist1, blist2 ) F

subtracts the boolean list blist2 from the boolean list blist1, which must have equal length. This is equivalent to assigning \( \text{blist1}[i] := \text{blist1}[i] \text{ and not blist2}[i] \) for all \( i \). SubtractBlist returns nothing, it is only called to change blist1.

```
gap> blist1 := [ true, true, false, false ];
gap> blist2 := [ true, false, true, false ];
gap> SubtractBlist( blist1, blist2 );

[ false, true, false, false ]
```

The function DifferenceBlist (see 22.2.3) is the nondestructive counterpart to the procedure SubtractBlist.
22.4 More about Boolean Lists

We defined a boolean list as a list that has no holes and contains only \texttt{true} and \texttt{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 \texttt{true} and reset if the entry is \texttt{false}. This representation is of course much more compact than the ordinary representation of lists, which needs (at least) 32 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.

The results of \texttt{BlistList}, \texttt{UnionBlist}, \texttt{IntersectionBlist} and \texttt{DifferenceBlist} are known to be boolean lists by construction, and thus are represented in the compact representation upon creation.

If an argument of \texttt{IsBlist}, \texttt{IsSubsetBlist}, \texttt{ListBlist}, \texttt{UnionBlist}, \texttt{IntersectionBlist}, \texttt{DifferenceBlist}, \texttt{UniteBlist}, \texttt{IntersectBlist} and \texttt{SubtractBlist} 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, \texttt{IsBlist} returns \texttt{false} and the other functions signal an error. 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 \texttt{Add} (see 21.4.4) 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.
Just as in mathematics, a vector in GAP is any object which supports appropriate addition and scalar multiplication operations (see Chapter 59). 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.

```
1 ▶ IsRowVector( obj )
```

A row vector is a vector (see 30.14.14) that is also a homogeneous list. Typical examples are lists of integers and rationals, lists of finite field elements of the same characteristic, lists of polynomials from a common polynomial ring, and matrices.

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 scalar product via `*`; for example, matrices are row vectors but the matrix product 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` for all entries of the row vector (see the comment for `IsVector` in 30.14.14).

Note that methods for special types of row vectors really must be installed with the requirement `IsRowVector`, since `IsVector` may lead to a rank of the method below that of the default method for row vectors (see file `vecmat.gi`).

```
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.

### 23.1 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`, and the multiplicative behaviour is defined only for lists in the category `IsMultiplicativeGeneralizedRowVector` (see 21.12).

```
1 ▶ 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 \( \text{vec1} \) and a vector \( \text{vec2} \) over a finite field yields the list of pointwise sums, which will be a mixture of finite field elements and integers if \( \text{vec1} \) is longer than \( \text{vec2} \).

\[ \text{vec1} + \text{vec2} - \text{scalar} \]

returns the sum of the scalar \( \text{scalar} \) and the row vector \( \text{vec} \). Probably the most usual situation is that the elements of \( \text{vec} \) lie in a common field with \( \text{scalar} \); 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.

\[ [1, 2, 3] + [1/2, 1/3, 1/4]; \]
\[ [3/2, 7/3, 13/4]; \]
\[ [1/2, 3/2, 1/2] + 1/2; \]
\[ [1, 2, 1]; \]

Subtracting a vector or scalar is defined as adding its additive inverse, so the statements for the addition hold likewise.

\[ [1, 2, 3] - [1/2, 1/3, 1/4]; \]
\[ [1/2, 5/3, 11/4]; \]
\[ [1/2, 3/2, 1/2] - 1/2; \]
\[ [0, 1, 0]; \]

returns the product of the scalar \( \text{scalar} \) and the row vector \( \text{vec} \). Probably the most usual situation is that the elements of \( \text{vec} \) lie in a common field with \( \text{scalar} \); 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.

\[ [1, 2, 3] * [1/2, 1/3, 1/4]; \]
\[ 23/12; \]
\[ [1/2, 3/2, 1/2] * 2; \]
\[ [1, 3, 1]; \]

returns the standard scalar product of \( \text{vec1} \) and \( \text{vec2} \), i.e., the sum of the products of the corresponding entries of the vectors. Probably the most usual situation is that \( \text{vec1} \) and \( \text{vec2} \) 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.

For the mutability of results of arithmetic operations, see 12.6.

Further operations with vectors as operands are defined by the matrix operations (see 24.2).
NormedRowVector( v )

returns a scalar multiple \( w = c \cdot v \) of the row vector \( v \) with the property that the first nonzero entry of \( w \) is an identity element in the sense of \( \text{IsOne} \).

gap> \text{NormedRowVector}([5,2,3]);
[ 1, 2/5, 3/5 ]

23.2 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 \( \text{ConvertToVectorRep} \) 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.

ConvertToVectorRep( list ) \quad F
ConvertToVectorRep( list , field ) \quad F
ConvertToVectorRep( list , fieldsize ) \quad F

\( \text{ConvertToVectorRep} \) converts \( \text{list} \) to an internal vector representation if possible.

\( \text{ConvertToVectorRep} \) converts \( \text{list} \) to an internal vector representation appropriate for a vector over \( \text{field} \). It is forbidden to call this function unless \( \text{list} \) is a plain list or a vector, \( \text{field} \) a field, and all elements of \( \text{list} \) lie in \( \text{field} \), violation of this condition can lead to unpredictable behaviour or a system crash.

Instead of a \( \text{field} \) also its size \( \text{fieldsize} \) may be given.

\( \text{list} \) may already be a compressed vector. In this case, if no \( \text{field} \) or \( \text{fieldsize} \) is given, then nothing happens. If one is given then the vector is rewritten as a compressed vector over the given \( \text{field} \) unless it has the filter \( \text{IsLockedRepresentationVector} \), 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. Otherwise it is \text{true} if the vector does consist entirely of elements of finite fields and \text{fail} otherwise.

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).

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> \text{RepresentationsOfObject(v)};
[ "IS_PLIST_REP", "IsInternalRep" ]
gap> \text{ConvertToVectorRep(v)};
2
gap> v;
<\text{a GF2 vector of length 4}>
gap> \text{ConvertToVectorRep(v,4)};
4
gap> v;
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> \text{RepresentationsOfObject(v)};
[ "IsDataObjectRep", "Is8BitVectorRep" ]
Section 3. Coefficient List Arithmetic

A vector in the special representation over $GF(2)$ is always viewed as a 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.

2 ▶ NumberFFVector( vec, sz )
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.

23.3 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 in preference to arithmetic operators. Firstly, the three argument form has no single-step equivalent. Secondly AddRowVector changes its first argument in-place, rather than allocating a new vector to hold the result, and may thus produce less garbage.

1 ▶ AddRowVector( dst, src, [mul [, from, to]] )

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.

2 ▶ AddCoeffs( list1, poss1, list2, poss2, mul )
 ▶ AddCoeffs( list1, list2, mul )
 ▶ AddCoeffs( list1, list2 )

AddCoeffs adds the entries of list2{poss2}, multiplied by the scalar mul, to list1{poss1}. Non-existing 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 the 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.

    gap> l:=[1,2,3,4];;m:=[5,6,7];;AddCoeffs(l,m);
    4
    gap> l;
    [ 6, 8, 10, 4 ]

3 ▶ MultRowVector( list1, poss1, list2, poss2, mul )
 ▶ MultRowVector( list, mul )

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 list, in-place, by mul.

4 ▶ CoeffsMod( list1, [len1 , ] mod )

returns the coefficient list obtained by reducing the entries in list1 modulo mod. After reducing it shrinks the list to remove trailing zeroes.

    gap> l:=[1,2,3,4];;CoeffsMod(l,2);
    [ 1, 0, 1 ]
23.4 Shifting and Trimming Coefficient Lists

The following functions change coefficient lists by shifting or trimming.

1. \texttt{LeftShiftRowVector( list, shift )}
   changes list by assigning \( \texttt{list}[i] := \texttt{list}[i+\text{shift}] \) and removing the last \( \text{shift} \) entries of the result.

2. \texttt{RightShiftRowVector( list, shift, fill )}
   changes list by assigning \( \texttt{list}[i+\text{shift}] := \texttt{list}[i] \) and filling each of the \( \text{shift} \) first entries with \( \text{fill} \).

3. \texttt{ShrinkRowVector( list )}
   removes trailing zeroes from the list \( \texttt{list} \).

4. \texttt{RemoveOuterCoeffs( list, coef )}
   removes \( \text{coef} \) at the beginning and at the end of \( \texttt{list} \) and returns the number of elements removed at the beginning.

   \texttt{gap> l:=[1,1,2,1,2,1,1,2,1];;RemoveOuterCoeffs(l,1);}
   2
   \texttt{gap> l;}
   [ 2, 1, 2, 1, 1, 2 ]

23.5 Functions for Coding Theory

The following functions perform operations on Finite fields vectors considered as code words in a linear code.

1. \texttt{WeightVecFFE( vec )}
   returns the weight of the finite field vector \( \texttt{vec} \), i.e. the number of nonzero entries.

2. \texttt{DistanceVecFFE( vec1, vec2 )}
   returns the distance between the two vectors \( \texttt{vec1} \) and \( \texttt{vec2} \), which must have the same length and whose elements must lie in a common field. The distance is the number of places where \( \texttt{vec1} \) and \( \texttt{vec2} \) differ.

3. \texttt{DistancesDistributionVecFFEVecFFE( vecs, vec )}
   returns the distances distribution of the vector \( \texttt{vec} \) to the vectors in the list \( \texttt{vecs} \). 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 \( \text{Length}(\texttt{vec})+1 \), such that the value \( d[i] \) is the number of vectors in \( \texttt{vecs} \) that have distance \( i+1 \) to \( \texttt{vec} \).

4. \texttt{DistancesDistributionMatFFEVecFFE( mat, f, vec )}
   returns the distances distribution of the vector \( \texttt{vec} \) to the vectors in the vector space generated by the rows of the matrix \( \texttt{mat} \) over the finite field \( f \). The length of the rows of \( \texttt{mat} \) and the length of \( \texttt{vec} \) must be equal, and all elements must lie in \( f \). The rows of \( \texttt{mat} \) must be linearly independent. The distances distribution is a list \( d \) of length \( \text{Length}(\texttt{vec})+1 \), such that the value \( d[i] \) is the number of vectors in the vector space generated by the rows of \( \texttt{mat} \) that have distance \( i+1 \) to \( \texttt{vec} \).

5. \texttt{AClosestVectorCombinationsMatFFEVecFFE( mat, f, vec, l, stop )}
   runs through the \( f \)-linear combinations of the vectors in the rows of the matrix \( \texttt{mat} \) that can be written as linear combinations of exactly \( l \) rows (that is without using zero as a coefficient) and returns a vector from these that is closest to the vector \( \texttt{vec} \). The length of the rows of \( \texttt{mat} \) and the length of \( \texttt{vec} \) must be equal, and all elements must lie in \( f \). The rows of \( \texttt{mat} \) must be linearly independent. If it finds a vector of distance at most \( \text{stop} \), which must be a nonnegative integer, then it stops immediately and returns this vector.

6. \texttt{CosetLeadersMatFFE( mat, f )}
   returns a list of representatives of minimal weight for the cosets of a code. \( \texttt{mat} \) must be a \textbf{check matrix} for the code, the code is defined over the finite field \( f \). All rows of \( \texttt{mat} \) must have the same length, and all elements must lie in \( f \). The rows of \( \texttt{mat} \) must be linearly independent.
23.6 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 64), 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.

\begin{verbatim}
1 ▶ ValuePol( coeff, x )
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.

   gap> ValuePol([1,2,3],4);
57

2 ▶ ProductCoeffs( list1 [, len1], list2 [, len2] )
Let pol1 (and pol2) 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 pol1 and pol2.

   gap> l:= [1,2,3,4];; m:= [5,6,7];; ProductCoeffs(l,m);
   [ 5, 16, 34, 52, 45, 28 ]

3 ▶ ReduceCoeffs( list1 [, len1], list2 [, len2] )
changes list1 to the coefficient list of the remainder when dividing pol1 by pol2. 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.

   gap> l:= [1,2,3,4];; m:= [5,6,7];; ReduceCoeffs(l,m);
   2
   gap> l;
   [ 64/49, -24/49, 0, 0 ]

4 ▶ ReduceCoeffsMod( list1 [, len1], list2 [, len2], mod )
changes list1 to the coefficient list of the remainder when dividing pol1 by pol2 modulo mod. mod 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.

   gap> l:= [1,2,3,4];; m:= [5,6,7];; ReduceCoeffsMod(l,m,3);
   1
   gap> l;
   [ 1, 0, 0, 0 ]

5 ▶ PowerModCoeffs( list1[], len1, exp, list2[], len2 )
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
\end{verbatim}
power of $p_1$ by $p_2$. The coefficients are reduced already while powers are computed, therefore avoiding an explosion in list length.

```gap
gap> l:=[1,2,3,4];;m:=[5,6,7];;PowerModCoeffs(l,5,m);
[ -839462813696/678223072849, -7807439437824/678223072849, 0 ]
```

6. **ShiftedCoeffs(list, shift)**

produces a new coefficient list new obtained by the rule new[i+shift]:=list[i] and filling initial holes by the appropriate zero.

```gap
gap> l:=[1,2,3];;ShiftedCoeffs(l,2);ShiftedCoeffs(l,-2);
[ 0, 0, 1, 2, 3 ]
[ 3 ]
```

7. **ShrinkCoeffs(list)**

removes trailing zeroes from list. It returns the position of the last non-zero entry, that is the length of list after the operation.

```gap
gap> l:=[1,0,0];;ShrinkCoeffs(l);1
1
[ 1 ]
```
Matrices are represented in GAP by lists of row vectors (see 23). The vectors must all have the same length, and their elements must lie in a common ring.

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 for matrices is just a special case of ShallowCopy for lists (see 21.7); called with an immutable matrix mat, ShallowCopy 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 ).

The info class for matrix operations is InfoMatrix.

### 24.1 Categories of Matrices

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.

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 \textit{IsOrdinaryMatrix}, see 24.1.2.) Each matrix created by \texttt{LieObject} is in the category \textit{IsLieMatrix}, and arithmetic operations with objects in \textit{IsLieMatrix} produce again matrices in \textit{IsLieMatrix}.

### 24.2 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 \textit{IsGeneralizedRowVector}, and the multiplicative behaviour is defined only for lists in the category \textit{IsMultiplicativeGeneralizedRowVector} (see 21.12).

1. \texttt{mat1 + mat2}

returns the sum of the two matrices \texttt{mat1} and \texttt{mat2}, Probably the most usual situation is that \texttt{mat1} and \texttt{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 \texttt{mat1} and a matrix \texttt{mat2} over a finite field yields the table of pointwise sums, which will be a mixture of finite field elements and integers if \texttt{mat1} has bigger dimensions than \texttt{mat2}.

2. \texttt{scalar + mat}

returns the sum of the scalar \texttt{scalar} and the matrix \texttt{mat}. Probably the most usual situation is that the entries of \texttt{mat} lie in a common field with \texttt{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.

3. \texttt{mat1 - mat2}

Subtracting a matrix or scalar is defined as adding its additive inverse, so the statements for the addition hold likewise.

4. \texttt{scalar * mat}

returns the product of the scalar \texttt{scalar} and the matrix \texttt{mat}. Probably the most usual situation is that the elements of \texttt{mat} lie in a common field with \texttt{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.
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.

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.

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 \(mat1[i][l] \cdot mat2[l][j]\), with \(l\) running from 1 to \(n\).

returns the inverse of the matrix mat, which must be an invertible square matrix. If mat is not invertible then fail is returned.

In general, \(left \div right\) is defined as \(left \cdot right^{-1}\). Thus in the above forms the right operand must always be invertible.

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 \(mat2^{-1} \cdot mat1 \cdot mat2\).

Powering a row vector vec by a matrix mat is in every respect equivalent to vec \(\cdot mat\). 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.

returns the commutator of the square invertible matrices mat1 and mat2 of the same dimensions and over a common field, which is the matrix \(mat1^{-1} \cdot mat2^{-1} \cdot mat1 \cdot mat2\).
Chapter 24. Matrices

The following cases are still special cases of the general list arithmetic defined in 21.11.

12 ▶ 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.

13 ▶ 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.

14 ▶ matlist / mat

Dividing a list matlist of matrices by an invertible matrix mat evaluates to matlist * mat^-1.

15 ▶ 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.

24.3 Properties and Attributes of Matrices

1 ▶ DimensionsMat( mat )

is a list of length 2, the first being the number of rows, the second being the number of columns of the matrix mat.

    gap> DimensionsMat([[1,2,3],[4,5,6]]);
    [ 2, 3 ]

2 ▶ DefaultFieldOfMatrix( mat )

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 57.3 and 18.1).

    gap> DefaultFieldOfMatrix([[Z(4),Z(8)]]);
    GF(2^6)

3 ▶ TraceMat( mat )
▶ Trace( mat )

The trace of a square matrix is the sum of its diagonal entries.
Section 4. Matrix Constructions

```gap
gap> TraceMat([[1,2,3],[4,5,6],[7,8,9]]);
15
```

4► **DeterminantMat** ( **mat** )

4► **Determinant** ( **mat** )

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 24.3.6.

5► **DeterminantMatDestructive** ( **mat** )

Does the same as **DeterminantMat**, with the difference that it may destroy its argument. The matrix **mat** must be mutable.

```gap
gap> DeterminantMat([[1,2],[2,1]]);
-3
```

6► **DeterminantMatDivFree** ( **mat** )

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].

7► **IsMonomialMatrix** ( **mat** )

A matrix is monomial if and only if it has exactly one nonzero entry in every row and every column.

```gap
gap> IsMonomialMatrix([[0,1],[1,0]]);
true
```

8► **IsDiagonalMat** ( **mat** )

returns true if **mat** has only zero entries off the main diagonal, false otherwise.

9► **IsUpperTriangularMat** ( **mat** )

returns true if **mat** has only zero entries below the main diagonal, false otherwise.

10► **IsLowerTriangularMat** ( **mat** )

returns true if **mat** has only zero entries below the main diagonal, false otherwise.

24.4 Matrix Constructions

1► **IdentityMat** ( **m** [, **F**] )

returns a (mutable) $m \times m$ identity matrix over the field given by **F** (i.e. the smallest field containing the element **F** or **F** itself if it is a field).

2► **NullMat** ( **m** , **n** [, **F**] )

returns a (mutable) $m \times n$ null matrix over the field given by **F**.
Chapter 24. Matrices

```gap
gap> IdentityMat(3,1);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> NullMat(3,2,Z(3));
[ [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ] ]
```

3 ▶ EmptyMatrix( `char` )

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 `^`) on empty lists.

```gap
gap> EmptyMatrix(5);
EmptyMatrix( 5 )
gap> AsList(last);
[ ]
```

4 ▶ DiagonalMat( `vector` )

returns a diagonal matrix `mat` with the diagonal entries given by `vector`.

```gap
gap> DiagonalMat([1,2,3]);
[ [ 1, 0, 0 ], [ 0, 2, 0 ], [ 0, 0, 3 ] ]
```

5 ▶ PermutationMat( `perm`, `dim` [, `F`] )

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.

```gap
gap> PermutationMat((1,2,3),4,1);
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ] ]
```

6 ▶ TransposedMatImmutable( `mat` )
▶ TransposedMatAttr( `mat` )
▶ TransposedMat( `mat` )
▶ TransposedMatMutable( `mat` )
▶ TransposedMatOp( `mat` )

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 (see 30.10.3).

7 ▶ TransposedMatDestructive( `mat` )

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.
Section 4. Matrix Constructions

\[
gap\text{TransposedMat([[1,2,3],[4,5,6],[7,8,9]]);}\n\]
\[
\text{[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]}\n\]
\[
gap\text{mm:= [[1,2,3],[4,5,6],[7,8,9]];}\n\]
\[
gap\text{TransposedMatDestructive( mm );}\n\]
\[
\text{[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]}\n\]
\[
gap\text{mm;}\n\]
\[
\text{[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]}\n\]

8\text{KroneckerProduct( mat1, mat2 )}\n
The Kronecker product of two matrices is the matrix obtained when replacing each entry \( a \) of \( \text{mat1} \) by the product \( a \times \text{mat2} \) in one matrix.

\[
gap\text{KroneckerProduct([[1,2]],[[5,7],[9,2]]);}\n\]
\[
\text{[ [ 5, 7, 10, 14 ], [ 9, 2, 18, 4 ] ]}\n\]

9\text{ReflectionMat( coeffs )}\n\text{ReflectionMat( coeffs, root )}\n\text{ReflectionMat( coeffs, conj )}\n\text{ReflectionMat( coeffs, conj, root )}\n
Let \( \text{coeffs} \) be a row vector. \text{ReflectionMat} returns the matrix of the reflection in this vector.

More precisely, if \( \text{coeffs} \) is the coefficients of a vector \( v \) w.r.t. a basis \( B \) (see 59.4.2), 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 \( \text{root} \) 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 18.1.1).

\( \text{conj} \) is a function of one argument that conjugates a ring element. The default is \text{ComplexConjugate}.

The matrix of the reflection in \( v \) is defined as

\[
M = I_n + v v^\text{conj} \cdot \frac{w - 1}{w} \cdot v
\]

where \( w = \text{root}, n \) is the length of the coefficient list, and \( \text{conj} \) denotes the conjugation.

10\text{PrintArray( array )}\n
pretty-prints the array \( \text{array} \).

11\text{MutableIdentityMat( m [, F] )}\n
returns a (mutable) \( m \times m \) identity matrix over the field given by \( F \). This is identical to \text{IdentityMat} and is present in \text{GAP} 4.1 only for the sake of compatibility with beta-releases. It should not be used in new code.

12\text{MutableNullMat( m, n [, F] )}\n
returns a (mutable) \( m \times n \) null matrix over the field given by \( F \). This is identical to \text{NullMat} and is present in \text{GAP} 4.1 only for the sake of compatibility with beta-releases. It should not be used in new code.
24.5 Random Matrices

1. ▶ RandomMat( m, n [, , R ] )

RandomMat returns a new mutable random matrix with m rows and n columns with elements taken from the ring R, which defaults to Integers.

2. ▶ RandomInvertibleMat( m [, , R ] )

RandomInvertibleMat returns a new mutable invertible random matrix with m rows and columns with elements taken from the ring R, which defaults to Integers.

3. ▶ RandomUnimodularMat( m )

returns a new random mutable m×m matrix with integer entries that is invertible over the integers.

gap> RandomMat(2,3,GF(3));
[ [ Z(3)^0, Z(3), Z(3)^0 ], [ Z(3), Z(3)^0, Z(3)^0 ] ]
gap> RandomInvertibleMat(4);
[ [ -1, 0, 1, -1 ], [ 2, 1, 3, 0 ], [ 1, 4, 0, 2 ], [ -3, 2, 1, 0 ] ]

24.6 Matrices Representing Linear Equations and the Gaussian Algorithm

1. ▶ RankMat( mat )

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.

gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> RankMat(mat);
2

2. ▶ TriangulizeMat( mat )

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”).

gap> m:=TransposedMatMutable(mat);
[ [ 1, 4, 7 ], [ 2, 5, 9 ], [ 3, 6, 9 ] ]
gap> TriangulizeMat(m);m;
[ [ 1, 0, -1 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]

3. ▶ NullspaceMat( mat )
▶ TriangulizedNullspaceMat( mat )

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.

4. ▶ NullspaceMatDestructive( mat )
▶ TriangulizedNullspaceMatDestructive( mat )

This function does the same as NullspaceMat. However, the latter function makes a copy of mat to avoid having to change it. This function does not do that; it returns the null space 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.

    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 ] ]

SolutionMat(mat, vec ) returns a row vector x that is a solution of the equation x * mat = vec. It returns fail if no such vector exists.

SolutionMatDestructive(mat, vec ) does the same as SolutionMat(mat, vec ) except that it may destroy the matrix mat. The matrix mat must be mutable.

    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> SolutionMatDestructive(mm,[3,5,7]);
    [ 5/3, 1/3, 0 ]
    gap> mm;
    [ [ 1, 2, 3 ], [ 0, -3, -6 ], [ 0, 0, 0 ] ]

BaseFixedSpace(mats ) 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.)

    gap> BaseFixedSpace([[1,2],[0,1]]);
    [ [ 0, 1 ] ]

24.7 Eigenvectors and eigenvalues

GeneralisedEigenvalues( F, A ) The generalised eigenvalues of the matrix A over the field F.

GeneralizedEigenvalues( F, A )

    gap> GeneralisedEigenvalues( F, A );

GeneralisedEigenspaces( F, A ) The generalised eigenspaces of the matrix A over the field F.

GeneralizedEigenspaces( F, A )

    gap> GeneralizedEigenspaces( F, A );

Eigenvectors( F, A ) The eigenspaces of the matrix A over the field F.

Eigenspaces( F, A )

    gap> Eigenvectors( F, A );

    gap> Eigenspaces( F, A );
24.8 Elementary Divisors

ElementaryDivisorsMat( [ring, ] mat )

ElementaryDivisors 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 ring $ring$, which must contain all matrix entries. For compatibility reasons it can be omitted and defaults to Integers.

```gap
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];
gap> ElementaryDivisorsMat(mat);
[ 1, 3, 0 ]
```

DiagonalizeMat( ring, mat )

brings the mutable matrix $mat$, considered as a matrix over $ring$, into diagonal form by elementary row and column operations.

```gap
gap> m:=[[1,2],[2,1]];
gap> DiagonalizeMat(Integers,m);m;
[ [ 1, 0 ], [ 0, 3 ] ]
```

See also chapter 25

24.9 Echelonized Matrices

SemiEchelonMat( mat )

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 $i$.

SemiEchelonMatDestructive( mat )

This does the same as SemiEchelonMat( mat ), except that it may (and probably will) destroy the matrix $mat$.

```gap
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 ] ]
```

SemiEchelonMatTransformation( mat )

does the same as SemiEchelonMat but additionally stores the linear transformation $T$ performed on the matrix. The additional components of the result are
Section 10. Matrices as Basis of a Row Space

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.

gap> SemiEchelonMatTransformation([[1,2,3],[0,0,1]]);
rec( heads := [ 1, 0, 2 ], vectors := [ [ 1, 2, 3 ], [ 0, 0, 1 ] ],
    coeffs := [ [ 1, 0 ], [ 0, 1 ] ], relations := [ ] )

4▶ SemiEchelonMats( mats )
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 24.9.1).
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

5▶ SemiEchelonMatsDestructive( mats )
Does the same as SemiEchelonMats, except that it may destroy its argument. Therefore the argument must be a list of matrices that are mutable.

24.10 Matrices as Basis of a Row Space

1▶ BaseMat( mat )
returns a basis for the row space generated by the rows of mat in the form of an immutable matrix.

2▶ BaseMatDestructive( mat )
Does the same as BaseMat, with the difference that it may destroy the matrix mat. The matrix mat must be mutable.

gap> BaseMat(mat);
[ [ 1, 2, 3 ], [ 0, 1, 2 ] ]
gap> mm:=[[1,2,3],[4,5,6],[5,7,9]];

3▶ BaseOrthogonalSpaceMat( mat )
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.
Chapter 24. Matrices

4\textbf{SumIntersectionMat}(\textit{M1, M2})

performs Zassenhaus’ algorithm to compute bases for the sum and the intersection of spaces generated by
the rows of the matrices \textit{M1}, \textit{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 24.9).

\textbf{gap}\textgreater\textbf{SumIntersectionMat(mat,[[2,7,6],[5,9,4]]);} 
\texttt{[[[1, 2, 3], [0, 1, 2], [0, 0, 1]], [[1, -3/4, -5/2]]]}

5\textbf{BaseSteinitzVectors}(\textit{bas, mat})

find vectors extending \textit{mat} to a basis spanning the span of \textit{bas}. Both \textit{bas} and \textit{mat} must be matrices of full
(row) rank. It returns a record with the following components:

\textbf{subspace}

is a basis of the space spanned by \textit{mat} in upper triangular form with leading ones at all echelon
steps and zeroes above these ones.

\textbf{factorspace}

is a list of extending vectors in upper triangular form.

\textbf{factorzero}

is a zero vector.

\textbf{heads}

is a list of integers which can be used to decompose vectors in the basis vectors. The \textit{i}th entry
indicating the vector that gives an echelon step at position \textit{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 \textbf{subspace} and \textbf{factorspace}.

\textbf{gap}\textgreater\textbf{BaseSteinitzVectors(IdentityMat(3,1),[[11,13,15]]);} 
\texttt{rec( factorspace := [[0, 1, 15/13], [0, 0, 1]], factorzero := [0, 0, 0], subspace := [[1, 13/11, 15/11]], heads := [-1, 1, 2])}

See also chapter 25

24.11 Triangular Matrices

1\textbf{DiagonalOfMat}(\textit{mat})

returns the diagonal of \textit{mat} as a list.

\textbf{gap}\textgreater\textbf{DiagonalOfMat([[1,2],[3,4]]);} 
\texttt{[1, 4]}

2\textbf{UpperSubdiagonal}(\textit{mat, pos})

returns a mutable list containing the entries of the \textit{pos}th upper subdiagonal of \textit{mat}.

\textbf{gap}\textgreater\textbf{UpperSubdiagonal(mat,1);} 
\texttt{[2, 6]}

3\textbf{DepthOfUpperTriangularMatrix}(\textit{mat})

If \textit{mat} is an upper triangular matrix this attribute returns the index of the first nonzero diagonal.

\textbf{gap}\textgreater\textbf{DepthOfUpperTriangularMatrix([[0,1,2],[0,0,1],[0,0,0]]);} 
1
\textbf{gap}\textgreater\textbf{DepthOfUpperTriangularMatrix([[0,0,2],[0,0,0],[0,0,0]]);} 
2
24.12 Matrices as Linear Mappings

For a square matrix \( mat \), \( \text{CharacteristicPolynomial} \) returns the characteristic polynomial of \( mat \), that is, the \( \text{StandardAssociate} \) of the determinant of the matrix \( mat - X \cdot I \), where \( X \) is an indeterminate and \( I \) is the appropriate identity matrix.

If a field \( F \) is given as first argument then the characteristic polynomial of the \( F \)-linear mapping induced by \( mat \) is computed. If \( F \) contains the entries of \( mat \) then this is of course the same polynomial as the one computed by the one argument version; if \( F \) is a proper subfield of the default field (see 24.3.2) of \( mat \) then the characteristic polynomial is computed using \( \text{BlownUpMat} \) (see 24.12.3).

The returned polynomials are expressed in the indeterminate number \( ind \). If \( ind \) is not given, it defaults to 1.

The characteristic polynomial is a multiple of the minimal polynomial (see 64.8.1).

\[
\text{gap> CharacteristicPolynomial( } [ [ 1, 1 ], [ 0, 1 ] ] \); 1-2*x_1+x_1^2
\]

\[
\text{gap> mat := } [[0,1],[E(4)-1,E(4)]];
\text{gap> CharacteristicPolynomial( mat ); 1-E(4)+E(4)*x_1+x_1^2}
\]

\[
\text{gap> CharacteristicPolynomial( Rationals, mat ); 2+2*x_1+3*x_1^2+x_1^4}
\]

\[
\text{gap> mat:= [ [ E(4), 1 ], [ 0, -E(4) ] ];;}
\text{gap> CharacteristicPolynomial( mat ); 1+x_1^2}
\]

\[
\text{gap> CharacteristicPolynomial( Rationals, mat ); 1+2*x_1^2+x_1^4}
\]

\[
2 \text{ ▶ } \text{JordanDecomposition( } mat \text{ )}
\]

\( \text{JordanDecomposition( } mat \text{ ) returns a list } [S,N] \text{ such that } S \text{ is a semisimple matrix and } N \text{ is nilpotent. Furthermore, } S \text{ and } N \text{ commute and } mat=S+N. \)

\[
\text{gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;}
\text{gap> JordanDecomposition(mat);}
[[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
 [ [ 0, 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ] ]
\]

\[
3 \text{ ▶ } \text{BlownUpMat( } B, \text{ mat } \)
\]

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.) \( \text{BlownUpMat} \) 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 \( K^m \) w.r.t. the \( K \)-basis whose vectors are \( (b_1v_1, \ldots, b_mv_n) \).

Note that the linear transformations act on row vectors, i.e., each row of the matrix is a concatenation of vectors of \( B \)-coefficients.

\[
4 \text{ ▶ } \text{BlownUpVector( } B, \text{ vector } \)
\]

Let \( B \) be a basis of a field extension \( F/K \), and \( vector \) a row vector whose entries are all in \( F \). \( \text{BlownUpVector} \) returns a row vector over \( K \) that is obtained by replacing each entry of \( vector \) by its coefficients w.r.t. \( B \).
So \texttt{BlownUpVector} and \texttt{BlownUpMat} (see 24.12.3) are compatible in the sense that for a matrix \texttt{mat} over \(F\), \(\texttt{BlownUpVector}(B, \texttt{mat} \ast \texttt{vector})\) is equal to \(\texttt{BlownUpMat}(B, \texttt{mat}) \ast \texttt{BlownUpVector}(B, \texttt{vector})\).

\begin{verbatim}
gap> B := Basis( CF(4), [ 1, E(4) ] );
gap> mat := [ [ 1, E(4) ], [ 0, 1 ] ];
gap> vec := [ 1, E(4) ];
gap> bmat := BlownUpMat( B, mat );
gap> bvec := BlownUpVector( B, vec );
gap> Display( bmat ); bvec;
gap> bvec * bmat = BlownUpVector( B, vec * mat );
true
\end{verbatim}

\texttt{CompanionMat(\, poly\,)} \(F\) computes a companion matrix of the polynomial \(\texttt{poly}\). This matrix has \(\texttt{poly}\) as its minimal polynomial.

### 24.13 Matrices over Finite Fields

Just as for row vectors, (see section 23.2), 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.

\begin{verbatim}
gap> v := Z(2) * [1,0,0,1,1];
gap> ConvertToVectorRep(v,2);
gap> m := [v];
gap> m := [v,v];
gap> m := [v,v,v];
gap> m := [1,0,0,1,1];
gap> ConvertToVectorRep(v,2);
gap> m := [v];
gap> m := [v,v];
gap> m := [v,v,v];
gap> m := [1,0,0,1,1];
gap> ConvertToVectorRep(v);
gap> m := [v];
gap> m := [v,v];
gap> m := [v,v,v];
gap> m := [1,0,0,1,1];
gap> RepresentationsOfObject(m);
gap> "IsPositionalObjectRep", "Is8BitMatrixRep"
gap> m := [v,v,v];
gap> ConvertToMatrixRep(m,GF(3));
gap> RepresentationsOfObject(m);
gap> "IsPositionalObjectRep", "Is8BitMatrixRep"
\end{verbatim}

All compressed matrices over GF(2) are viewed as \(<\texttt{n} \times \texttt{m} \text{ matrix over GF2}\>\), 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.

1. `ImmutableMatrix(field, matrix, [change])` returns an immutable matrix equal to `matrix` which is in the most compact representation possible over `field`. The input matrix `matrix` or its rows might change the representation, however the result of `ConvertedMatrix` 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).

2. `ConvertToMatrixRep(list)` converts `list` to an internal matrix representation if possible. `ConvertToMatrixRep(list, field)` converts `list` to an internal matrix representation appropriate for a matrix over `field`. It is forbidden to call this function unless all elements of `list` are vectors with entries in `field`.

Instead of a `field` also its size `fields`ize may be given.

`list` may already be a compressed matrix. In this case, if no `field` or `fields`ize is given, then nothing happens. `list` itself may be mutable, but its entries must be immutable.

The return value is the size of the field over which the matrix ends up written, if it is written in a compressed representation. Otherwise it is `fail`.

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`, `ConvertToVectorRep` and `ConvertToMatrixRep` are all prevented from altering a vector with this filter.
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.

\begin{itemize}
\item \texttt{ProjectiveOrder(mat)}
\end{itemize}

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.

\begin{verbatim}
gap> ProjectiveOrder([[1,4],[5,2]]*Z(11)^0); [ 5, Z(11)^5 ]
\end{verbatim}

\begin{itemize}
\item \texttt{SimultaneousEigenvalues(matlist, expo)}
\end{itemize}

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 \(\xi\) in the splitting field, which is of order \(expo\). \texttt{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.

Finally, there are two operations that 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 \(p\).

\begin{itemize}
\item \texttt{InverseMatMod(mat, obj)}
\end{itemize}

For a square matrix \(mat\), \texttt{InverseMatMod} returns a matrix \(inv\) such that \(inv \ast mat\) is congruent to the identity matrix modulo \(obj\), if such a matrix exists, and \texttt{fail} otherwise.

\begin{verbatim}
gap> mat:= [[ 1, 2 ], [ 3, 4 ]]; inv:= InverseMatMod( mat, 5 );
[ [ 3, 1 ], [ 4, 2 ] ]
\end{verbatim}

\begin{itemize}
\item \texttt{NullspaceModQ(E, q)}
\end{itemize}

\(E\) must be a matrix of integers and \(q\) a prime power. Then \texttt{NullspaceModQ} returns the set of all vectors of integers modulo \(q\), which solve the homogeneous equation system given by \(E\) modulo \(q\).

\begin{verbatim}
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 ] ]
\end{verbatim}

\section*{24.14 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. \texttt{GAP} uses the representation \texttt{IsBlockMatrixRep} to store block matrices.

\begin{itemize}
\item \texttt{AsBlockMatrix(m, nrb, ncb)}
\end{itemize}

returns a block matrix with \(nrb\) row blocks and \(ncb\) column blocks which is equal to the ordinary matrix \(m\).

\begin{itemize}
\item \texttt{BlockMatrix(blocks, nrb, ncb)}
\item \texttt{BlockMatrix(blocks, nrb, ncb, rpb, cpb, zero)}
\end{itemize}

\texttt{BlockMatrix} returns an immutable matrix in the sparse representation \texttt{IsBlockMatrixRep}. The nonzero blocks are described by the list \texttt{blocks} of triples, the 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 24.1.2), and also the block matrix is an ordinary matrix.

```latex
\textbf{3} \textgreater \textbf{MatrixByBlockMatrix( blockmat )} \hspace{1cm} \text{A}
```
returns a plain ordinary matrix that is equal to the block matrix \( \text{blockmat} \).
25  Integral matrices and lattices

25.1 Normal Forms over the Integers

DiagonalizeIntMatNormDriven(mat)

DiagonalizeIntMatNormDriven diagonalizes the integer matrix mat.
It tries to keep the entries small through careful selection of pivots.
First it selects a nonzero entry for which the product of row and column norm is minimal (this need not be the entry with minimal absolute value). Then it brings this pivot to the upper left corner and makes it positive.
Next it subtracts multiples of the first row from the other rows, so that the new entries in the first column have absolute value at most pivot/2. Likewise it subtracts multiples of the 1st column from the other columns.
If afterwards not all new entries in the first column and row are zero, then it selects a new pivot from those entries (again driven by product of norms) and reduces the first column and row again.
If finally all offdiagonal entries in the first column and row are zero, then it starts all over again with the submatrix mat[2..][2..].
The original idea is explained in [HM97].

gap> m:=[[14,20],[6,9]];
gap> DiagonalizeIntMatNormDriven(m);
gap> m;
[ [ 2, 0 ], [ 0, 3 ] ]

SNFNormDriven(mat[, trans])
SNFChouCollins(mat[, trans])
SNFLLLDriven(mat[, trans])

These operations have been superceded for most purposes by NormalFormIntMat (see 25.1.9) which should in most cases be faster than any of them, and produce smaller transforming matrix entries.
These operations compute the Smith normal form of a matrix with integer entries, using the strategy specified in the name. If no optional argument trans is given mat must be a mutable matrix which will be changed by the algorithm.
If the optional integer argument trans is given, it determines which transformation matrices will be computed.
It is interpreted binary as:
1 Row transformations.
2 Inverse row transformations.
4 Column transformations.
8 Inverse column transformations.
The operation then returns a record with the component normal containing the computed normal form and optional components rowtrans, rowinverse, coltrans, and invcoltrans which hold the computed transformation matrices. Note, if trans is given the operation does not change mat.

This functionality is still to be fully implemented for SNF with transforms. However, NormalFormIntMat performs this calculation.

3 ▶ SNFofREF( mat )

Computes the Smith Normal Form of an integer matrix in row echelon (RE) form. Caveat – No testing is done to ensure that mat is in RE form.

4 ▶ HNFNormDriven( mat[, trans[, reduction]] )
▶ HNFChouCollins( mat[, trans[, reduction]] )
▶ HNFLLLDriven( mat[, trans[, reduction]] )

These operations have been superceded for most purposes by NormalFormIntMat (see 25.1.9) which should in most cases be faster than any of them, and produce smaller transforming matrix entries.

These operations compute the Hermite normal form of a matrix with integer entries, using the strategy specified in the name. If no optional argument trans is given mat must be a mutable matrix which will be changed by the algorithm.

If the optional integer argument trans is given, it determines which transformation matrices will be computed. It is interpreted binary as for the Smith normal form (see 25.1.2) but note that only row operations are performed. The function then returns a record with components as specified for the Smith normal form.

If the further optional argument reduction (a rational in the range [0..1]) is given, it specifies which representatives are used for entries modulo c when cleaning column entries to the top. Off-diagonal entries are reduced to the range \( \lfloor c(r-1) \rfloor \ldots \lfloor cr \rfloor \), where r is the value of reduction. If reduction is not given, a value of 1 is assumed. Note, if trans is given the operation does not change mat.

5 ▶ TriangulizeIntegerMat( mat[, trans] )

Computes an upper triangular form of a matrix with integer entries. If no optional argument trans is given mat must be a mutable matrix which will be changed by the algorithm.

If the optional integer argument trans is given, it determines which transformation matrices will be computed. It is interpreted binary as for the Smith normal form (see 25.1.2) but note that only row operations are performed. The function then returns a record with components as specified for the Smith normal form. Note, if trans is given the operation does not change mat.
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 $PAQ = S$.

These operations compute the Smith normal form of a matrix $mat$ with integer entries. The operations will try to select a suitable strategy. The first operation returns a new immutable matrix in the Smith normal form. The other operations also compute matrices for the row and column transformations or inverses thereof respectively. They return a record with the component $normal$ containing the computed normal form and optional components $rowtrans$ and $coltrans$, or $invrowtrans$ and $invcoltrans$ which hold the computed transformation matrices.

```gap
gap> m:=[[14,20],[6,9]];
[ [ 14, 20 ], [ 6, 9 ] ]
gap> SmithNormalFormIntegerMat(m);
[ [ 1, 0 ], [ 0, 6 ] ]
```

The Hermite Normal Form, $H$ 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 $QA = H$.

These operations compute the Hermite normal form of a matrix $mat$ with integer entries. The operations will try to select a suitable strategy. The first operation returns a immutable matrix which is the Hermite normal form of $mat$.

The other two operations also compute matrices for the row transformations or inverses respectively. They return a record with the component $normal$ containing the computed normal form and optional components $rowtrans$ or 'invrowtrans' which hold the computed transformation matrix.

If the optional argument $reduction$ (a rational in the range $[0..1]$) is given, it specifies which representatives are used for entries modulo $c$ when cleaning column entries to the top. Off-diagonal entries are reduced to the range $\lfloor c(r-1)\rfloor \ldots \lfloor cr \rfloor$ where $r$ is the value of $reduction$. If $reduction$ is not given, a value of 1 is assumed.

```gap
gap> m;
[ [ 14, 20 ], [ 6, 9 ] ]
gap> HermiteNormalFormIntegerMat(m);
[ [ 2, 2 ], [ 0, 3 ] ]
gap> HermiteNormalFormIntegerMatTransforms(m);
rec( normal := [ [ 2, 2 ], [ 0, 3 ] ], rowtrans := [ [ 1, -2 ], [ -3, 7 ] ] )
```

The first operation computes a row equivalent upper triangular form of a matrix $mat$ with integer entries. It returns an immutable matrix in upper triangular form.

The other two operations also compute matrices for the row transformations or inverses respectively. They return a record with the component $normal$ containing the computed normal form and optional components $rowtrans$ or $invrowtrans$ which hold the computed transformation matrix.
NormalFormIntMat(mat, options)

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.

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 \( QAP = S \). The routines used are based on work by Arne Storjohann and were implemented in GAP 4 by A. Storjohann and R. Wainwright.

Note option is a value ranging from 0 - 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.

```
gap> m:=[[14,20],[6,9]];;
gap> NormalFormIntMat(m,0); # Triangular, no transforms
rec( normal := [ [ 2, 2 ], [ 0, 3 ] ], rank := 2, signdet := 1 )

gap> NormalFormIntMat(m,6); # Hermite Normal Form with row transforms
rec( normal := [ [ 2, 2 ], [ 0, 3 ] ], rank := 2, signdet := 1,
     rowtrans := [ [ 1, -2 ], [ -3, 7 ] ] )

gap> NormalFormIntMat(m,13); # Smith Normal Form with both transforming matrices
rec( normal := [ [ 1, 0 ], [ 0, 6 ] ], rank := 2, signdet := 1,
     rowtrans := [ [ -11, 25 ], [ -15, 34 ] ],
     coltrans := [ [ 1, -5 ], [ 1, -4 ] ] )
gap> last.rowtrans*m*last.coltrans;
[ [ 1, 0 ], [ 0, 6 ] ]
```

25.2 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 \([\frac{-p-1}{2}, \frac{p-1}{2}]\). 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

\[
x_i = (b_iA') \mod p, \quad b_{i+1} = \frac{1}{p}(b_i - x_iA), \quad i = 0, 1, 2, \ldots.
\]

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 24.13.5).

1. **Decomposition**($A$, $B$, $depth$)  
   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 $x \ast A = B[i]$, 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 $\sum_{k=0}^{depth} a_k p^k$, with all $a_k$ vectors of integers with entries bounded by $\pm p^{\frac{2}{3}}$. The prime $p$ is 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 $x \ast A = B[i]$ is a solution, we have $X[i] = x$, otherwise $X[i] = fail$.

   **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 $x \ast A = B[i]$, 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.

2. **LinearIndependentColumns**($mat$)  
   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 $mat$.

3. **PadicCoefficients**($A$, $Amodpinv$, $b$, $prime$, $depth$)  
   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 $b_{l+1} = 0$.

4. **IntegralizedMat**($A$)  
   **IntegralizedMat** returns for a matrix $A$ of cyclotomics a record $inmat$ with components $mat$ and $inforec$.

   Each family of algebraic conjugate columns of $A$ is encoded in a set of columns of the rational matrix $inmat.mat$ by replacing cyclotomics in $A$ by their coefficients w.r.t. an integral basis. $inmat.inforec$ 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 $A$.

5. **DecompositionInt**($A$, $B$, $depth$)  
   **DecompositionInt** does the same as **Decomposition** (see 25.2.1), except that $A$ and $B$ must be integral matrices, and $depth$ must be a nonnegative integer.
25.3 Lattice Reduction

LLLReducedBasis\(( [\ L, ]vectors[, y][, "linearcomb"][, lllout] )\)

provides an implementation of the LLL algorithm by Lenstra, Lenstra and Lovász (see [LLL82], [Poh87]). The implementation follows the description on pages 94f. in [Coh93].

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 \( \frac{1}{4} \) and 1 (the default value is \( \frac{3}{4} \)).

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 (see 25.3.2) computes an LLL reduced Gram matrix.

LLLReducedGramMat\(( \ G \ )\)

provides an implementation of the LLL algorithm by Lenstra, Lenstra and Lovász (see [LLL82], [Poh87]). The implementation follows the description on pages 94f. in [Coh93].

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 \(T \cdot G \cdot T^\top\) is the remainder component of the result.

The optional argument \(y\) denotes the “sensitivity” of the algorithm, it must be a rational number between \(\frac{1}{4}\) and 1; the default value is \(y = \frac{3}{4}\).

The function LLLReducedBasis (see 25.3.1) computes an LLL reduced basis.

```gap
gap> g:= [ [ 4, 6, 5, 2, 2 ], [ 5, 7, 11, 2, 0 ], [ 2, 4, 2, 8, 4 ], [ 2, 4, 0, 4, 8 ] ];;
gap> LLLReducedGramMat( g );; Display( last );
rec(
  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 ] ],
  relations := [ ],
  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 ] ],
  mue := [ [ ], [ 1/2 ], [ 1/4, -1/8 ], [ 1/2, 1/4, -2/25 ],
        [ -1/4, 1/8, 37/75, 8/21 ] ],
  B := [ 4, 4, 75/16, 168/25, 32/7 ] )
```

### 25.4 Orthogonal Embeddings

\texttt{OrthogonalEmbeddings( gram, "positive"}[, maxdim] ) \texttt{F}

computes all possible orthogonal embeddings of a lattice given by its Gram matrix \(\text{gram}\), which must be a regular matrix. In other words, all solutions \(X\) of the problem

\[ X^\top \cdot X = \text{gram} \]

are calculated (see [Ple90]). Usually there are many solutions \(X\) but all their rows are chosen from a small set of vectors, so \texttt{OrthogonalEmbeddings} returns the solutions in an encoded form, namely as a record with components

- \texttt{vectors}
  - the list \(L = [x_1, x_2, \ldots, x_n]\) of vectors that may be rows of a solution; these are exactly those vectors that fulfill the condition \(x_i \cdot \text{gram}^{-1} \cdot x_i^\top \leq 1\) (see 25.4.2), and we have \(\text{gram} = \sum_{i=1}^{n} x_i^\top \cdot x_i\),

- \texttt{norms}
  - the list of values \(x_i \cdot \text{gram}^{-1} \cdot x_i^\top\), and

- \texttt{solutions}
  - a list \(S\) of lists; the \(i\)-th solution matrix is \(L \cdot S[i]\), so the dimension of the \(i\)-th solution is the length of \(S[i]\).

The optional argument "positive" will cause \texttt{OrthogonalEmbeddings} to compute only vectors \(x_i\) with nonnegative entries. In the context of characters this is allowed (and useful) if \(\text{gram}\) is the matrix of scalar products of ordinary characters.

When \texttt{OrthogonalEmbeddings} is called with the optional argument \(\text{maxdim}\) (a positive integer), only solutions up to dimension \(\text{maxdim}\) are computed; this will accelerate the algorithm in some cases.
Section 4. Orthogonal Embeddings

gap> b := [ [ 3, -1, -1 ], [ -1, 3, -1 ], [ -1, -1, 3 ] ];

gap> c := OrthogonalEmbeddings( b ); Display( c ); ;
rec(
  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 ] ],
  norms := [ 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, 9, 9 ], [ 4, 5, 6, 7, 8, 9 ] ]
)

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 (see 70.10.7) may be able to compute irreducibles, see 70.10.

2। ShortestVectors( $G$, $m$, "positive" ) F

Let $G$ be a regular matrix of a symmetric bilinear form, and $m$ a nonnegative integer. ShortestVectors
computes the vectors $x$ that satisfy $x \cdot G \cdot x^t \leq m$, 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.

gap> g := [ [ 2, 1, 1 ], [ 1, 2, 1 ], [ 1, 1, 2 ] ];

gap> ShortestVectors(g,4);; Display( last );
rec(
  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 ] ],
  norms := [ 4, 2, 2, 4, 2, 4, 2, 2, 2 ] )
26 Strings and Characters

- `IsChar( obj )`
- `IsCharCollection( obj )`

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 '.

```gap
gap> x := 'a'; IsChar( x );
'a'
true

gap> '*';
'*'
```

- `IsString( obj )`

A string is a dense list (see 21.1.1, 21.1.2) of characters (see 26); thus strings are always homogeneous (see 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 26.2.1 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 26.1.

```gap
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' ]); IsString([ '1', '2', 3 ]); 
false # strings must be dense
false # strings must only contain characters

gap> s := "\007";
"\007"
gap> Print(s); # rings bell in many terminals
gap>

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 28.5), ask for the length, concatenate strings (see 21.20.1), form substrings etc. You can even assign to a mutable list. 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.

    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"

If a string is displayed by View, for example as result of an evaluation (see 6.1), or by ViewObj and PrintObj, 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 objects; it is only printed like a string if it was input as empty string or converted to a string with 26.2.2.) The difference between ViewObj and PrintObj is that the latter prints all non-printable and non-ASCII characters in three digit octal notation, while ViewObj sends all printable characters to the output stream. The output of PrintObj can be read back into GAP.

Strings behave differently from other GAP objects with respect to Print, PrintTo, or AppendTo. 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.

    gap> s4:= "abc"\"def\ngghi";;
    gap> View( s4 ); Print("\n" );
"abc\"def\ngghi"
    gap> ViewObj( s4 ); Print("\n" );
"abc\"def\ngghi"
    gap> PrintObj( s4 ); Print("\n" );
"abc\"def\ngghi"
    gap> Print( s4 ); Print("\n" );
abc"def
gghi
    gap> s := "German uses strange characters: \344\366\374\337\n";
"German uses strange characters: \n"
    gap> PrintObj(s);
"German uses strange characters: \344\366\374\337\n"gap>
gap> Print(s);
German uses strange characters:

Note that only those line breaks are printed by Print that are contained in the string (\n characters, see 26.1), as is shown in the example below.

gap> s1;
"Hello world."
gap> Print( s1 );
Hello world.
next line
gap>

26.1 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 \64+Y8+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, PrintTo, or AppendTo send the characters directly to the output stream.
gap> "This is one line.\nThis is another line.\n";
"This is one line.\nThis is another line.\n"

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.

    gap> "This is a very long string that does not fit on a line \n    gap> and is therefore continued on the next line.";
    "This is a very long string that does not fit on a line and is therefore conti\n    nued on the next line."

Note that the output is also continued, but at a different place that is determined by the value of SizeScreen (see 6.11.1).

### 26.2 Internally Represented Strings

1. **IsStringRep**  
   
   IsStringRep is a special (internal) representation of dense lists of characters. Dense lists of characters can be converted into this representation using ConvertToStringRep. Note that calling IsString does **not** change the representation.

2. **ConvertToStringRep**  
   
   If obj is a dense internally represented list of characters then ConvertToStringRep changes the representation to IsStringRep. This is useful in particular for converting the empty list [], which usually is in IsPlistRep, to IsStringRep. If obj is not a string then ConvertToStringRep signals an error.

3. **IsEmptyString**  
   
   IsEmptyString returns true if str is the empty string in the representation IsStringRep, 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.

    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
Each character lies in the family \texttt{CharFamily}, each nonempty string lies in the collections family of this family. Note the subtle differences between the empty list \texttt{[]} and the empty string \texttt{""} when both are printed.

### 26.3 Recognizing Characters

1. \texttt{IsDigitChar( }c\texttt{ )} checks whether the character \texttt{c} is a digit, i.e., occurs in the string "0123456789".

2. \texttt{IsLowerAlphaChar( }c\texttt{ )} checks whether the character \texttt{c} is a lowercase alphabet letter, i.e., occurs in the string "abcdefghijklmnopqrstuvwxyz".

3. \texttt{IsUpperAlphaChar( }c\texttt{ )} checks whether the character \texttt{c} is an uppercase alphabet letter, i.e., occurs in the string "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

4. \texttt{IsAlphaChar( }c\texttt{ )} checks whether the character \texttt{c} is either a lowercase or an uppercase alphabet letter.

### 26.4 Comparisons of Strings

1. \texttt{string1} = \texttt{string2}

2. \texttt{string1} <> \texttt{string2}

The equality operator \texttt{=} returns \texttt{true} if the two strings \texttt{string1} and \texttt{string2} are equal and \texttt{false} otherwise. The inequality operator \texttt{<>} returns \texttt{true} if the two strings \texttt{string1} and \texttt{string2} are not equal and \texttt{false} otherwise.

- \texttt{gap> "Hello world.\n" = "Hello world.\n";} \texttt{true}
- \texttt{gap> "Hello World.\n" = "Hello world.\n";} \texttt{false} # string comparison is case sensitive
- \texttt{gap> "Hello world." = "Hello world.\n";} \texttt{false} # the first string has no <newline>
- \texttt{gap> "Goodbye world.\n" = "Hello world.\n";} \texttt{false}
- \texttt{gap> [ 'a', 'b' ] = "ab";} \texttt{true}

2. \texttt{string1} < \texttt{string2}

The ordering of strings is lexicographically according to the order implied by the underlying, system dependent, character set.
Strings can be compared via < with certain GAP objects that are not strings, see 4.11 for the details.

26.5 Operations to Produce or Manipulate Strings

1 ▶ String( obj )

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 may be safely applied to it.

```gap
gap> String(123); String([1,2,3]);
"123" 
"[ 1, 2, 3 ]"
```

2 ▶ HexStringInt( int )

returns a string which represents the integer int with hexa-decimal digits (using A-F as digits 10-15). The inverse translation can be achieved with 26.6.1.

3 ▶ StringPP( int )

returns a string representing the prime factor decomposition of the integer int.

```gap
gap> StringPP(40320);
"2^7*3^2*5*7"
```

4 ▶ WordAlp( alpha, nr )

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 ).

```gap
gap> List([0..5],i->WordAlp("abc",i));
[ "", "a", "b", "c", "aa", "ab" ]
```

5 ▶ LowercaseString( string )

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.

```gap
gap> LowercaseString("This Is UpperCase");
"this is uppercase"
```

6 ▶ SplitString( string, seps[, wspace] )

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 \textit{wspace} are treated as one white space character and they are ignored at the beginning and end of a string.

Both arguments \textit{seps} and \textit{wspace} can be single characters.

Each string in the resulting list of substring does not contain any characters in \textit{seps} or \textit{wspace}.

A character that occurs both in \textit{seps} and \textit{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 26.5.11.

\begin{verbatim}
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" ]
\end{verbatim}

\textbf{7} \textbf{\textit{ReplacedString( string, old, new )}} F

replaces occurrences of the string \textit{old} in \textit{string} by \textit{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.

\begin{verbatim}
gap> ReplacedString("abacab","a","zl");
"zlbzlczlb"
gap> ReplacedString("ababa", "aba","c");
"cba"
gap> ReplacedString("abacab", "a","ba");
"babbacbab"
\end{verbatim}

\textbf{8} \textbf{\textit{NormalizeWhitespace( string )}} F

This function changes the string \textit{string} in place. The characters (space), \texttt{\textbackslash n}, \texttt{\textbackslash r} and \texttt{\textbackslash t} are considered as white space. Leading and trailing white space characters in \textit{string} are removed. Sequences of white space characters between other characters are replaced by a single space character.

See 26.5.9 for a non-destructive version.

\begin{verbatim}
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"
\end{verbatim}

\textbf{9} \textbf{\textit{NormalizedWhitespace( str )}} F

This function returns a copy of string \textit{str} to which 26.5.8 was applied.

For the possibility to print GAP objects to strings, see 10.7.

\textbf{10} \textbf{\textit{JoinStringsWithSeparator( list[, sep] )}} F

joins \textit{list} (a list of strings) after interpolating \textit{sep} (or ";" if the second argument is omitted) between each adjacent pair of strings; \textit{sep} should be a string.

\textbf{Examples}
Section 6. Operations to Evaluate Strings

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> JoinStringsWithSeparator(["a", "b", "c", "d"], ",\n ");
"a,\n b,\n c,\n d"
gap> Print(" ", last, 
"");
a,
b,
c,
d

Recall, last is the last expression output by GAP.

11 ▶ Chomp( str ) F

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, for example.

gap> Chomp("The quick brown fox jumps over the lazy dog.\n");  "The quick brown fox jumps over the lazy dog."

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 (see 26.5.6), e.g.

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."

26.6 Operations to Evaluate Strings

1 ▶ Int( str ) A

1 ▶ Rat( str ) A

1 ▶ IntHexString( str ) F

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 10-15. An error occurs when a wrong character is found in the string. This function can be used (together with 26.5.2) for efficiently storing and reading large integers from respectively into GAP. Note that the translation between integers and their hexa-decimal representation costs linear computation time in terms of the number of digits, while translation from and into decimal representation needs substantial computations.

```gap
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"
```

2 ► **Ordinal( n )**

returns the ordinal of the integer n as a string.

```gap
gap> Ordinal(2);
"2nd"
gap> Ordinal(21);
"21st"
gap> Ordinal(33);
"33rd"
gap> Ordinal(-33);
"-33rd"
```

3 ► **EvalString( expr )**

passes expr (a string) through an input text stream so that GAP interprets it, and returns the result. The following trivial example demonstrates its use.

```gap
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** (see 10.7.1) and **ReadAsFunction** (see 10.3.2) together; see 10.3 for an example.
All calendar functions use the Gregorian calendar.

1. DaysInYear( year )
   returns the number of days in a year.

2. DaysInMonth( month, year )
   returns the number of days in month number month of year (and fail if month is integer not in valid range).

   \[
   \text{gap> DaysInYear(1998);}
   \]
   \[
   365
   \]
   \[
   \text{gap> DaysInMonth(3,1998);}
   \]
   \[
   31
   \]

3. DMYDay( day )
   converts a number of days, starting 1-Jan-1970 to a list [day, month, year] in Gregorian calendar counting.

4. DayDMY( dmy )
   returns the number of days from 01-Jan-1970 to the day given by dmy. dmy 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 countries or before 1917 in Russia.

5. WeekDay( date )
   returns the weekday of a day given by date. date can be a number of days since 1-Jan-1970 or a list [day, month, year].

6. StringDate( date )
   converts date to a readable string. date can be a number of days since 1-Jan-1970 or a list [day, month, year].

   \[
   \text{gap> DayDMY([1,1,1970]);DayDMY([2,1,1970]);}
   \]
   \[
   0
   \]
   \[
   1
   \]
   \[
   \text{gap> DMYDay(12345);}
   \]
   \[
   [ 20, 10, 2003 ]
   \]
   \[
   \text{gap> WeekDay([11,3,1998]);}
   \]
   \[
   \text{"Wed"}
   \]
   \[
   \text{gap> StringDate([11,3,1998]);}
   \]
   \[
   \text{"11-Mar-1998"}
   \]

7. HMSMSec( msec )
   converts a number msec of milliseconds into a list [hour, min, sec, milli].

8. SecHMSM( hmsm )
   is the reverse of HMSMSec.

9. StringTime( time )
   converts time (given as a number of milliseconds or a list [hour, min, sec, milli]) to a readable string.
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"

returns the number of seconds from 01-Jan-1970, 00:00:00, to the time given by \emph{DMYhms}. \emph{DMYhms} must be a list of the form \emph{[day, month, year, hour, minute, second]}. The remarks on the Gregorian calendar in the section on 26.7.4 apply here as well. The last three arguments must lie in the appropriate ranges.

This is the inverse function to 26.7.10.

\begin{verbatim}
gap> SecondsDMYhms([9, 9, 2001, 1, 46, 40]);
1000000000
gap> DMYhmsSeconds(-1000000000);
[ 24, 4, 1938, 22, 13, 20 ]
\end{verbatim}
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().

```gap
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 function to illustrate the hierarchy of the record components.

```gap
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.

```gap
gap> IsRecord( rec( a := 1, b := 2 ) );
true
gap> IsRecord( IsRecord );
false
```

returns a list of strings corresponding to the names of the record components of the record rec.

```gap
gap> r := rec( a := 1, b := 2 );
gap> RecNames( r );
[ "a", "b" ]
```

Note that you cannot use the string result in the ordinary way to access or change a record component. You must use the rec.(name) construct (see 27.1 and 27.2).
27.1 Accessing Record Elements

1 ▶ rec.name

The above construct evaluates to the value of the record component with the name name in the record rec. Note that the name is not evaluated, i.e. it is taken literal.

   gap> r := rec( a := 1, b := 2 );;
   gap> r.a;
   1
   gap> r.b;
   2

2 ▶ rec.(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 rec whose name is, as a string, equal to name.

   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 )

27.2 Record Assignment

1 ▶ rec.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 rec. That means that accessing the element with name name of the record rec will return obj after this assignment. If the record rec has no component with the name name, the record is automatically extended to make room for the new component.

   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 can be used to test if a record has a component with a certain name, the function Unbind (see 4.8.1) can be used to remove a component with a certain name again.
Section 3. Identical Records

```gap
gap> IsBound(r.a);
true
gap> IsBound(r.d);
false
gap> Unbind(r.b);
```

```gap
gap> Display( r );
rec(
a := 10,
c := 3
)
```

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 rec whose name is, as a string, equal to name.

### 27.3 Identical Records

With the record assignment (see 27.2) 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).

```gap
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.

```gap
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 := record means in this interpretation that var is a name for the object record. 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.

```gap
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 ).

```gap
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.
r1 := rec( a := 1 );
r2 := rec( a := 1 );

Also in the following example, no records in the list l are identical.

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.

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 record

r1 := rec( a := 1 );
f := function ( r ) return r; end;
r2 := f( r1 );

The functions StructuralCopy and ShallowCopy (see 12.7.2 and 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 and ShallowCopy is that in the case of ShallowCopy the corresponding components of the new and the old records will be identical, whereas in the case of StructuralCopy they will only be equal. So in the following example r1 and r2 are not identical records.

r1 := rec( a := 1 );
r2 := Copy( 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.

r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;

27.4 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.
Section 5. IsBound and Unbind for Records

```gap
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
```

To compare records we imagine that the components of both records are sorted according to their names. 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`.

```gap
gap> rec( a := 1, b := 2 ) < rec( b := 2, a := 1 );
false # they are equal
gap> rec( a := 1, b := 2 ) < rec( a := 2, b := 0 );
true # the 'a' elements are compared first and 1 is less than 2
gap> rec( a := 1 ) < rec( a := 1, b := 2 );
true # unbound is less than 2
gap> rec( a := 1 ) < rec( a := 0, b := 2 );
false # the 'a'-s are compared first and unbound is less than 2
```

27.5 IsBound and Unbind for Records

```gap
'IsBound( rec.name )' 0
```

`IsBound` returns `true` if the record `rec` has a component with the name `name` (which must be an identifier) and `false` otherwise. `rec` must evaluate to a record, otherwise an error is signalled.

```gap
gap> r := rec( a := 1, b := 2 );
gap> IsBound( r.a );
true
gap> IsBound( r.c );
false
```

```gap
'Unbind( rec.name )' 0
```

`Unbind` deletes the component with the name `name` in the record `rec`. That is, after execution of `Unbind`, `rec` no longer has a record component with this name. Note that it is not an error to unbind a nonexisting record component. `rec` must evaluate to a record, otherwise an error is signalled.

```gap
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` and `Unbind` are special in that they do not evaluate their argument, otherwise `IsBound` 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.
27.6 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 list of all RNams that are used and provides functions to translate RNams to strings that give the component names and vice versa.

\begin{verbatim}
1 ▶ NameRNam(nr)
returns a string representing the component name corresponding to the RNam nr.

2 ▶ RNamObj(str)
   ▶ RNamObj(int)
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.

   gap> NameRNam(798);
   "BravaisSupergroups"
   gap> RNamObj("blubberflutsch");
   2075
   gap> NameRNam(last);
   "blubberflutsch"
\end{verbatim}

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 runs of GAP.

The following 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.

\begin{verbatim}
3 ▶ \.(obj,rnam)
   ▶ IsBound\.(obj,rnam)
   ▶ \.(obj,rnam)
   ▶ Unbind\.(obj,rnam)

These 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 declared as IsPosInt.
\end{verbatim}
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 homogeneous.

Basic operations for collections are Size (see 28.3.6) and Enumerator (see 28.2.2); for finite collections, Enumerator admits to delegate the other operations for collections (see 28.3 and 28.4) 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.

1 ▶ IsCollection( obj )

tests whether an object is a collection.

Some of the functions for lists and collections have been described in the chapter about lists, mainly in Section 21.20. In this chapter, we describe those functions for which the “collection aspect” seems to be more important than the “list aspect”. As in Chapter 21, an argument that is a list will be denoted by list, and an argument that is a collection will be denoted by C.

28.1 Collection Families

1 ▶ CollectionsFamily( Fam )

For a family Fam, CollectionsFamily returns the family of all collections that consist of elements in Fam. Note that families (see 13.1) are used to describe relations between objects. Important such relations are that between an element elm and each collection of elements that lie in the same family as elm, 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 ).

2 ▶ IsCollectionFamily( Fam )

is true if Fam is a family of collections, and false otherwise.

3 ▶ ElementsFamily( Fam )

returns the family from which the collections family Fam was created by CollectionsFamily. The way a collections family is created, it always has its elements family stored. If Fam is not a collections family (see 28.1.2) then an error is signalled.

```
gap> fam:= FamilyObj( (1,2) );;
gap> collfam:= CollectionsFamily( fam );;
gap> fam = collfam; fam = ElementsFamily( collfam );
false
true

gap> collfam = FamilyObj( [ (1,2,3) ] ); collfam = FamilyObj( Group( () ) );
true
true

gap> collfam = CollectionsFamily( collfam );
```
CategoryCollections( \text{filter} ) \]

Let \text{filter} be a filter that is true for all elements of a family \text{Fam}, by construction of \text{Fam}. Then \text{CategoryCollections} returns a category that is true for all elements in \text{CollectionsFamily}( \text{Fam} ).

For example, the construction of \text{PermutationsFamily} guarantees that each of its elements lies in the filter \text{IsPerm}, and each collection of permutations lies in the category \text{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.

### 28.2 Lists and Collections

\text{IsListOrCollection( obj )} \]

Several functions are defined for both lists and collections, for example \text{Intersection (see 28.4.2)}, \text{Iterator (see 28.7.1)}, and \text{Random (see 28.6.1)}. \text{IsListOrCollection} is a supercategory of \text{IsList} and \text{IsCollection} (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.

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 21.1.5).

\text{Enumerator( C )} \]

\text{Enumerator( list )} \]

\text{Enumerator} returns an immutable list \text{enum}. If the argument is a list \text{list} (which may contain holes), then \text{Length( enum )} is \text{Length( list )}, and \text{enum} contains the elements (and holes) of \text{list} in the same order. If the argument is a collection \text{C} that is not a list, then \text{Length( enum )} is the number of different elements of \text{C}, and \text{enum} contains the different elements of \text{C} in an unspecified order, which may change for repeated calls of \text{Enumerator}. \text{enum}[\text{pos}] may not execute in constant time (see 21.1.5), and the size of \text{enum} in memory is as small as is feasible.

For lists \text{list}, the default method is \text{Immutable}. For collections \text{C} that are not lists, there is no default method.

\text{EnumeratorSorted( C )} \]

\text{EnumeratorSorted( list )} \]

\text{EnumeratorSorted} returns an immutable list \text{enum}. The argument must be a collection \text{C} or a list \text{list} which may contain holes but whose elements lie in the same family (see 13.1). \text{Length( enum )} is the number of different elements of \text{C} resp. \text{list}, and \text{enum} contains the different elements in sorted order, w.r.t. <. \text{enum}[\text{pos}] may not execute in constant time (see 21.1.5), and the size of \text{enum} in memory is as small as is feasible.

\text{gap> Enumerator( [ 1, 3,, 2 ] );}
[ 1, 3, 2 ]
\text{gap> enum:= Enumerator( Rationals );; elm:= enum[ 10^6 ];}
-69/907
\text{gap> Position( enum, elm );}
1000000
\text{gap> IsMutable( enum ); IsSortedList( enum );}
false
\text{false}
\text{gap> IsConstantTimeAccessList( enum );}
false
gap> EnumeratorSorted( [ 1, 3,, 2 ] );
[ 1, 2, 3 ]

\[\text{List}( C )\]
\[\text{List}( \text{list} )\]

This function is described in 21.20.16, together with the probably more frequently used version which takes a function as second argument and returns the list of function values of the list elements.

gap> l:= List( Group( (1,2,3) ) );
[ (), (1,3,2), (1,2,3) ]
gap> IsMutable( l ); IsSortedList( l ); IsConstantTimeAccessList( l );
true
false
true

gap> SortedList( C ) \quad O
\[\text{SortedList}( \text{list} )\]

SortedList returns a new mutable and dense list new. The argument must be a collection C or a list list which may contain holes but whose elements lie in the same family (see 13.1). Length( new ) is the number of elements of C resp. list, and new contains the elements in sorted order, w.r.t. \(<=\). new[pos] executes in constant time (see 21.1.5), and the size of new in memory is proportional to its length.

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 ]

\[\text{SSortedList}( C )\]
\[\text{SSortedList}( \text{list} )\]
\[\text{Set}( C )\]

SSortedList ("strictly sorted list") returns a new dense, mutable, and duplicate free list new. The argument must be a collection C or a list list which may contain holes but whose elements lie in the same family (see 13.1). Length( new ) is the number of different elements of C resp. list, and new contains the different elements in strictly sorted order, w.r.t. \(<\). new[pos] executes in constant time (see 21.1.5), and the size of new in memory is proportional to its length.

Set is simply a synonym for SSortedList.

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 ]
AsList returns an immutable list \( \text{imm} \). If the argument is a list \( \text{list} \) (which may contain holes), then \( \text{Length}( \text{imm} ) \) is \( \text{Length}( \text{list} ) \), and \( \text{imm} \) contains the elements (and holes) of \( \text{list} \) in the same order. If the argument is a collection \( C \) that is not a list, then \( \text{Length}( \text{imm} ) \) is the number of different elements of \( C \), and \( \text{imm} \) contains the different elements of \( C \) in an unspecified order, which may change for repeated calls of \( \text{AsList} \). \( \text{imm}[\text{pos}] \) executes in constant time (see 21.1.5), and the size of \( \text{imm} \) 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 \( \text{AsSortedList} \).

\begin{verbatim}
gap> l := AsList( [ 1, 3, 3,, 2 ] );
[ 1, 3, 3,, 2 ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
false  true  true

gap> AsList( Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
\end{verbatim}

AsSortedList returns a dense and immutable list \( \text{imm} \). The argument must be a collection \( C \) or a list \( \text{list} \) which may contain holes but whose elements lie in the same family (see 13.1). \( \text{Length}( \text{imm} ) \) is the number of elements of \( C \) resp. \( \text{list} \), and \( \text{imm} \) contains the elements in sorted order, w.r.t. \(<=\). \( \text{imm}[\text{pos}] \) executes in constant time (see 21.1.5), and the size of \( \text{imm} \) in memory is proportional to its length.

The only difference to the operation \( \text{SortedList} \) (see 28.2.4) is that \( \text{AsSortedList} \) returns an immutable list.

\begin{verbatim}
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
\end{verbatim}

AsSSortedList ("as strictly sorted list") returns a dense, immutable, and duplicate free list \( \text{imm} \). The argument must be a collection \( C \) or a list \( \text{list} \) which may contain holes but whose elements lie in the same family (see 13.1). \( \text{Length}( \text{imm} ) \) is the number of different elements of \( C \) resp. \( \text{list} \), and \( \text{imm} \) contains the different elements in strictly sorted order, w.r.t. \(<\). \( \text{imm}[\text{pos}] \) executes in constant time (see 21.1.5), and the size of \( \text{imm} \) 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 \( \text{AsList} \) instead if you do not require the result to be sorted.

The only difference to the operation \( \text{SSortedList} \) (see 28.2.5) is that \( \text{AsSSortedList} \) 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.

\[
gap> l := \text{AsSSortedList}(1);
\begin{array}{c}
\text{[ 1, 2, 3 ]}
\end{array}
\]
\[
gap> \text{IsEmpty( l );} \hspace{1em} \text{IsSSortedList( l );} \hspace{1em} \text{IsConstantTimeAccessList( l );}
\begin{array}{c}
\text{false}
\end{array}
\]
\[
\text{true}
\]
\[
\text{true}
\]
\[
gap> \text{AsSSortedList( Group( (1,2,3), (1,2) );}
\begin{array}{c}
\text{[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]}
\end{array}
\]

Elements \( C \) does the same as AsSSortedList (see 28.2.8), 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 (see 28.2.6). If one is really interested in the strictly sorted list of elements in \( C \) then one should use AsSet or AsSSortedList instead.

### 28.3 Attributes and Properties for Collections

#### 1. IsEmpty( \( C \) )
IsEmpty returns true if the collection \( C \) resp. the list \( list \) is empty (that is it contains no elements), and false otherwise.

#### 2. IsFinite( \( C \) )
IsFinite returns true if the collection \( C \) is finite, and false otherwise.

The default method for IsFinite checks the size (see 28.3.6) of \( C \).

Methods for IsFinite may call Size, but methods for Size must not call IsFinite.

#### 3. IsTrivial( \( C \) )
IsTrivial returns true if the collection \( C \) consists of exactly one element.

#### 4. IsNonTrivial( \( C \) )
IsNonTrivial returns true if the collection \( C \) is empty or consists of at least two elements (see 28.3.3).
IsWholeFamily returns `true` if the collection `C` contains the whole family (see 13.1) of its elements.

```gap
gap> IsWholeFamily( Integers );
false
# all rationals and cyclotomics lie in the family
gap> IsWholeFamily( Integers mod 3 );
false
# all finite field elements in char. 3 lie in this family
gap> IsWholeFamily( Integers mod 4 );
true
gap> IsWholeFamily( FreeGroup( 2 ) );
true
```

Size returns the size of the collection `C`, which is either an integer or `infinity`. The argument may also be a list `list`, in which case the result is the length of `list` (see 21.17.5).

The default method for `Size` checks the length of an enumerator of `C`.

Methods for `IsFinite` may call `Size`, but methods for `Size` must not call `IsFinite`.

```gap
gap> Size( [1,2,3] ); Size( Group( () ) ); Size( Integers );
3
1
infinity
```

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` (see 28.6.1) is that `Representative` is free to choose a value that is cheap to compute, while `Random` 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 30. 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.

Returns the smallest element in the collection `C`, w.r.t. the ordering `<`. While the operation defaults to comparing all elements, better methods are installed for some collections.

```gap
gap> Representative( Rationals );
1
gap> Representative( [ -1, -2 .. -100 ] );
-1
gap> RepresentativeSmallest( [ -1, -2 .. -100 ] );
-100
```
28.4 Operations for Collections

1 ▶ IsSubset( C1, C2 )

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).

    gap> IsSubset( Rationals, Integers );
    true
    gap> IsSubset( Integers, [ 1, 2, 3 ] );
    true
    gap> IsSubset( Group( (1,2,3,4) ), [ (1,2,3) ] );
    false

2 ▶ Intersection( C1, C2 ... )
    ▶ Intersection( list )
    ▶ Intersection2( C1, C2 )

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 (see 28.2.5) to it first.

The result of Intersection is the set of elements that lie in every of the collections 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.

    gap> Intersection( CyclotomicField(9), CyclotomicField(12) );
    CF(3)  # 'CF' is a shorthand for 'CyclotomicField'
    # this is one of the rare cases where the intersection
    # of two infinite domains works
    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) ] );
    [ (1,3)(4,6) ]  # although the result is mathematically a
    # group it is returned as a proper set
    # because the second argument was not
    # regarded as a group
    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] ] );
    [ 4 ]  # or one list of lists or collections
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` (see 28.2.5) to it first.

The result of `Union` is the set of elements that lie in any of the collections `C1, C2, etc.`

Methods can be installed for the operation `Union2` that takes only two arguments. `Union` calls `Union2`.

```gap
gap> Union( [ (1,2,3), (1,2,3,4) ], Group( (1,2,3), (1,2) ) );
[ (1,2,3,4) ]
gap> Union( [ [1,2,4], [2,3,4], [1,3,4] ] );
[ 1, 2, 3, 4 ]  # or one list of lists or collections
gap> Union( [ ] );
[ ]
gap> Union( [1,2,3], Group( (1,2,3), (1,2) ) );
[1, 2, 3, 4]
```

4 ▶ `Difference( C1, C2 )`

`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` (see 28.2.5) 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.

```gap
gap> Difference( [ (1,2,3), (1,2,3,4) ], Group( (1,2,3), (1,2) ) );
[ (1,2,3,4) ]
gap> Difference( [ [1,2,3], [1,3,4] ] );
[ 1, 3, 4 ]
gap> Difference( Group( (1,2) ), Group( (1,2,3) ) );
[ (1,2,3) ]
gap> Difference( Group( (1,2,3), (1,2) ) );
[ ]
```

28.5 Membership Test for Collections

1 ▶ `obj in C`

The infix version of the command calls the operation `\in`, for which methods can be installed.

```gap
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
```
28.6 Random Elements

\textbf{Random} returns a (pseudo-)random element of the collection \(C\) respectively the list \(\text{list}\).

The distribution of elements returned by \textbf{Random} depends on the argument. For a list \(\text{list}\), all elements are equally likely. The same holds usually for finite collections \(C\) that are not lists. For infinite collections \(C\) 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 \textbf{PseudoRandom} should be used instead.

Note that \textbf{Random} is of course not an attribute.

\begin{verbatim}
gap> Random(Rationals); -4
gap> g:= Group( (1,2,3) );; Random( g ); Random( g ); () (1,2,3)
\end{verbatim}

\textbf{StateRandom} returns a GAP object that represents the current state of the random number generator used by \textbf{RandomList}.

By calling \textbf{RestoreStateRandom} with this object as argument, the random number is reset to this same state.

(The same result can be obtained by accessing the two global variables \texttt{R.N} and \texttt{R.X}.)

\begin{verbatim}
gap> seed:=StateRandom();
gap> List([1..10],i->Random(Integers)); [-1, -3, -1, 1, 2, 0, 1, 1, -1, 1]
gap> List([1..10],i->Random(Integers)); [ 2, -2, -1, -4, -2, 1, -1, 1, -2, -3]
gap> RestoreStateRandom(seed);
gap> List([1..10],i->Random(Integers)); [-1, -3, -1, 1, 2, 0, 1, 1, -1, 1]
\end{verbatim}

\textbf{PseudoRandom} returns a pseudo random element of the collection \(C\) respectively the list \(\text{list}\), which can be roughly described as follows. For a list \(\text{list}\), \textbf{PseudoRandom} returns the same as \textbf{Random}. For collections \(C\) that are not lists, the elements returned by \textbf{PseudoRandom} are not necessarily equally distributed, even for finite collections \(C\); the idea is that \textbf{Random} (see 28.6.1) returns elements according to a reasonable distribution, \textbf{PseudoRandom} returns elements that are cheap to compute but need not satisfy this strong condition, and \textbf{Representative} (see 28.3.7) returns arbitrary elements, probably the same element for each call.

The method used by GAP to obtain random elements may depend on the type object.
Many random methods in the library are eventually based on the function `RandomList`. As `RandomList` is restricted to lists of up to $2^{28}$ elements, this may create problems for very large collections. Also note that the method used by `RandomList` is intended to provide a fast algorithm rather than to produce high quality randomness for statistical purposes.

If you implement your own Random methods we recommend that they initialize their seed to a defined value when they are loaded to permit to reproduce calculations even if they involved random elements.

For a dense list `list` of up to $2^{28}$ elements, `RandomList` returns a (pseudo-)random element with equal distribution.

The algorithm used is an additive number generator (Algorithm A in section 3.2.2 of [Knu98] with lag 30). 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 `StatusRandom` for a description on how to reset the random number generator to a previous state.

### 28.7 Iterators

Iterators provide a possibility to loop over the elements of a (countable) collection `C` or a list `list`, 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 is a list `list` (which may contain holes), then `iter` iterates over the elements (but not the holes) of `list` in the same order (see 28.7.6 for details). If this is a collection `C` but not a list then `iter` iterates over the elements of `C` 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`, `NextIterator`, and `ShallowCopy`. In particular, it is only possible to access the next element of the iterator with `NextIterator` if there is one, and this can be checked with `IsDoneIterator` (see 28.7.5). For an iterator `iter`, `ShallowCopy( iter )` is a mutable iterator `new` that iterates over the remaining elements independent of `iter`; the results of `IsDoneIterator` for `iter` and `new` are equal, and if `iter` is mutable then also the results of `NextIterator` 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 `list` (see 28.7.6 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 59.5.6 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( list )`. For collections 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.
Section 7. Iterators

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 ]

IteratorSorted returns a mutable iterator. The argument must be a collection \( C \) or a list \( \text{list} \) 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 collections \( C \) that are not lists, the generic method is IteratorList( EnumeratorSorted( \( C \) ) ).

Every iterator lies in the category IsIterator.

If \( \text{iter} \) is an iterator for the list or collection \( C \) then IsDoneIterator( \( \text{iter} \) ) is true if all elements of \( C \) have been returned already by NextIterator( \( \text{iter} \) ), and false otherwise.

Let \( \text{iter} \) be a mutable iterator for the list or collection \( C \). If IsDoneIterator( \( \text{iter} \) ) is false then NextIterator is applicable to \( \text{iter} \), and the result is the next element of \( C \), according to the succession defined by \( \text{iter} \).

If IsDoneIterator( \( \text{iter} \) ) is true then it is not defined what happens if NextIterator is called for \( \text{iter} \); that is, it may happen that an error is signalled or that something meaningless is returned, or even that GAP crashes.

IteratorList returns a new iterator that allows iteration over the elements of the list \( \text{list} \) (which may have holes) in the same order.

If \( \text{list} \) is mutable then it is in principle possible to change \( \text{list} \) 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 \( \text{list} \) is enlarged then the iterator will iterate also over the new elements at the end of the changed list.

Note that changes of \( \text{list} \) will also affect all shallow copies of \( \text{list} \).

is a mutable iterator for the collection \( \text{[ elm ]} \) that consists of exactly one element \( elm \) (see 28.3.3).
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

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
In GAP an ordering is a relation defined on a family, which is reflexive, anti-symmetric and transitive.

1. IsOrdering( ord )

   returns true if and only if the object ord is an ordering.

2. OrderingsFamily( fam )

   for a family fam, returns the family of all orderings on elements of fam.

29.1 Building new orderings

1. OrderingByLessThanFunctionNC( fam, lt )

   In the first form, OrderingByLessThanFunctionNC returns the ordering on the elements of the elements of the family fam according to the LessThanFunction given by lt, where lt is a function that takes two arguments in fam and returns true or false.

   In the second form, 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 given by lt and with the properties from l set to true.

2. OrderingByLessThanOrEqualFunctionNC( fam, lteq )

   In the first form, OrderingByLessThanOrEqualFunctionNC returns the ordering on the elements of the family fam according to the LessThanOrEqualFunction given by lteq, where lteq is a function that takes two arguments in fam and returns true or false.

   In the second form, 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 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 true.

```gap
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;
genfunction( x, y ) ... end
gap> fam := FamilyObj(a);;
gap> ord := OrderingByLessThanFunctionNC(fam,lt);
Ordering
```
29.2 Properties and basic functionality

1. \( \text{IsWellFoundedOrdering}(ord) \) for an ordering \( ord \), returns \text{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.

2. \( \text{IsTotalOrdering}(ord) \) for an ordering \( ord \), returns \text{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.

3. \( \text{IsIncomparableUnder}(ord, el1, el2) \) for an ordering \( ord \) on the elements of the family of \( el1 \) and \( el2 \), returns \text{true} if \( el1 \neq el2 \) and \( \text{IsLessThanUnder}(ord,el1,el2), \text{IsLessThanUnder}(ord,el2,el1) \) are both false; and returns \text{false} otherwise.

4. \( \text{FamilyForOrdering}(ord) \) for an ordering \( ord \), returns the family of elements that the ordering \( ord \) compares.

5. \( \text{LessThanFunction}(ord) \) for an ordering \( ord \), returns a function \( f \) which takes two elements \( el1, el2 \) in the \( \text{FamilyForOrdering}(ord) \) and returns \text{true} if \( el1 \) is strictly less than \( el2 \) (with respect to \( ord \)) and returns \text{false} otherwise.

6. \( \text{LessThanOrEqualFunction}(ord) \) for an ordering \( ord \), returns a function that takes two elements \( el1, el2 \) in the \( \text{FamilyForOrdering}(ord) \) and returns \text{true} if \( el1 \) is less than or equal to \( el2 \) (with respect to \( ord \)) and returns \text{false} otherwise.

7. \( \text{IsLessThanUnder}(ord, el1, el2) \) for an ordering \( ord \) on the elements of the family of \( el1 \) and \( el2 \), returns \text{true} if \( el1 \) is (strictly) less than \( el2 \) with respect to \( ord \), and \text{false} otherwise.

8. \( \text{IsLessThanOrEqualUnder}(ord, el1, el2) \) for an ordering \( ord \) on the elements of the family of \( el1 \) and \( el2 \), returns \text{true} if \( el1 \) is less than or equal to \( el2 \) with respect to \( ord \), and \text{false} otherwise.

\begin{verbatim}
gap> IsLessThanUnder(ord,a,a*b); true
\end{verbatim}

\begin{verbatim}
gap> IsLessThanOrEqualUnder(ord,a*b,a*b); true
\end{verbatim}

\begin{verbatim}
gap> IsIncomparableUnder(ord,a,b); true
\end{verbatim}

\begin{verbatim}
gap> FamilyForOrdering(ord) = FamilyObj(a); true
\end{verbatim}
29.3 Orderings on families of associative words

We now consider orderings on families of associative words.

1. \textit{IsOrderingOnFamilyOfAssocWords(\ ord )}\hspace{1cm} P

for an ordering \textit{ord}, returns true if \textit{ord} is an ordering over a family 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 \cite{Sim94}.

2. \textit{IsTranslationInvariantOrdering(\ ord )}\hspace{1cm} P

for an ordering \textit{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 \textit{ord} over a family \textit{fam}, with alphabet \textit{X} is translation invariant if \textit{IsLessThanUnder(\ ord, u, v)} implies that for any \( a, b \in X^* \) \textit{IsLessThanUnder(\ ord, a * u * b, a * v * b)}.

3. \textit{IsReductionOrdering(\ ord )}\hspace{1cm} P

for an ordering \textit{ord} on a family of associative words, returns true if and only if the ordering is a reduction ordering. An ordering \textit{ord} is a reduction ordering if it is founded and translation invariant.

4. \textit{OrderingOnGenerators(\ ord )}\hspace{1cm} A

for an ordering \textit{ord} on a family of associative words, returns a list \textit{alphabet} 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 \textit{WeightLexOrdering} (29.3.8).

5. \textit{LexicographicOrdering(\ fam )}\hspace{1cm} O

\textit{LexicographicOrdering(\ fam, gensord )}\hspace{1cm} O

\textit{LexicographicOrdering(\ fam, alphabet )}\hspace{1cm} O

\textit{LexicographicOrdering(\ f )}\hspace{1cm} O

\textit{LexicographicOrdering(\ f, alphabet )}\hspace{1cm} O

\textit{LexicographicOrdering(\ f, gensord )}\hspace{1cm} O

In the first form, for a family \textit{fam} of associative words, \textit{LexicographicOrdering} returns the lexicographic ordering on the elements of \textit{fam}.

In the second form, for a family \textit{fam} of associate words and a list \textit{alphabet} which is the actual list of generators in the desired order, \textit{LexicographicOrdering} returns the lexicographic ordering on the elements of \textit{fam} with the ordering on the alphabet as given.

In the third form, for a family \textit{fam} of associative words and a list \textit{gensorder} of the length of the alphabet, \textit{LexicographicOrdering} returns the lexicographic ordering on the elements of \textit{fam} with the order on the alphabet given by \textit{gensord}.

In the fourth form, for a free semigroup of a free monoid \textit{f}, \textit{LexicographicOrdering} returns the lexicographic ordering on the family of the elements of \textit{f} with the order in the alphabet being the default one.

In the fifth form, for a free semigroup or a free monoid \textit{f} and a list \textit{alphabet} which is the actual list of generators in the desired order, \textit{LexicographicOrdering} returns the lexicographic ordering on the elements of \textit{f} with the ordering on the alphabet as given.

In the sixth form, for a free semigroup of a free monoid \textit{f}, and a list \textit{gensorder}, \textit{LexicographicOrdering} returns the lexicographic ordering on the elements of \textit{f} with the order on the alphabet given by \textit{gensord}.
Chapter 29. Orderings

```gap
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
```

6 ▶ **ShortLexOrdering**

```
O ▶ ShortLexOrdering( fam )
O ▶ ShortLexOrdering( fam, alphabet )
O ▶ ShortLexOrdering( fam, gensord )
O ▶ ShortLexOrdering( f )
O ▶ ShortLexOrdering( f, alphabet )
O ▶ ShortLexOrdering( f, gensord )
```

In the first form, for a family `fam` of associative words, **ShortLexOrdering** returns the ShortLex ordering on the elements of `fam` with the order in the alphabet being the default one.

In the second form, for a family `fam` of associate words and a list `alphabet` which is the actual list of generators in the desired order, **ShortLexOrdering** returns the ShortLex ordering on the elements of `fam` with the order on the alphabet as given.

In the third form, for a family `fam` of associative words and a list `gensorder` of the length of the alphabet, **ShortLexOrdering** returns the ShortLex ordering on the elements of `fam` with the order on the alphabet given by `gensord`.

In the fourth form, for a free semigroup of a free monoid `f`, **ShortLexOrdering** returns the ShortLex ordering on the family of the elements of `f` with the order in the alphabet being the default one.

In the fifth form, for a free semigroup or a free monoid `f` and a list `alphabet` which is the actual list of generators in the desired order, **ShortLexOrdering** returns the ShortLex ordering on the elements of `f` with the ordering on the alphabet as given.

In the sixth form, for a free semigroup of a free monoid `f`, and a list `gensorder`, **ShortLexOrdering** returns the ShortLex ordering on the elements of `f` with the order on the alphabet given by `gensord`.

7 ▶ **IsShortLexOrdering**

```
P for an ordering `ord` of a family of associative words, returns true if and only if `ord` is a ShortLex ordering.
```

```gap
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
```

8 ▶ **WeightLexOrdering**

```
O ▶ WeightLexOrdering( fam, alphabet, wt )
O ▶ WeightLexOrdering( fam, gensord, wt )
O ▶ WeightLexOrdering( f, alphabet, wt )
O ▶ WeightLexOrdering( f, gensord, wt )
```

In the first form, for a family `fam` of associative words and a list `wt`, **WeightLexOrdering** returns the WeightLex ordering on the elements of `fam` with the order in the alphabet being the default one and the weights of the letters in the alphabet being given by `wt`. 

In the second form, for a family $\text{fam}$ of associative words, a list $\text{wt}$ and a list $\text{gensorder}$ of the length of the alphabet, $\text{WeightLexOrdering}$ returns the WeightLex ordering on the elements of $\text{fam}$ with the order on the alphabet given by $\text{gensord}$ and the weights of the letters in the alphabet being given by $\text{wt}$.

In the third form, for a free semigroup of a free monoid $f$ and a list $\text{wt}$, $\text{WeightLexOrdering}$ returns the WeightLex ordering on the family of the elements of $f$ with the order in the alphabet being the default one and the weights of the letters in the alphabet being given by $\text{wt}$.

In the fourth form, for a free semigroup of a free monoid $f$, a list $\text{wt}$ and a list $\text{gensorder}$ of the length of the alphabet, $\text{WeightLexOrdering}$ returns the WeightLex ordering on the elements of $f$ with the order on the alphabet given by $\text{gensord}$ and the weights of the letters in the alphabet being given by $\text{wt}$.

$\text{IsWeightLexOrdering}(\text{ord})$ for an ordering $\text{ord}$ on a family of associative words, returns $\text{true}$ if and only if $\text{ord}$ is a WeightLex ordering.

$\text{WeightOfGenerators}(\text{ord})$ for a WeightLex ordering $\text{ord}$, returns a list $l$ 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 $\text{OrderingOnGenerators}$ (see 29.3.4).

```
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 ]
```

In the first form, for a family of associative words, $\text{BasicWreathProductOrdering}$ returns the basic wreath product ordering on the elements of $\text{fam}$ with the order in the alphabet being the default one.

In the second form, for a family of associative words and a list $\text{alphabet}$, $\text{BasicWreathProductOrdering}$ returns the basic wreath product ordering on the elements of $\text{fam}$ with the order on the alphabet given by $\text{alphabet}$.

In the third form, for a family of associative words and a list $\text{gensorder}$ of the length of the alphabet, $\text{BasicWreathProductOrdering}$ returns the basic wreath product ordering on the elements of $\text{fam}$ with the order on the alphabet given by $\text{gensord}$.

In the fourth form, for a free semigroup of a free monoid $f$, $\text{BasicWreathProductOrdering}$ returns the basic wreath product ordering on the family of the elements of $f$ with the order in the alphabet being the default one.
In the fifth form, for a free semigroup or a free monoid \( f \), and a list \( \text{alphabet} \) of generators, BasicWreathProductOrdering returns the basic wreath product ordering on the family of the elements of \( f \) with the order on the alphabet given by \( \text{alphabet} \).

In the sixth form, for a free semigroup or a free monoid \( f \), and a list \( \text{gensorder} \), BasicWreathProductOrdering returns the basic wreath product ordering on the family of the elements of \( f \) with the order on the alphabet given by \( \text{gensord} \).

\begin{verbatim}
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.3);
false
\end{verbatim}

returns the wreath product ordering of the family \( \text{fam} \) of associative words or a free semigroup/monoid \( f \). The ordering on the generators may be omitted (in which case the default one is considered), or may be given either by a list \( \text{alphabet} \) consisting of the alphabet of the family in the appropriate ordering, or by a list \( \text{gensord} \) giving the permutation of the alphabet. It also needs a list \( \text{levels} \) giving the levels of each generator. Notice that this list gives the levels of the generators in the new ordering (not necessarily the default one), i.e. \( \text{levels}[i] \) is the level of the generator that comes \( i \)-th in the ordering of generators given by \( \text{alphabet} \) or \( \text{gensord} \).

\begin{verbatim}
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 ]
\end{verbatim}

for a wreath product ordering \( \text{ord} \), returns the levels of the generators as given at creation (with respect to OrderingOnGenerators; see 29.3.4).
30 Domains and their Elements

**Domain** is GAP’s name for structured sets. The ring of Gaussian integers \( \mathbb{Z}[i] \) 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` (see 58.4) and the field of rationals is predefined as `Rationals` (see 16). Most domains are constructed by functions, which are called **domain constructors** (see 30.3). For example the group \( D_{12} \) is constructed by the construction `Group( (1,2,3,4,5,6), (2,6)(3,5) )` (see 37.2.1) and the finite field with 16 elements is constructed by `GaloisField( 16 )` (see 57.3.1).

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` 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`, 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 \( \exp(2\pi i/4) \) is written as \( E(4) \) in GAP.)

The most important facts about domains are stated in Chapter 7 of the GAP Tutorial.

There are only few operations especially for domains (see 30.9), operations such as `Intersection` and `Random` are defined for the more general situation of collections (see Chapter 28).

### 30.1 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 31.2).

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 30.6.

### 30.2 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 \texttt{AsSSortedList} (see 28.2.8) 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 28.2.8), and \( = \) will return \texttt{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 \texttt{IsSubset}.)

### 30.3 Constructing Domains

For several operational structures (see 30.1), GAP provides functions to construct domains with this structure. For example, \texttt{Group} returns groups, \texttt{VectorSpace} returns vector spaces etc.

\begin{verbatim}
▶ Struct( arg1, arg2, ... )
\end{verbatim}

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 \texttt{Group}, \texttt{Algebra}, or \texttt{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 \textit{Struct} first checks whether the arguments admit the construction of a domain with the desired structure. This is done by calling the operation

\begin{verbatim}
2 ▶ IsGeneratorsOfStruct( [info, ]gens) O
\end{verbatim}

where \textit{arglist} is the list of given generators and \textit{info} an argument of \textit{Struct}, for example the field of scalars in the case that a vector space shall be constructed. If the check failed then \textit{Struct} returns \textit{fail}, otherwise it returns the result of \textit{StructByGenerators} (see below). (So if one wants to omit the check then one should call \textit{StructByGenerators} directly.)

\begin{verbatim}
3 ▶ GeneratorsOfStruct( D) A
\end{verbatim}

For a domain \( D \) with operational structure corresponding to \textit{Struct}, the attribute \textit{GeneratorsOfStruct} returns a list of corresponding generators of \( D \). If these generators were not yet stored in \( D \) then \( D \) must know \textit{some} generators if \textit{GeneratorsOfStruct} 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.

\begin{verbatim}
4 ▶ StructByGenerators( [info, ]gens ) O
\end{verbatim}

Domain construction from generators \textit{gens} is implemented by operations \textit{StructByGenerators}, which are called by the simple functions \textit{Struct}; methods can be installed only for the operations. Note that additional information \textit{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 \textit{GeneratorsOfStruct} value of the returned domain need \textbf{not} be equal to \textit{gens}. But if a domain \( D \) is printed as \textit{Struct}([a, b, ...]) and if there is an attribute \textit{GeneratorsOfStruct} then the list \textit{GeneratorsOfStruct} of \( D \) is guaranteed to be equal to \([ a, b, ... \]).

\begin{verbatim}
5 ▶ StructWithGenerators( [info, ]gens ) O
\end{verbatim}

The only difference between \textit{StructByGenerators} and \textit{StructWithGenerators} is that the latter guarantees that the \textit{GeneratorsOfStruct} value of the result is equal to the given generators \textit{gens}.

\begin{verbatim}
6 ▶ ClosureStruct( D, obj ) O
\end{verbatim}

For constructing a domain as the closure of a given domain with an element or another domain, one can use the operation \textit{ClosureStruct}. It returns the smallest domain with operational structure corresponding to \textit{Struct} that contains \( D \) as a subset and \textit{obj} as an element.

\section*{30.4 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 30.6) then it is reasonable to ask for the group that is equal to \( M \).

\begin{verbatim}
1 ▶ AsStruct( [info, ]D ) O
\end{verbatim}

If \( D \) is a domain that is closed under the operational structure given by \textit{Struct} then \textit{AsStruct} returns a domain \( E \) that consists of the same elements (that is, \( D = E \)) and that has this operational structure (that is, \textit{IsStruct}( \( E \)) is \textit{true}); if \( D \) is not closed under the structure given by \textit{Struct} then \textit{AsStruct} returns \textit{fail}. 
Chapter 30. Domains and their Elements

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 $\text{AsVectorSpace}( \text{GF}(2), D )$; 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 $\text{AsStruct}$ is in fact an attribute (see 13.5).

See the index of the GAP Reference Manual for an overview of the available $\text{AsStruct}$ functions.

30.5 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 30.4).

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 42) then one can compute an isomorphism to a polycyclicly presented group $G$, say (see Chapter 43); 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 $G$.

If $D$ is a domain that is closed under the operational structure given by Struct then $\text{IsomorphismRepStruct}( D )$ returns a mapping $\text{hom}$ from $D$ to a domain $E$ having structure given by Struct, such that $\text{hom}$ respects the structure $\text{Struct}$ and $\text{Rep}$ describes the representation of the elements in $E$. If no domain $E$ with the required properties exists then fail is returned.

For example, $\text{IsomorphismPermGroup}$ (see 41.2.1) takes a group as its argument and returns a group homomorphism (see 38) onto an isomorphic permutation group (see Chapter 41) provided the original group is finite; for infinite groups, $\text{IsomorphismPermGroup}$ returns fail. Similarly, $\text{IsomorphismPcGroup}$ (see 44.5.2) returns a group homomorphism from its argument to a polycyclicly presented group (see 44) if the argument is polycyclic, and fail otherwise.

See the index of the GAP Reference Manual for an overview of the available $\text{IsomorphismRepStruct}$ functions.

30.6 Domain Categories

As mentioned in 30.1, the operational structure of a domain is fixed when the domain is constructed. For example, if $D$ was constructed by $\text{Monoid}$ 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, $\text{IsGroup}$ returns false for $D$. The operational structure determines which operations are applicable for a domain, so for example $\text{SylowSubgroup}$ is not defined for $D$ and therefore will signal an error.

The functions $\text{IsStruct}$ implement the tests whether a domain $D$ has the respective operational structure (upon construction). $\text{IsStruct}$ is a filter (see 13) that involves certain categories (see 13.3) and usually also certain properties (see 13.7). For example, $\text{IsGroup}$ is equivalent to $\text{IsMagmaWithInverses}$ and $\text{IsAssociative}$, 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.

- $\text{IsDomain}$ is implied by each domain category,
- $\text{IsMagma}$ is implied by each category that describes the closure under multiplication $\ast$, 

- **IsAdditiveMagma** is implied by each category that describes the closure under addition $+$,
- **IsMagmaWithOne** implies **IsMagma**: 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** implies **IsMagmaWithOne**: 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 54, 56, 37, and 59, respectively. More general functions for arbitrary collections can be found in Chapter 28.

### 30.7 Parents

- **Parent** ($D$)
- **SetParent** ($D, P$)
- **HasParent** ($D$)

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** (see 30.13.1) 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 6.2 in “Extending GAP”.) 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** (see 30.13.1), 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 returns **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**, are described in Section 30.8. Whenever a function has this property, the Reference Manual states this explicitly. Note that these functions do not guarantee a certain parent, for example **DerivedSubgroup** (see 37.11.3) 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**.
Chapter 30. Domains and their Elements

```gap
gap> g := Group( (1,2,3), (1,2) );
; h := Group( (1,2) );
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
```

### 30.8 Constructing Subdomains

For many domains \( D \), there are functions that construct certain subsets \( S \) of \( D \) as domains with parent (see 30.7) already set to \( D \). For example, if \( G \) is a group that contains the elements in the list \( \text{gens} \) then `Subgroup( G, gens )` returns a group \( S \) that is generated by the elements in \( \text{gens} \) and with `Parent( S ) = G`.

1. `Substruct( 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 `Substruct` (for example `Subgroup`, `Subalgebra`, `Subfield`) returns domains with structure `Struct` and parent set to the first argument.

2. `SubstructNC( D, gens )`

   Each function `Substruct` 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 `SubstructNC` instead; the suffix `NC` stands for “no check”.

3. `AsSubstruct( D, S )`

   First constructs `Asstruct([ info, ]S )`, where `info` depends on \( D \) and \( S \), and then sets the parent (see 30.7) of this new domain to \( D \).

4. `IsSubstruct( D, S )`

   There is no real need for functions that check whether a domain \( S \) is a `Substruct` 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` (see 28.4.1) 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 `IsSubstruct` is available in GAP then it is implemented as first a call to `IsStruct` for the second argument and then a call to `IsSubset` for the two arguments.

### 30.9 Operations for Domains

For the meaning of the attributes `Characteristic`, `One`, `Zero` in the case of a domain argument, see 30.10.

1. `IsGeneralizedDomain( D )` C

   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.

2 GeneratorsOfDomain( $D$ )

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 30.1), the only possible choice of elements is to take all elements of $D$. See 30.3 and 30.4 for the concepts of other notions of generation.

Domain( [Fam, generators ] )

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 first argument, and the returned (empty) domain lies in the collections family of Fam.

DomainByGenerators is the operation called by Domain.

30.10 Attributes and Properties of Elements

The following attributes and properties for elements and domains correspond to the operational structure.

Characteristic( obj )

Characteristic returns the characteristic of obj, where obj must either be an additive element, a domain or a family.

For a domain $D$, the characteristic is defined if $D$ is closed under addition and has a zero element $z = \text{Zero}(D)$ (see 30.10.3); in this case, Characteristic( $D$ ) is the smallest positive integer $p$ such that $p \times x = z$ for all elements $x$ in $D$, if such an integer exists, and the integer zero 0 otherwise.

If a family has a characteristic then this means that all domains of elements in this family have this characteristic. In this case, also each element in the family has this characteristic. (Note that also the zero element $z$ of a finite field in characteristic $p$ has characteristic $p$, although $n \times z = z$ for any integer $n$.)

OneImmutable( obj )

OneAttr( obj )

One( obj )

Identity( obj )

OneMutable( obj )

OneOp( obj )

OneSameMutability( obj )

OneSM( obj )

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 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, Identity, OneOp, and OneSM are synonyms as listed above.
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 28.1.1) 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 33.1.2); use MultiplicativeNeutralElement (see 33.4.10) 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 54). 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 (see 28.3.7), but Representative is allowed to fetch the identity of a domain D only if HasOne(D) is true.

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 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, Zero, ZeroOp and ZeroSM are synonyms as listed above.

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 28.1.1) 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 53.1.5); use AdditiveNeutralElement (see 53.3.5) 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 (see 28.3.7), but Representative is allowed to fetch the zero of a domain D only if HasZero(D) is true.

MultiplicativeZeroOp( elt ) returns the element z in the family F of elt with the property that z * m = z = m * z holds for all m ∈ F, if such an element is known.

Families of elements in the category IsMultiplicativeElementWithZero often arise from adjoining a new zero to an existing magma. See 33.2.12 for details.
IsOne( elm )
is true if elm = One( elm ), and false otherwise.

IsZero( elm )
is true if elm = Zero( elm ), and false otherwise.

IsIdempotent( elt )
true iff elt is its own square. (Even if IsZero(elt) is also true.)

InverseImmutable( elm )
InverseAttr( elm )
Inverse( elm )
InverseMutable( elm )
InverseSaneMutability( elm )
InverseSM( elm )

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 20.1.1) 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 31.13). For a bijective mapping f whose source and range differ, InverseGeneralMapping (see 31.1.3) 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, Inverse, InverseOp and InverseSM are synonyms as listed above.

The default method of Inverse calls InverseMutable (note that methods for InverseMutable must not delegate to Inverse); other methods to compute inverses need to be installed only for InverseMutable and (in the case of copyable objects) InverseSameMutability.

AdditiveInverseImmutable( elm )
AdditiveInverseAttr( elm )
AdditiveInverse( elm )
AdditiveInverseMutable( elm )
AdditiveInverseOp( elm )
AdditiveInverseSaneMutability( elm )
AdditiveInverseSM( elm )

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 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.

\[ \text{AdditiveInverseSameMutability}(elm) \] is equivalent to \(-elm\).

\text{AdditiveInverseAttr}, \text{AdditiveInverse}, \text{AdditiveInverseOp} and \text{AdditiveInverseSM} are synonyms as listed above.

The default method of \text{AdditiveInverse} calls \text{AdditiveInverseMutable} (note that methods for \text{AdditiveInverseMutable} must not delegate to \text{AdditiveInverse}); so other methods to compute additive inverses need to be installed only for \text{AdditiveInverseMutable} and (in the case of copyable objects) \text{AdditiveInverseSameMutability}.

\text{Order}(elm)

is the multiplicative order of \(elm\). This is the smallest positive integer \(n\) such that \(elm^n = \text{One}(elm)\) if such an integer exists. If the order is infinite, \text{Order} may return the value \text{infinity}, but it also might run into an infinite loop trying to test the order.

### 30.11 Comparison Operations for Elements

Binary comparison operations have been introduced already in 4.11. The underlying operations for which methods can be installed are the following.

\[
\begin{align*}
\text{O} & \quad \text{\(\leq\)(left-expr, right-expr)} \\
\text{O} & \quad \text{\(<\)(left-expr, right-expr)}
\end{align*}
\]

Note that the comparisons via \(\neq\), \(\leq\), \(\geq\), and \(\geq\) are delegated to the operations \(\leq\) and \(\langle\).

In general, objects in different families cannot be compared with \(<\). For the reason and for exceptions from this rule, see 4.11.

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.

\[
\begin{align*}
\text{P} & \quad \text{\text{CanEasilyCompareElements}(obj)} \\
\text{F} & \quad \text{\text{CanEasilyCompareElementsFamily}(fam)} \\
\text{P} & \quad \text{\text{CanEasilySortElements}(obj)} \\
\text{F} & \quad \text{\text{CanEasilySortElementsFamily}(fam)}
\end{align*}
\]

\text{CanEasilyCompareElements} indicates whether the elements in the family \text{fam} of \text{obj} can be easily compared with \(=\). (In some cases element comparisons are very hard, for example in cases where no normal forms for the elements exist.)

The default method for this property is to ask the family of \text{obj}, the default method for the family is to return \text{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 \text{false} too early in a family. Instead the function \text{CanEasilyCompareElementsFamily} should be called for the family of \text{obj} which returns \text{false} if the value of \text{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 \text{NewFamily} respectively). This guarantees that code which directly asks the family gets a right answer.

The property \text{CanEasilySortElements} and the function \text{CanEasilySortElementsFamily} behave exactly in the same way, except that they indicate that objects can be compared via \(<\). This property implies \text{CanEasilyCompareElements}, as the ordering must be total.
30.12 Arithmetic Operations for Elements

**Binary** arithmetic operations have been introduced already in 4.12. The underlying operations for which methods can be installed are the following.

1. \(\langle left-expr, right-expr \rangle\)
2. \(\langle left-expr, right-expr \rangle\)
3. \(\langle left-expr, right-expr \rangle\)
4. \(\langle left-expr, right-expr \rangle\)
5. \(\langle left-expr, right-expr \rangle\)

For details about special methods for \mod, consult the index entries for “mod”.

2. \(\langle left-expr, right-expr \rangle\)

returns the product \(elm1^{-1} \cdot elm2\). 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 \(elm2\).

3. \(\langle left-expr, right-expr \rangle\)

returns the commutator of \(elm1\) and \(elm2\). The commutator is defined as the product \(elm1^{-1} \cdot elm2^{-1} \cdot elm1 \cdot elm2\).

4. \(\langle left-expr, right-expr \rangle\)

returns the element \(elm1 \cdot elm2 - elm2 \cdot elm1\).

The addition \(\langle left-expr, right-expr \rangle\) is assumed to be associative but **not** assumed to be commutative (see 53.3.1). The multiplication \(\langle left-expr, right-expr \rangle\) is **not** assumed to be commutative or associative (see 33.4.9, 33.4.7).

5. \(\langle left-expr, right-expr \rangle\)

\(Sqrt\) 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, \(ER\) (see 18.4.2) is the \(Sqrt\) method for rationals (see 16.1.1).

30.13 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 28.4.2) of two domains, or a Sylow subgroup of a given group (see 37.12.1).

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 30.8 for the exception), this is the right moment to take advantage of the subset relation.

Analogous relations occur when a **factor structure** is created from a domain and a subset, and when a domain **isomorphic** to a given one is created.
Methods for this operation transfer possibly useful information from the domain \textit{super} to its subset \textit{sub}, and vice versa.

\texttt{UseSubsetRelation} is designed to be called automatically whenever substructures of domains are constructed. So the methods must be \textit{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 \texttt{TryNextMethod()}. This default method deals with the information that is available by the calls of \texttt{InstallSubsetMaintenance} in the GAP library.

\begin{verbatim}
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
\end{verbatim}

Methods for this operation transfer possibly useful information from the domain \textit{old} to the isomorphic domain \textit{new}.

\texttt{UseIsomorphismRelation} is designed to be called automatically whenever isomorphic structures of domains are constructed. So the methods must be \textit{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 \texttt{TryNextMethod()}. This default method deals with the information that is available by the calls of \texttt{InstallIsomorphismMaintenance} in the GAP library.

\begin{verbatim}
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
\end{verbatim}

Methods for this operation transfer possibly useful information from the domain \textit{numer} or its subset \textit{denom} to the domain \textit{factor} that is isomorphic to the factor of \textit{numer} by \textit{denom}, and vice versa. \textit{denom} may be \texttt{fail}, for example if \textit{factor} is just known to be a factor of \textit{numer} but \textit{denom} is not available as a GAP object; in this case those factor relations are used that are installed without special requirements for \textit{denom}.

\texttt{UseFactorRelation} is designed to be called automatically whenever factor structures of domains are constructed. So the methods must be \textit{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 \texttt{TryNextMethod()}. This default method deals with the information that is available by the calls of \texttt{InstallFactorMaintenance} in the GAP library.
Section 13. Relations Between Domains

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

The following functions 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 3.3 in “Programming in GAP”. For the attributes already available, such as IsFinite and IsCommutative, the maintenances are already notified.

4 InstallSubsetMaintenance( opr, super_req, sub_req )

opr must be a property or an attribute. The call of InstallSubsetMaintenance has the effect that for a domain \( D \) in the filter \( \text{super\_req} \), and a domain \( S \) in the filter \( \text{sub\_req} \), the call UseSubsetRelation( \( D, S \) ) (see 30.13.1) sets a known value of \( \text{opr} \) for \( D \) as value of \( \text{opr} \) also for \( S \). A typical example for which InstallSubsetMaintenance is applied is given by \( \text{opr} = \text{IsFinite}, \text{super\_req} = \text{IsCollection and IsFinite}, \) and \( \text{sub\_req} = \text{IsCollection} \).

If \( \text{opr} \) is a property and the filter \( \text{super\_req} \) lies in the filter \( \text{opr} \) then we can use also the following inverse implication. If \( D \) is in the filter whose intersection with \( \text{opr} \) is \( \text{super\_req} \) and if \( S \) is in the filter \( \text{sub\_req} \), \( S \) is a subset of \( D \), and the value of \( \text{opr} \) for \( S \) is \text{false} then the value of \( \text{opr} \) for \( D \) is also \text{false}.

5 InstallIsomorphismMaintenance( opr, old_req, new_req )

opr must be a property or an attribute. The call of InstallIsomorphismMaintenance has the effect that for a domain \( D \) in the filter \( \text{old\_req} \), and a domain \( E \) in the filter \( \text{new\_req} \), the call UseIsomorphismRelation( \( D, E \) ) (see 30.13.2) sets a known value of \( \text{opr} \) for \( D \) as value of \( \text{opr} \) also for \( E \). A typical example for which InstallIsomorphismMaintenance is applied is given by \( \text{opr} = \text{Size}, \text{old\_req} = \text{IsCollection and new\_req = IsCollection} \).

For the other direction, if \( \text{old\_req} \) involves the filter \( \text{opr} \) then a known \text{false} value of \( \text{opr} \) for \( E \) implies a \text{false} value for \( D \) provided that \( D \) lies in the filter obtained from \( \text{numerator}\_\text{req} \) by removing \( \text{opr} \).

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 (see 30.13.2) will transfer a known IsFinite value because of the installed factor maintained method.
30.14 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 33) makes sense only if its elements can be multiplied, and the latter is indicated by the category \texttt{IsMultiplicativeElement} 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.

1\hspace{1em} \texttt{IsExtAElement( obj )} \hspace{1em} C

An external additive element is an object that can be added via + with other elements (not necessarily in the same family, see 13.1).

2\hspace{1em} \texttt{IsNearAdditiveElement( obj )} \hspace{1em} C

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.

3\hspace{1em} \texttt{IsAdditiveElement( obj )} \hspace{1em} C

An additive element is an object that can be added via + with elements in its family (see 13.1); this addition is commutative.

4\hspace{1em} \texttt{IsNearAdditiveElementWithZero( obj )} \hspace{1em} C

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 \texttt{Zero} (see 30.10.3); this addition is not necessarily commutative.

5\hspace{1em} \texttt{IsAdditiveElementWithZero( obj )} \hspace{1em} C

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 \texttt{Zero} (see 30.10.3); this addition is commutative.

6\hspace{1em} \texttt{IsNearAdditiveElementWithInverse( obj )} \hspace{1em} C

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 \texttt{Zero} (see 30.10.3) and \texttt{AdditiveInverse} (see 30.10.9); this addition is not necessarily commutative.

7\hspace{1em} \texttt{IsAdditiveElementWithInverse( obj )} \hspace{1em} C

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 \texttt{Zero} (see 30.10.3) and \texttt{AdditiveInverse} (see 30.10.9); this addition is commutative.

8\hspace{1em} \texttt{IsExtLElement( obj )} \hspace{1em} C

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).

9\hspace{1em} \texttt{IsExtRElement( obj )} \hspace{1em} C

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).

10\hspace{1em} \texttt{IsMultiplicativeElement( obj )} \hspace{1em} C

A multiplicative element is an object that can be multiplied via \(*\) with elements in its family (see 13.1).
11▶ IsMultiplicativeElementWithOne( obj )

A **multiplicative element-with-one** is an object that can be multiplied via \( \ast \) with elements in its family (see 13.1), and that is an admissible argument for the operation \( \text{One} \) (see 30.10.2).

12▶ IsMultiplicativeElementWithZero( elt )

Elements in a family which can be the operands of the \( \ast \) and the operation MultiplicativeZero.

13▶ IsMultiplicativeElementWithInverse( obj )

A **multiplicative element-with-inverse** is an object that can be multiplied via \( \ast \) with elements in its family (see 13.1), and that is an admissible argument for the operations \( \text{One} \) (see 30.10.2) and \( \text{Inverse} \) (see 30.10.8). (Note the word “admissible”: an object in this category does not necessarily have an inverse, \( \text{Inverse} \) may return \text{fail}.)

14▶ IsVector( obj )

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 \( \text{IsAdditiveElementWithInverse} \) is implied by the join of its collections category and \( \text{IsList} \), the family of a list entry may not imply \( \text{IsAdditiveElementWithInverse} \) for all its elements.

15▶ IsNearRingElement( obj )

\( \text{IsNearRingElement} \) is just a synonym for the join of \( \text{IsNearAdditiveElementWithInverse} \) and \( \text{IsMultiplicativeElement} \).

16▶ IsRingElement( obj )

\( \text{IsRingElement} \) is just a synonym for the join of \( \text{IsAdditiveElementWithInverse} \) and \( \text{IsMultiplicativeElement} \).

17▶ IsNearRingElementWithOne( obj )

\( \text{IsNearRingElementWithOne} \) is just a synonym for the join of \( \text{IsNearAdditiveElementWithInverse} \) and \( \text{IsMultiplicativeElementWithOne} \).

18▶ IsRingElementWithOne( obj )

\( \text{IsRingElementWithOne} \) is just a synonym for the join of \( \text{IsAdditiveElementWithInverse} \) and \( \text{IsMultiplicativeElementWithOne} \).

19▶ IsNearRingElementWithInverse( obj )

\( \text{IsNearRingElementWithInverse} \) is just a synonym for the join of \( \text{IsNearAdditiveElementWithInverse} \) and \( \text{IsMultiplicativeElementWithInverse} \).

20▶ IsRingElementWithInverse( obj )

\( \text{IsRingElementWithInverse} \) and \( \text{IsScalar} \) are just synonyms for the join of \( \text{IsAdditiveElementWithInverse} \) and \( \text{IsMultiplicativeElementWithInverse} \).

More special categories of this kind are described in the contexts where they arise, they are \( \text{IsRowVector} \) (see 23), \( \text{IsMatrix} \) (see 24.1.1), \( \text{IsOrdinaryMatrix} \) (see 24.1.2), and \( \text{IsLieMatrix} \) (see 24.1.3).
Chapter 30. Domains and their Elements

30.15 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 3.6 in “Programming in GAP”). 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 28.1.4).

For example, the multiplication of permutations is associative, and it is stored in the family of permutations that each permutation lies in IsAssociativeElement. As a consequence, each magma consisting of permutations (more precisely: each collection that lies in the family CollectionsFamily( PermutationsFamily ), see 28.1.1) automatically lies in CategoryCollections( IsAssociativeElement ). A magma in this category is always known to be associative, via a logical implication (see 2.7 in “Programming in GAP”).

Similarly, if a family knows that all its elements are in the categories IsJacobianElement and IsZeroSquaredElement, then each algebra of these elements is automatically known to be a Lie algebra (see 60).

1 ► IsAssociativeElement( obj ) C
► IsAssociativeElementCollection( obj ) C
► IsAssociativeElementCollColl( obj ) C

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 24.1.2) whose entries are also in IsAssociativeElement.

2 ► IsAdditivelyCommutativeElement( obj ) C
► IsAdditivelyCommutativeElementCollection( obj ) C
► IsAdditivelyCommutativeElementCollColl( obj ) C
► IsAdditivelyCommutativeElementFamily( obj ) C

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.

3 ► IsCommutativeElement( obj ) C
► IsCommutativeElementCollection( obj ) C
► IsCommutativeElementCollColl( obj ) C

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.

4 ► IsFiniteOrderElement( obj ) C
► IsFiniteOrderElementCollection( obj ) C
► IsFiniteOrderElementCollColl( obj ) C

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.

5 ► IsJacobianElement( obj ) C
► IsJacobianElementCollection( obj ) C
► IsJacobianElementCollColl( obj ) C

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 $F$. 

1 ▶ IsAssociativeElement ( obj )
2 ▶ IsAdditivelyCommutativeElement ( obj )
3 ▶ IsCommutativeElement ( obj )
4 ▶ IsFiniteOrderElement ( obj )
5 ▶ IsJacobianElement ( obj )
For example, each Lie matrix (see 24.1.3) lies in this category.

\[
6 \quad \text{IsZeroSquaredElement( } \text{obj } \text{ )} \\
\text{C} \\
\text{IsZeroSquaredElementCollection( } \text{obj } \text{ )} \\
\text{C} \\
\text{IsZeroSquaredElementCollColl( } \text{obj } \text{ )} \\
\text{C}
\]

An element \( \text{obj} \) in the category \text{IsZeroSquaredElement} knows that \( \text{obj}^2 = \text{Zero( } \text{obj } \text{ )} \). For example, each Lie matrix (see 24.1.3) 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.
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 \( \text{Rel} \) of the direct product \( S \times R \), which is called the underlying relation of \( F \). \( S \), \( R \), and \( \text{Rel} \) are generalized domains (see Chapter 12.4). The corresponding attributes for general mappings are `Source`, `Range`, and `UnderlyingRelation`.

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 31.5); both should not apply to domains.

Each element of an underlying relation of a general mapping lies in the category of tuples (see 31).

For each \( s \in S \), the set \( \{ r \in R | (s, r) \in \text{Rel} \} \) is called the set of **images** of \( s \). Analogously, for \( r \in R \), the set \( \{ s \in S | (s, r) \in \text{Rel} \} \) 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 \( \text{map1} \) and \( \text{map2} \) are the same, then one considers the union of the preimages of \( \text{map1} \) and \( \text{map2} \) as a strictly ordered set. The smaller of \( \text{map1} \) and \( \text{map2} \) 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 31.6.

Some technical details of general mappings are described in section 31.12.

\[ \text{IsTuple( obj )} \]

`IsTuple` is a subcategory of the meet of `IsDenseList` (see 21.1.2), `IsMultiplicativeElementWithInverse` (see 30.14.13), and `IsAdditiveElementWithInverse` (see 30.14.7), 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.

The sum and the product of a tuple and a list in `IsListDefault` is the list of sums and products, respectively. The sum and the product of a tuple and a non-list is the tuple of componentwise sums and products, respectively.
### 31.1 Creating Mappings

1. GeneralMappingByElements( $S$, $R$, elms )

   is the general mapping with source $S$ and range $R$, and with underlying relation consisting of the tuples collection $elms$.

2. MappingByFunction( $S$, $R$, fun )
   - MappingByFunction( $S$, $R$, fun, invfun )
   - MappingByFunction( $S$, $R$, fun, 'false, prefun '

   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.

   In the third variant, a function prefun is given that can be used to compute a single preimage. In this case, the third entry must be false.

   MappingByFunction creates a mapping which IsNonSPGeneralMapping.

3. InverseGeneralMapping( $map$ )

   The inverse general mapping of a general mapping $map$ is the general mapping whose underlying relation (see 31.2.9) contains a pair $(r, s)$ if and only if the underlying relation of $map$ contains the pair $(s, r)$.

   See the introduction to Chapter 31 for the subtleties concerning the difference between InverseGeneralMapping and Inverse.

   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.

4. CompositionMapping( $map1$, $map2$, ... )

   CompositionMapping allows one to compose arbitrarily many general mappings, and delegates each step to CompositionMapping2.

   Additionally, the properties IsInjective and IsSingleValued 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 and IsSurjective are maintained. (So one should not call CompositionMapping2 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 31.1.6).

5. CompositionMapping2( $map2$, $map1$ )

   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 *).

6. IsCompositionMappingRep( $map$ )

   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.
ConstituentsCompositionMapping( map )

If map is stored in the representation IsCompositionMappingRep as composition of two mappings map1 and map2, this function returns the two constituent mappings in a list [map1,map2].

ZeroMapping( S, R )

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.)

IdentityMapping( D )

is the bijective mapping with source and range equal to the collection D, which maps each element of D to itself.

Embedding( S, T )

Embedding( S, i )

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 (61.1.3); for group products, see 47.5 for a general description, or for examples see 47.1 for direct products, 47.2 for semidirect products, or 47.4 for wreath products; or for magma rings see 63.3.

Projection( S, T )

Projection( S, i )

Projection( S )

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 subdomain of S. Various methods are defined for group products; see 47.5 for a general description, or for examples see 47.1 for direct products, 47.2 for semidirect products, 47.3 for subdirect products, or 47.4 for wreath products.

RestrictedMapping( map, subdom )

If subdom is a subdomain of the source of the general mapping map, this operation returns the restriction of map to subdom.

31.2 Properties and Attributes of (General) Mappings

IsTotal( map )

is true if each element in the source S of the general mapping map has images, i.e., \( s^\text{map} \neq \emptyset \) for all \( s \in S \), and false otherwise.

IsSingleValued( map )

is true if each element in the source S of the general mapping map has at most one image, i.e., \( |s^\text{map}| \leq 1 \) 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.

IsMapping( map )

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 31.2.1, 31.2.2).
Section 3. Images under Mappings

4 ▶ IsInjective( map )

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 $S$.

5 ▶ IsSurjective( map )

is true if each element in the range $R$ of the general mapping $map$ has preimages in the source $S$ of $map$, i.e., \{ $s \in S$ | $x \in s^{map}$ \} $\neq \emptyset$ for all $x \in R$, and false otherwise.

6 ▶ IsBijective( map )

A general mapping $map$ is bijective if and only if it is an injective and surjective mapping (see 31.2.3, 31.2.4, 31.2.5).

7 ▶ Range( map )

8 ▶ Source( map )

9 ▶ UnderlyingRelation( map )

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 31.2.8, 31.2.7), and $r \in \text{ImagesElm}( map, s )$.

Each element of the underlying relation is a tuple (see 31).

10 ▶ UnderlyingGeneralMapping( map )

attribute for underlying relations of general mappings

31.3 Images under Mappings

1 ▶ ImagesSource( map )

is the set of images of the source of the general mapping $map$.

ImagesSource delegates to ImagesSet, it is introduced only to store the image of $map$ as attribute value.

2 ▶ ImagesRepresentative( map, elm )

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 $map$.

3 ▶ ImagesElm( map, elm )

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 $map$.

4 ▶ ImagesSet( map, elms )

If $elms$ is a subset of the source of the general mapping $map$ then ImagesSet returns the set of all images of $elms$ under $map$.

Anything may happen if $elms$ is not a subset of the source of $map$. 
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 \( map \).

\[ ImageElm( map, elm ) \]

\( Image( map ) \) 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.

\[ Image( map, elm ) \]

\( Image( map, coll ) \)

\[ Images( map ) \]

\[ Images( map, elm ) \]

\[ Images( map, coll ) \]

\[ PreImagesRange( map ) \]

is the set of preimages of the range of the general mapping \( map \).

\( PreImagesRange \) delegates to \( PreImagesSet \), it is introduced only to store the preimage of \( map \) as attribute value.

\[ PreImagesElm( map, elm ) \]

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 \( map \).
Section 4. Preimages under Mappings

3 ▶ PreImageElm( map, elm )

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.

4 ▶ PreImagesRepresentative( map, elm )

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.

5 ▶ PreImagesSet( map, elms )

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.

6 ▶ PreImage( map )
▶ PreImage( map, elm )
▶ PreImage( map, coll )

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 31.4.7).

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 when called with one argument, and to PreImageElm resp. PreImagesSet 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.

7 ▶ PreImages( map )
▶ PreImages( map, elm )
▶ PreImages( map, coll )

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 when called with one argument, and to PreImageElm resp. PreImagesSet 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.
31.5 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 $\ast$, general mappings can be composed –in reversed order– via CompositionMapping.

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 $\text{One}$ and $\text{Inverse}$ for general mappings as follows. $\text{One}$ returns fail when called for a general mapping whose source and range differ, otherwise $\text{One}$ returns the identity mapping of the source. (Note that the source may differ from the preimage). $\text{Inverse}$ returns fail when called for a non-bijective general mapping or for a general mapping whose source and range differ; otherwise $\text{Inverse}$ returns the inverse mapping.

Besides the usual inverse of multiplicative elements, which means that $\text{Inverse}(g) \ast g = g \ast \text{Inverse}(g)$, for general mappings we have the attribute InverseGeneralMapping. If $F$ is a general mapping with source $S$, range $R$, and underlying relation $Rel$ then $\text{InverseGeneralMapping}(F)$ 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.

$\text{Inverse}$ may delegate to $\text{InverseGeneralMapping}$. $\text{InverseGeneralMapping}$ must not delegate to $\text{Inverse}$, but a known value of $\text{Inverse}$ may be fetched. So methods to compute the inverse of a general mapping should be installed for $\text{InverseGeneralMapping}$. (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.)

31.6 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.)

31.7 Magma Homomorphisms

1 $\text{IsMagmaHomomorphism}(\text{mapp})$

A MagmaHomomorphism is a total single valued mapping which respects multiplication.

2 $\text{MagmaHomomorphismByFunctionNC}(G, H, fn)$

Creates the homomorphism from $G$ to $H$ without checking that $fn$ is a homomorphism.

3 $\text{NaturalHomomorphismByGenerators}(f, s)$

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 $s$. 
31.8 Mappings that Respect Multiplication

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 \( \text{RespectsMultiplication} \) returns \( \text{true} \) 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 \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsMultiplication} \) returns \( \text{true} \) if and only if the equation \( s_1^{\cdot mapp} * s_2^{\cdot mapp} = (s_1 * s_2)^{\cdot mapp} \) holds for all \( s_1, s_2 \) in \( S \).

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 \( \text{RespectsOne} \) returns \( \text{true} \) if \( S \) and \( R \) are magmas-with-one such that \((\text{One}(S), \text{One}(R)) \in F \), and \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsOne} \) returns \( \text{true} \) if and only if the equation \( \text{One}(S)^{\cdot mapp} = \text{One}(R) \) holds.

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 \( \text{RespectsInverses} \) returns \( \text{true} \) 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 \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsInverses} \) returns \( \text{true} \) if and only if the equation \( \text{Inverse}(s)^{\cdot mapp} = \text{Inverse}(s^{\cdot mapp}) \) holds for all \( s \) in \( S \).

Mappings that are defined on a group and respect multiplication and inverses are group homomorphisms. Chapter 38 explains them in more detail.

A \( \text{GroupGeneralMapping} \) is a mapping which respects multiplication and inverses. If it is total and single valued it is called a group homomorphism.

Let \( mapp \) be a general mapping. Then \( \text{KernelOfMultiplicativeGeneralMapping} \) 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.)

Let \( mapp \) be a general mapping. Then \( \text{CoKernelOfMultiplicativeGeneralMapping} \) 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.)
31.9 Mappings that Respect Addition

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 \( \text{RespectsAddition} \) returns \( \text{true} \) 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 \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsAddition} \) returns \( \text{true} \) if and only if the equation \( s_1 \cdot mapp + s_2 \cdot mapp = (s_1 + s_2) \cdot mapp \) holds for all \( s_1, s_2 \) in \( S \).

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 \( \text{RespectsAdditiveInverses} \) returns \( \text{true} \) if \( S \) and \( R \) are additive-magmas-with-inverses such that \((s, r) \in F \) implies \((-s, -r) \in F\), and \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsAdditiveInverses} \) returns \( \text{true} \) if and only if the equation \( s \cdot mapp^\text{-1} + r \cdot mapp^\text{-1} = (s + r) \cdot mapp^\text{-1} \) holds for all \( s, r \) in \( S \).

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 \( \text{RespectsZero} \) returns \( \text{true} \) if \( S \) and \( R \) are additive-magmas-with-zero such that \((\text{Zero}(S), \text{Zero}(R)) \in F\), and \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsZero} \) returns \( \text{true} \) if and only if the equation \( S^\text{-1} \cdot mapp = \text{Zero}( R ) \) holds.

Let \( mapp \) be a general mapping. Then \( \text{KernelOfAdditiveGeneralMapping} \) returns the set of all elements in the source of \( mapp \) that have the zero of the range in their set of images.

Let \( mapp \) be a general mapping. Then \( \text{CoKernelOfAdditiveGeneralMapping} \) returns the set of all elements in the range of \( mapp \) that have the zero of the source in their set of preimages.

31.10 Linear Mappings

Also see Sections 31.8 and 31.9.

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 \( \text{RespectsScalarMultiplication} \) returns \( \text{true} \) 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 \((c \cdot s, c \cdot r) \in F\) for all \( c \in D\), and \( \text{false} \) otherwise.

If \( mapp \) is single-valued then \( \text{RespectsScalarMultiplication} \) returns \( \text{true} \) if and only if the equation \( c \cdot s \cdot mapp = (c \cdot s) \cdot mapp \) holds for all \( c \in D \) and \( s \) in \( S \).

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 \( \text{IsLeftModuleGeneralMapping} \) returns \( \text{true} \) if \( mapp \) is an \( F \)-linear mapping, and \( \text{false} \) 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 $f(s \cdot a) = s \cdot f(a)$ hold for all elements $a$, $b$ in the source of $f$ and $s \in D$.

See also KernelOfMultiplicativeGeneralMapping (31.8.5) and CoKernelOfMultiplicativeGeneralMapping (31.8.6).

### 31.11 Ring Homomorphisms

1. **IsRingGeneralMapping**($mapp$)
2. **IsRingHomomorphism**($mapp$)
3. **IsRingWithOneGeneralMapping**($mapp$)
4. **IsRingWithOneHomomorphism**($mapp$)
5. **IsAlgebraGeneralMapping**($mapp$)
6. **IsAlgebraHomomorphism**($mapp$)
7. **IsAlgebraWithOneGeneralMapping**($mapp$)
8. **IsAlgebraWithOneHomomorphism**($mapp$)
9. **IsFieldHomomorphism**($mapp$)

A general mapping is a field homomorphism if and only if it is a ring homomorphism with source a field.

### 31.12 General Mappings

1. **IsGeneralMapping**($map$)
2. **IsConstantTimeAccessGeneralMapping**($map$)
3. **IsEndoGeneralMapping**($obj$)

Each general mapping lies in the category **IsGeneralMapping**. It implies the categories **IsMultiplicativeElementWithInverse** (see 30.14.13) and **IsAssociativeElement** (see 30.15.1); for a discussion of these implications, see 31.5.

**IsConstantTimeAccessGeneralMapping**($map$) is true if the underlying relation of the general mapping $map$ knows its AsList value, and false otherwise. In the former case, $map$ is allowed to use this list for calls to ImagesElm etc.

If a general mapping has this property then its source and range are equal.

### 31.13 Technical Matters Concerning General Mappings

**Source** and **Range** are basic operations for general mappings. **UnderlyingRelation** 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**, **IsSPGeneralMapping**. (The category **IsGeneralMapping** is defined as the disjoint union of these two.)

For general mappings of the first category, **ImagesElm** and **PreImagesElm** are basic operations. (Note that in principle it is possible to delegate from **PreImagesElm** to **ImagesElm**.) Methods for the secondary operations (PreImageElm, PreImagesSet, and PreImagesRepresentative) may use (Pre)ImagesElm, and methods for (Pre)ImagesElm must not call the secondary operations. In particular, there are no generic methods for (Pre)ImagesElm.
Methods for (Pre)ImagesSet must not use PreImagesRange and ImagesSource, 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 (Pre)ImagesRepresentative and Multiplicative(Co)Kernel as basic operations here, and to make (Pre)ImagesElm 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 for the underlying relation of a general mapping may use either ImagesElm or ImagesRepresentative and the appropriate cokernel. Conversely, if AsList for the underlying relation is known then ImagesElm resp. ImagesRepresentative may delegate to it, the general mapping gets the property IsConstantTimeAccessGeneralMapping for this; note that this is not allowed if only an enumerator of the underlying relation is known.)

Secondary operations are IsInjective, IsSingleValued, IsSurjective, IsTotal; they may use the basic operations, and must not be used by them.

1 $\triangleright$ IsSPGeneralMapping( map )
2 $\triangleright$ IsNonSPGeneralMapping( map )
3 $\triangleright$ FamilyRange( Fam )
4 $\triangleright$ FamilySource( Fam )
5 $\triangleright$ FamiliesOfGeneralMappingsAndRanges( Fam )
6 $\triangleright$ GeneralMappingsFamily( sourcefam, rangefam )
7 $\triangleright$ TypeOfDefaultGeneralMapping( source, range, filter )

1 $\triangleright$ IsSPGeneralMapping( map )
2 $\triangleright$ IsNonSPGeneralMapping( map )
3 $\triangleright$ FamilyRange( Fam )
4 $\triangleright$ FamilySource( Fam )
5 $\triangleright$ FamiliesOfGeneralMappingsAndRanges( Fam )
6 $\triangleright$ GeneralMappingsFamily( sourcefam, rangefam )
7 $\triangleright$ TypeOfDefaultGeneralMapping( source, range, filter )

is the elements family of the family of the range of each general mapping in the family Fam.

is the elements family of the family of the source of each general mapping in the family Fam.

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.

All general mappings with same source family FS and same range family FR lie in the family GeneralMappingsFamily( FS, FR ).

is the type of mappings with IsDefaultGeneralMappingRep with source source and range range and additional categories filter.

Methods for the operations ImagesElm, ImagesRepresentative, ImagesSet, ImageElm, PreImagesElm, PreImagesRepresentative, PreImagesSet, and PreImageElm 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, Images, PreImage, and PreImages, which then delegate to the operations listed above.
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 a tuple of $R$.

In GAP, a relation is conceptually represented as a general mapping from $X$ to itself. The category `IsBinaryRelation` is the same as the category `IsEndoGeneralMapping` (see 31.12.3). 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` (see 32.4.5), or enumerating the tuples of the relation.

### 32.1 General Binary Relations

1. `IsBinaryRelation( R )`
   
is exactly the same category as (i.e. a synonym for) `IsEndoGeneralMapping` (see 31.12.3).
   
   We have the following general constructors.

2. `BinaryRelationByElements( domain, elms )`
   
is the binary relation on `domain` and with underlying relation consisting of the tuples collection `elms`. This construction is similar to `GeneralMappingByElements` (see 31.1.1) where the source and range are the same set.

3. `IdentityBinaryRelation( degree )`

4. `EmptyBinaryRelation( degree )`

### 32.2 Properties and Attributes of Binary Relations

1. `IsReflexiveBinaryRelation( R )`
   
   returns `true` if the binary relation $R$ is reflexive, and `false` otherwise.
   
   A binary relation $R$ (as tuples) 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`; see 31.2.1).
IsSymmetricBinaryRelation( R ) returns true if the binary relation R is symmetric, and false otherwise.

A binary relation R (as tuples) on a set X is symmetric if (x, y) ∈ R then (y, x) ∈ R. Alternatively, R as a mapping is symmetric if for all x ∈ X, the preimage set of x under R equals the image set R(x).

IsTransitiveBinaryRelation( R ) returns true if the binary relation R is transitive, and false otherwise.

A binary relation R (as tuples) on a set X is transitive if (x, y), (y, z) ∈ R then (x, z) ∈ R. Alternatively, R as a mapping is transitive if for all x ∈ X, the image set R(R(x)) of the image set R(x) of x is a subset of R(x).

IsAntisymmetricBinaryRelation( rel ) returns true if the binary relation rel is antisymmetric, and false otherwise.

A binary relation R (as tuples) on a set X is antisymmetric if (x, y), (y, x) ∈ R implies x = y. Alternatively, R as a mapping is antisymmetric if for all x ∈ X, the intersection of the preimage set of x under R and the image set R(x) is \{x\}.

IsPreOrderBinaryRelation( rel ) 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.

IsPartialOrderBinaryRelation( rel ) returns true if the binary relation rel is a partial order, and false otherwise.

A partial order is a preorder which is also antisymmetric.

IsHasseDiagram( rel ) returns true if the binary relation rel is a Hasse Diagram of a partial order, i.e. was computed via Hasse-DiagramBinaryRelation (see 32.4.4).

IsEquivalenceRelation( R ) returns true if the binary relation R is an equivalence relation, and false otherwise.

Recall, that a relation R on the set X is an equivalence relation if it is symmetric, transitive, and reflexive.

Successors( R ) returns the list of images of a binary relation R. If the underlying domain of the relation is not \([1..n]\) 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.

DegreeOfBinaryRelation( R ) returns the size of the underlying domain of the binary relation R. This is most natural when working with a binary relation on points.

PartialOrderOfHasseDiagram( HD ) is the partial order associated with the Hasse Diagram HD i.e. the partial order generated by the reflexive and transitive closure of HD.
32.3 Binary Relations on Points

We have special construction methods when the underlying $X$ of our relation is the set of integers $\{1, \ldots, n\}$.

1. $\text{BinaryRelationOnPoints}(\ list\ )$  
   $\text{BinaryRelationOnPointsNC}(\ list\ )$

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.

2. $\text{RandomBinaryRelationOnPoints}(\ degree\ )$

creates a relation on points with degree $degree$.

3. $\text{AsBinaryRelationOnPoints}(\ trans\ )$  
   $\text{AsBinaryRelationOnPoints}(\ perm\ )$  
   $\text{AsBinaryRelationOnPoints}(\ rel\ )$

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.

32.4 Closure Operations and Other Constructors

1. $\text{ReflexiveClosureBinaryRelation}(\ R\ )$

is the smallest binary relation containing the binary relation $R$ which is reflexive. This closure inherents the properties symmetric and transitive from $R$. E.g. if $R$ is symmetric then its reflexive closure is also.

2. $\text{SymmetricClosureBinaryRelation}(\ R\ )$

is the smallest binary relation containing the binary relation $R$ which is symmetric. This closure inherents the properties reflexive and transitive from $R$. E.g. if $R$ is reflexive then its symmetric closure is also.

3. $\text{TransitiveClosureBinaryRelation}(\ rel\ )$

is the smallest binary relation containing the binary relation $R$ which is transitive. This closure inerents the properties reflexive and symmetric from $R$. E.g. if $R$ is symmetric then its transitive closure is also.

$\text{TransitiveClosureBinaryRelation}$ 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.

4. $\text{HasseDiagramBinaryRelation}(\ partial-order\ )$

is the smallest relation contained in the partial order $partial-order$ whose reflexive and transitive closure is equal to $partial-order$.

5. $\text{StronglyConnectedComponents}(\ R\ )$

returns an equivalence relation on the vertices of the binary relation $R$.

6. $\text{PartialOrderByOrderingFunction}(\ dom,\ orderfunc\ )$

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.
32.5 Equivalence Relations

An equivalence relation $E$ over the set $X$ is a relation on $X$ which is reflexive, symmetric, and transitive, of the set $X$. 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 $\text{EquivalenceRelationPartition}$ which provides the partition induced by the given equivalence.

1 $\triangleright$ $\text{EquivalenceRelationByPartition( domain, list )}$
2 $\triangleright$ $\text{EquivalenceRelationByPartitionNC( domain, list )}$

constructs the equivalence relation over the set $\text{domain}$ which induces the partition represented by $\text{list}$. This representation includes only the non-trivial blocks (or equivalent classes). $\text{list}$ is a list of lists, each of these lists contain elements of $\text{domain}$ 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 $\text{EquivalenceRelationPartition}$). a list of elements of $\text{domain}$ The partition $\text{list}$ is a list of lists, each of these is a list of elements of $\text{domain}$ that makes up a block (or equivalent class). The $\text{domain}$ is the domain over which the relation is defined, and $\text{list}$ is a list of lists, each of these is a list of elements of $\text{domain}$ which are related to each other. $\text{list}$ 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.

2 $\triangleright$ $\text{EquivalenceRelationByRelation( rel )}$

returns the smallest equivalence relation containing the binary relation $\text{rel}$.

3 $\triangleright$ $\text{EquivalenceRelationByPairs( D, elms )}$
4 $\triangleright$ $\text{EquivalenceRelationByPairsNC( D, elms )}$

return the smallest equivalence relation on the domain $D$ such that every pair in $\text{elms}$ is in the relation.

In the second form, it is not checked that $\text{elms}$ are in the domain $D$.

4 $\triangleright$ $\text{EquivalenceRelationByProperty( domain, property )}$

creates an equivalence relation on $\text{domain}$ whose only defining datum is that of having the property $\text{property}$.

32.6 Attributes of and Operations on Equivalence Relations

1 $\triangleright$ $\text{EquivalenceRelationPartition( equiv )}$

returns a list of lists of elements of the underlying set of the equivalence relation $\text{equiv}$. The lists are precisely the nonsingleton equivalence classes of the equivalence. This allows us to describe “small” equivalences on infinite sets.

2 $\triangleright$ $\text{GeneratorsOfEquivalenceRelationPartition( equiv )}$

is a set of generating pairs for the equivalence relation $\text{equiv}$. This set is not unique. The equivalence $\text{equiv}$ is the smallest equivalence relation over the underlying set $X$ which contains the generating pairs.

3 $\triangleright$ $\text{JoinEquivalenceRelations( equiv1, equiv2 )}$
4 $\triangleright$ $\text{MeetEquivalenceRelations( equiv1, equiv2 )}$

$\text{JoinEquivalenceRelations( equiv1, equiv2 )}$ returns the smallest equivalence relation containing both the equivalence relations $\text{equiv1}$ and $\text{equiv2}$.

$\text{MeetEquivalenceRelations( equiv1, equiv2 )}$ returns the intersection of the two equivalence relations $\text{equiv1}$ and $\text{equiv2}$.
32.7 Equivalence Classes

1. IsEquivalenceClass( O )
   returns true if the object O 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 object and not just a list of elements.

2. EquivalenceClassRelation( C )
   returns the equivalence relation of which C is a class.

3. EquivalenceClasses( rel )
   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).

4. EquivalenceClassOfElement( rel, elt )
   EquivalenceClassOfElementNC( rel, elt )
   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 second form, it is not checked that elt is in the domain over which rel is defined.
This chapter deals with domains (see 30) that are closed under multiplication \(*\). Following [Bou70], we call them **magmas** in GAP. Together with the domains closed under addition \(+\), (see 53), they are the basic algebraic structures; every semigroup (see 49), monoid (see 50), group (see 37), ring (see 54), or field (see 56) is a magma. In the cases of a **magma-with-one** or **magma-with-inverses**, additional multiplicative structure is present, see 33.1. For functions to create free magmas, see 34.4.

### 33.1 Magma Categories

1. **IsMagma( obj )**

   A **magma** in GAP is a domain \( M \) with (not necessarily associative) multiplication \(*\): \( M \times M \rightarrow M \).

2. **IsMagmaWithOne( obj )**

   A **magma-with-one** in GAP is a magma \( M \) with an operation \(^0\) (or \( \text{One} \)) that yields the identity of \( M \).
   
   So a magma-with-one \( M \) does always contain a unique multiplicatively neutral element \( e \), i.e., \( e \ast m = m = m \ast e \) holds for all \( m \in M \) (see 33.4.10). This element \( e \) can be computed with the operation \( \text{One} \) (see 30.10.2) as \( \text{One}( M ) \), and \( e \) is also equal to \( \text{One}( \text{elm} ) \) and to \( \text{elm}^0 \) for each element \( \text{elm} \) in \( M \).
   
   **Note** that a magma may contain a multiplicatively neutral element but not an identity (see 30.10.2), and a magma containing an identity may **not** lie in the category \( \text{IsMagmaWithOne} \) (see 30.6).

3. **IsMagmaWithInversesIfNonzero( obj )**

   An object in this GAP category is a magma-with-one \( M \) with an operation \(^-1\): \( M \setminus Z \rightarrow M \setminus Z \) that maps each element \( m \) of \( M \setminus Z \) to its inverse \( m^{-1} \) (or \( \text{Inverse}( m ) \), see 30.10.8), 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 \( \text{IsMagmaWithInversesIfNonzero} \) will usually have both a multiplicative and an additive structure (see 53), and the set \( Z \), if it is nonempty, contains exactly the zero element (see 30.10.3) of \( M \).

4. **IsMagmaWithInverses( obj )**

   A **magma-with-inverses** in GAP is a magma-with-one \( M \) with an operation \(^-1\): \( M \rightarrow M \) that maps each element \( m \) of \( M \) to its inverse \( m^{-1} \) (or \( \text{Inverse}( m ) \), see 30.10.8).

   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.
33.2 Magma Generation

1. \( \text{Magma}( \text{gens} ) \)
2. \( \text{Magma}( \text{Fam}, \text{gens} ) \)

returns the magma \( M \) that is generated by the elements in the list \( \text{gens} \), that is, the closure of \( \text{gens} \) under multiplication \( * \). The family \( \text{Fam} \) of \( M \) can be entered as first argument; this is obligatory if \( \text{gens} \) is empty (and hence also \( M \) is empty).

3. \( \text{MagmaWithOne}( \text{gens} ) \)
4. \( \text{MagmaWithOne}( \text{Fam}, \text{gens} ) \)

returns the magma-with-one \( M \) that is generated by the elements in the list \( \text{gens} \), that is, the closure of \( \text{gens} \) under multiplication \( * \) and \( \text{One} \). The family \( \text{Fam} \) of \( M \) can be entered as first argument; this is obligatory if \( \text{gens} \) is empty (and hence \( M \) is trivial).

5. \( \text{MagmaWithInverses}( \text{gens} ) \)
6. \( \text{MagmaWithInverses}( \text{Fam}, \text{gens} ) \)

returns the magma-with-inverses \( M \) that is generated by the elements in the list \( \text{gens} \), that is, the closure of \( \text{gens} \) under multiplication \( * \), \( \text{One} \), and \( \text{Inverse} \). The family \( \text{Fam} \) of \( M \) can be entered as first argument; this is obligatory if \( \text{gens} \) is empty (and hence \( M \) is trivial).

The underlying operations for which methods can be installed are the following.

7. \( \text{MagmaByGenerators}( \text{gens} ) \)
8. \( \text{MagmaByGenerators}( \text{Fam}, \text{gens} ) \)

\( \text{MagmaByGenerators} \) returns the magma-with-one generated by the elements in the list \( \text{gens} \), with parent the domain \( D \). \( \text{SubmagmaWithOne} \) does the same, except that it is not checked whether the elements of \( \text{gens} \) lie in \( D \).

9. \( \text{SubmagmaWithInverses}( \text{D}, \text{gens} ) \)
10. \( \text{AsMagma}( \text{C} ) \)

returns the magma-with-inverses generated by the elements in the list \( \text{gens} \), with parent the domain \( D \). \( \text{SubmagmaWithInversesNC} \) does the same, except that it is not checked whether the elements of \( \text{gens} \) lie in \( D \).

The following functions can be used to regard a collection as a magma.

For a collection \( C \) whose elements form a magma, \( \text{AsMagma} \) returns this magma. Otherwise \( \text{fail} \) is returned.
Chapter 33. Magmas

11 ▶ AsSubmagma( D, C )

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.

The following function creates a new magma which is the original magma with a zero adjoined.

12 ▶ InjectionZeroMagma( M )

The canonical homomorphism i from the magma M into the magma formed from M with a single new element which is a multiplicative zero for the resulting magma.

The elements of the new magma form a family of elements in the category IsMultiplicativeElementWithZero, and the new magma is obtained as Range(i).

33.3 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.

1 ▶ MagmaByMultiplicationTable( A )

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 \( m_i * m_j = A[i][j] \).

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 33.3.4.

2 ▶ MagmaWithOneByMultiplicationTable( A )

The only differences between MagmaByMultiplicationTable and MagmaWithOneByMultiplicationTable are that the latter returns a magma-with-one (see 33.2.2) if the magma described by the matrix A has an identity, and returns fail if not.

3 ▶ MagmaWithInversesByMultiplicationTable( A )

MagmaByMultiplicationTable and MagmaWithInversesByMultiplicationTable differ only in that the latter returns magma-with-inverses (see 33.2.3) if each element in the magma described by the matrix A has an inverse, and returns fail if not.

4 ▶ MagmaElement( M, i )

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.

5 ▶ MultiplicationTable( elms )

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.
Section 4. Attributes and Properties for Magmas

```gap
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
<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
```

33.4 Attributes and Properties for Magmas

1. **GeneratorsOfMagma( M )**

   A

   is a list gens of elements of the magma M that generates M as a magma, that is, the closure of gens under multiplication is M.

2. **GeneratorsOfMagmaWithOne( M )**

   A

   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 and One (see 30.10.2) is M.

3. **GeneratorsOfMagmaWithInverses( M )**

   A

   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 and taking inverses (see 30.10.8) is M.

4. **Centralizer( M, elm )**

   O

   **Centralizer( M, S )**

   O

   **Centralizer( class )**

   O

For an element elm of the magma M this operation returns the centralizer of elm. This is the domain of those elements m ∈ 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 (see 37.9.1), Centralizer returns the centralizer of Representative class (which is a slight abuse of the notation).
Chapter 33. Magmas

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(())

Centre\(\ M\) \hspace{1cm} \text{centre of the magma } \ M, \text{ i.e., the domain of those elements } \ m \in \ M \text{ that commute with all elements of } \ M. \text{ Center is just a synonym for Centre.}

We have Centre\(\ M\) = Centralizer\(\ M, \ M\), see 33.4.4.
The centre of a magma is always commutative (see 33.4.9). (When one installs a new method for Centre, one should set the IsCommutative value of the result to true, in order to make this information available.)

Idempotents\(\ M\) \hspace{1cm} \text{the set of elements of } \ M \text{ which are their own squares.}

IsAssociative\(\ M\) \hspace{1cm} \text{a magma } \ M \text{ is associative if for all elements } a, b, c \in \ M \text{ the equality } (a \ast b) \ast c = a \ast (b \ast c) \text{ holds.}

An associative magma is called a semigroup (see 49), an associative magma-with-one is called a monoid (see 50), and an associative magma-with-inverses is called a group (see 37).

IsCentral\(\ M, obj\) \hspace{1cm} \text{IsCentral returns true if the object } obj, \text{ which must either be an element or a magma, commutes with all elements in the magma } \ M.

IsCommutative\(\ M\) \hspace{1cm} \text{a magma } \ M \text{ is commutative if for all elements } a, b \in \ M \text{ the equality } a \ast b = b \ast a \text{ holds. IsAbelian is a synonym of IsCommutative.}

Note that the commutativity of the addition + in an additive structure can be tested with IsAdditivelyCommutative, see 53.3.1.

MultiplicativeNeutralElement\(\ M\) \hspace{1cm} \text{returns the element } e \text{ in the magma } \ M \text{ with the property that } e \ast m = m = m \ast e \text{ holds for all } m \in \ M, \text{ 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; \text{ in this case, } e \text{ cannot be obtained as One}\(\ M\), see 30.10.2.

MultiplicativeZero\(\ M\) \hspace{1cm} \text{Returns the multiplicative zero of the magma which is the element } z \text{ such that for all } m \in \ M, z \ast m = m \ast z = z.

IsMultiplicativeZero\(\ M, z\) \hspace{1cm} \text{returns true iff } z \ast m = m \ast z = z \text{ for all } m \in \ M.
13 ▶ SquareRoots( M, elm )

is the proper set of all elements \( r \) in the magma \( M \) such that \( r \ast r = elm \) holds.

14 ▶ TrivialSubmagmaWithOne( M )

is the magma-with-one that has the identity of the magma-with-one \( M \) as only element.

Note that IsAssociative and IsCommutative 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 54), then the addition \(+\) is always assumed to be associative and commutative, see 30.12.
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 35.

34.1 Categories of Words and Nonassociative Words

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 33.4.7), which are described in Chapter 35, and also the case of free magmas, where the multiplication is nonassociative (see 34.1.3).

Elements in free magmas (see 34.4.1) lie in the category IsWord; similarly, elements in free magmas-with-one (see 34.4.2) lie in the category IsWordWithOne, and so on.

IsWord is mainly a “common roof” for the two disjoint categories IsAssocWord (see 35.1.1) and IsNonassocWord (see 34.1.3) 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 34.5 and 35.7).

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 45 and 44.

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 34.4.1, 35.2.1).

Words are also used to implement free algebras, in the same way as group elements are used to implement group algebras (see 60.2 and Chapter 63).

```gap
gap> m:= FreeMagmaWithOne( 2 );; gens:= GeneratorsOfMagmaWithOne( m );
[ x1, x2 ]
( (x1*x2)*x1)
( 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 ]
m1*m2*m1
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

2 ▶ IsWordCollection( obj )

IsWordCollection is the collections category (see 28.1.4) of IsWord.

gap> IsWordCollection( m ); IsWordCollection( s );
true
true
gap> IsWordCollection( [ "a", "b" ] );
false

3 ▶ IsNonassocWord( obj )
▶ IsNonassocWordWithOne( obj )

A nonassociative word in GAP is an element in a free magma or a free magma-with-one (see 34.4).
The default methods for ViewObj and PrintObj (see 6.2) 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 asso-
ciative words can be interpreted as such programs, except that the exact succession of multiplications is not
prescribed due to the associativity.) The function MappedWord (see 34.3.1) implements a way to apply such
a program. A more general way is provided by straight line programs (see 35.8).

Note that associative words (see Chapter 35) are not regarded as special cases of nonassociative words
(see 34.1.1).

4 ▶ IsNonassocWordCollection( obj )
▶ IsNonassocWordWithOneCollection( obj )

IsNonassocWordCollection is the collections category (see 28.1.4) of IsNonassocWord, and IsNonassoc-
WordWithOneCollection is the collections category of IsNonassocWordWithOne.
34.2 Comparison of Words

1 ▶ \texttt{w1} = \texttt{w2}

Two words are equal if and only if they are words over the same alphabet and with equal external representations (see 34.5 and 35.7). For nonassociative words, the latter means that the words arise from the letters of the alphabet by the same sequence of multiplications.

2 ▶ \texttt{w1} < \texttt{w2}

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 34.5; associative words are ordered by the shortlex ordering via \texttt{<} (see 35.7).

Note that the alphabet of a word is determined by its family (see 13.1), and that the result of each call to \texttt{FreeMagma}, \texttt{FreeGroup} etc. consists of words over a new alphabet. In particular, there is no “universal” empty word, every families of words in \texttt{IsWordWithOne} has its own empty word.

```gap
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*(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
```

34.3 Operations for Words

Two words can be multiplied via \texttt{*} only if they are words over the same alphabet (see 34.2).

1 ▶ \texttt{MappedWord( w, gens, imgs )}

\texttt{MappedWord} returns the object that is obtained by replacing each occurrence in the word \texttt{w} of a generator in the list \texttt{gens} by the corresponding object in the list \texttt{imgs}. The lists \texttt{gens} and \texttt{imgs} must of course have the same length.

\texttt{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 \texttt{imgs} are all \texttt{associative words} (see Chapter 35) in the same family as the elements in \texttt{gens}, and some of them are equal to the corresponding generators in \texttt{gens}, then those may be omitted from \texttt{gens} and \texttt{imgs}. In this situation, the special case that the lists \texttt{gens} and \texttt{imgs} have only length 1 is handled more efficiently by \texttt{EliminatedWord} (see 35.4.6).)

```gap
```
Section 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 \texttt{GeneratorsOfMagma} in the case of a free magma, and via \texttt{GeneratorsOfMagmaWithOne} in the case of a free magma-with-one.

34.4 Free Magmas

Called in the first form, \texttt{FreeMagma} returns a free magma on \texttt{rank} generators. Called in the second form, \texttt{FreeMagma} returns a free magma on \texttt{rank} generators, printed as \texttt{name}1, \texttt{name}2 etc., that is, each name is the concatenation of the string \texttt{name} and an integer from 1 to \texttt{range}. Called in the third form, \texttt{FreeMagma} returns a free magma on as many generators as arguments, printed as \texttt{name}1, \texttt{name}2 etc. Called in the fourth form, \texttt{FreeMagma} returns a free magma on as many generators as the length of the list \texttt{names}, the \texttt{i}-th generator being printed as \texttt{names}\texttt{[i]}. Called in the fifth form, \texttt{FreeMagma} returns a free magma on infinitely many generators, where the first generators are printed by the names in the list \texttt{init}, and the other generators by \texttt{name} and an appended number.

Called in the first form, \texttt{FreeMagmaWithOne} returns a free magma-with-one on \texttt{rank} generators. Called in the second form, \texttt{FreeMagmaWithOne} returns a free magma-with-one on \texttt{rank} generators, printed as \texttt{name}1, \texttt{name}2 etc. Called in the third form, \texttt{FreeMagmaWithOne} returns a free magma-with-one on as many generators as arguments, printed as \texttt{name}1, \texttt{name}2 etc. Called in the fourth form, \texttt{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 \( \text{names}[i] \). Called in the fifth form, \texttt{FreeMagmaWithOne} returns a free magma on infinitely many generators, where the first generators are printed by the names in the list \textit{init}, and the other generators by \textit{name} and an appended number.

```gap
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.

```gap
gap> m:= FreeMagma( "x", "x" ); gens:= GeneratorsOfMagma( m );;
<free magma on the generators [ x, x ]>  
gap> gens[1] = gens[2];
false
```

### 34.5 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 \( v \ast w \) 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 \texttt{ObjByExtRep}.

```gap
gap> m:= FreeMagma( 2 );; gens:= GeneratorsOfMagma( m );
[ x1, x2 ]
((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))
```
35

Associative Words

35.1 Categories of Associative Words

Associative words are used to represent elements in free groups, semigroups and monoids in GAP (see 35.2). 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 35.4).

\begin{verbatim}
1 ▶ IsAssocWord( obj )
2 ▶ IsAssocWordWithOne( obj )
3 ▶ IsAssocWordWithInverse( obj )
\end{verbatim}

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 to compute an identity), IsAssocWordWithInverse is the category of associative words in free groups (which have an inverse). See 34.1.1 for more general categories of words.

Different alphabets correspond to different families of associative words. There is no relation whatsoever between words in different families.

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
gap> a := f.1;; b := f.2;; c := f.3;;
\end{verbatim}

(Interactively, the function AssignGeneratorVariables (see 35.2.5) provides a shorthand for this.) This allows us to define w more conveniently:

\begin{verbatim}
gap> w := a*b/c*b*a/a*c/b;
a*b*c^-1*b*c*b^-1
\end{verbatim}

Using homomorphisms it is possible to express elements of a group as words in terms of generators, see 37.5.
35.2 Free Groups, Monoids and Semigroups

Usually a family of associative words will be generated by constructing the free object generated by them.

Called in the first form, FreeGroup returns a free group on rank generators. Called in the second form, FreeGroup returns a free group on rank generators, printed as name1, name2 etc. Called in the third form, FreeGroup returns a free group on as many generators as arguments, printed as name1, name2 etc. Called in the fourth 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 fifth 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. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

Any group consisting of elements in IsAssocWordWithInverse lies in the filter IsFreeGroup; this holds in particular for any group created with FreeGroup (see 35.2.1), or any subgroup of such a group.

Also see Chapter 45.

Called in the first form, FreeMonoid returns a free monoid on rank generators. Called in the second form, FreeMonoid returns a free monoid on rank generators, printed as name1, name2 etc., that is, each name is the concatenation of the string name and an integer from 1 to range. Called in the third form, FreeMonoid returns a free monoid on as many generators as arguments, printed as name1, name2 etc. 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 or IsLetterWordsFamily or IsWLetterWordsFamily or IsBLetterWordsFamily. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

Also see Chapter 50.

Called in the first form, FreeSemigroup returns a free semigroup on rank generators. Called in the second form, FreeSemigroup returns a free semigroup on rank generators, printed as name1, name2 etc., that is,
Section 2. Free Groups, Monoids and Semigroups

Each name is the concatenation of the string name and an integer from 1 to range. Called in the third form, FreeSemigroup returns a free semigroup on as many generators as arguments, printed as name1, name2 etc. Called in the fourth 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 fifth 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 or IsLetterWordsFamily or IsWLetterWordsFamily or IsBLetterWordsFamily. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

Also see Chapter 49 and 49.

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.

```gap
gap> FreeGroup( 5 );<free group on the generators [ f1, f2, f3, f4, f5 ]>
```

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.

```gap
gap> f:= FreeGroup( "x", "x" ); gens:= GeneratorsOfGroup( f );;
<free group on the generators [ x, x ]>
gap> gens[1] = gens[2]; false
```

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 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

```gap
gap> var1:=GeneratorsOfGroup(G)[1];
gap> var2:=GeneratorsOfGroup(G)[2];
...
```
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.

35.3 Comparison of Associative Words

1 ▶ \( w_1 = w_2 \)

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 35.7 and 34.2.

There is no “universal” empty word, every alphabet (that is, every family of words) has its own empty word.

\[
\text{gap> } f := \text{FreeGroup( "a", "b", "b" )};;
\text{gap> } \text{gens} := \text{GeneratorsOfGroup}(f);
\text{[ a, b, b ]}
\text{gap> } \text{gens}[2] = \text{gens}[3];
\text{false}
\text{gap> } x := \text{gens}[1]*\text{gens}[2];
a*b
\text{gap> } y := \text{gens}[2]/\text{gens}[2]*\text{gens}[1]*\text{gens}[2];
a*b
\text{gap> } x = y;
\text{true}
\text{gap> } z := \text{gens}[2]/\text{gens}[2]*\text{gens}[1]*\text{gens}[3];
a*b
\text{gap> } x = z;
\text{false}
\]

2 ▶ \( w_1 < w_2 \)

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 \( g \).

\[
\text{gap> } f := \text{FreeGroup( 2 )};;
\text{gap> } \text{gens} := \text{GeneratorsOfGroup}( f );;
\text{gap> } a := \text{gens}[1];; \text{b} := \text{gens}[2];;
\text{gap> } \text{One}(f) < a^-1; \text{a}^-1 < a; \text{a} < b^-1; \text{b}^-1 < b; \text{b} < \text{a}^2; \text{a}^2 < \text{a*b};
\text{true}
\text{true}
\text{true}
\text{true}
\text{true}
\text{true}
\]

3 ▶ \textbf{IsShortLexLessThanOrEqual}( u, v )

returns \textbf{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.)

4 ▶ \textbf{IsBasicWreathLessThanOrEqual}( u, v )

returns \textbf{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.)
35.4 Operations for Associative Words

The product of two given associative words is defined as the freely reduced concatenation of the words; so adjacent pairs of a generator and its inverse never occur in words. Besides the multiplication *, the arithmetical operators One (if the word lies in a family with identity) and (if the generators are invertible) Inverse, /, Comm, and LeftQuotient are applicable to associative words (see 30.12).

For the operation MappedWord, which is applicable to arbitrary words, see 34.3.1.

There are two internal representations of associative words: By letters and by syllables (see 35.6). Unless something else is specified, words are stored in the letter representation. Note, however, 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.

1 ▶ Length( w )

For an associative word w, Length returns the number of letters in w.

    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

2 ▶ ExponentSumWord( w, gen )

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.

    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

3 ▶ Subword( w, from, to )

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.

    gap> w; Subword( w, 3, 7 );
a^5*b*a^-2*b^-4*a
a^-3*b*a

4 ▶ PositionWord( w, sub, from )

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 35.4.3.
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, from-1 ), by, and Subword( w, to+1, Length( w ) ), see 35.4.3.

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.

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.

35.5 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 \( x_i^{m_i} \) are uniquely determined; these powers of generators are called the syllables of w.

NumberSyllables returns the number of syllables of the associative word w.
Section 6. Representations for Associative Words

2> ExponentSyllable( w, i )

ExponentSyllable returns the exponent of the i-th syllable of the associative word w.

3> GeneratorSyllable( w, i )

GeneratorSyllable returns the number of the generator that is involved in the i-th syllable of the associative word w.

4> SubSyllables( w, from, to )

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.

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

There are two internal representations of associative words: By letters and by syllables (see 35.6). Unless something else is specified, words are stored in the letter representation. Note, however, 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.

35.6 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 absed (for example GeneratorSyllable and Subword) perform substantially better in the corresponding representation. For example when creating pc groups (see 44), it is advantageous to use a syllable representation while calculations in free groups usually benefit from using a letter representation.

1> IsLetterAssocWordRep( obj )

A word in letter representation stores a list of generator/inverses numbers (as given by LetterRepAssocWord). Letter access is fast, syllable access is slow for such words.

2> IsLetterWordsFamily( obj )

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).
3. IsBLetterAssocWordRep( obj )
   IsWLetterAssocWordRep( obj )

   These two subrepresentations of IsLetterAssocWordRep indicate whether the word is stored as a list of bytes (in a string) or as a list of integers.

4. IsBLetterWordsFamily( obj )
   IsWLetterWordsFamily( obj )

   These two subcategories of IsLetterWordsFamily specify the type of letter representation to be used.

5. IsSyllableAssocWordRep( obj )

   A word in syllable representation stores generator/exponents pairs (as given by ExtRepOfObj). Syllable access is fast, letter access is slow for such words.

6. IsSyllableWordsFamily( obj )

   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.

7. Is8BitsFamily( obj )
   Is16BitsFamily( obj )
   Is32BitsFamily( obj )
   IsInfBitsFamily( obj )

   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:

8. LetterRepAssocWord( w )
   LetterRepAssocWord( w, gens )

   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 second variant, 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.)

9. AssocWordByLetterRep( Fam, lrep [, gens] )

   takes a letter representation lrep (see LetterRepAssocWord, section 35.6.8) 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.

   gap> w:=AssocWordByLetterRep( FamilyObj(a), [-1,2,1,-2,-2,1,1,1,1,1,1,1,1,1,1,1,1,1] );
   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 35.7) can be used if a syllable representation is needed.
35.7 The External Representation for Associative Words

The external representation of the associative word $w$ is defined as follows. If $w = g_1^{e_1} \cdot g_2^{e_2} \cdot \ldots \cdot g_n^{e_n}$ 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, $g_1 \cdot g_2 \cdot g_2^{-1} \cdot g_1$ has the external representation $[1, 2]$. Regardless of the family preference for letter or syllable representations (see 35.6), ExtRepOfObj and ObjByExtRep can be used and interface to this “syllable”-like representation.

```
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 ]
```

35.8 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 (see 35.3.1). 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 $cd$; Alternatively, one can compute $c = ab, e = bc$, and $ace$. 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 (see 35.8.1) 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 35.7): 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.

Straight line programs can be constructed using StraightLineProgram (see 35.8.2).

Defining attributes for straight line programs are NrInputsOfStraightLineProgram (see 35.8.4) and LinesOfStraightLineProgram (see 35.8.3). Another operation for straight line programs is ResultOfStraightLineProgram (see 35.8.5).

Special methods applicable to straight line programs are installed for the operations Display, IsInternallyConsistent, PrintObj, and ViewObj.

For a straight line program $prog$, the default Display 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; in this case these names are used, the default for gensnames is $[g_1, g_2, \ldots]$, the default for listname is $r$.

```
1 ▶ IsStraightLineProgram( obj )

Each straight line program in GAP lies in the category IsStraightLineProgram.

2 ▶ StraightLineProgram( lines[, nrgens] )
etectionNC( lines[, nrgens] )

In the first form, $lines$ must be a list of lists that defines a unique straight line program (see 35.8.1); 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**, see 26.3.4) such that no entry is an initial part of another one. Called with this input, **StraightLineProgramNC** returns a straight line program that evaluates to the word corresponding to **string** when called with generators corresponding to **gens**.

**StraightLineProgramNC** does the same as **StraightLineProgram**, except that the internal consistency of the program is not checked.

3▶ **LinesOfStraightLineProgram**(*prog*)

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.

4▶ **NrInputsOfStraightLineProgram**(*prog*)

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 then the number of generators is not determined by the lines, and must be set in the construction of the straight line program (see 35.8.2). So if **prog** contains a line of form 1. other than the last and does not store the number of generators then **NrInputsOfStraightLineProgram** signals an error.

5▶ **ResultOfStraightLineProgram**(*prog*, **gens**)

**ResultOfStraightLineProgram** evaluates the straight line program (see 35.8.1) **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).

Here are some examples.

```gap
f := FreeGroup( "x", "y" );; gens := GeneratorsOfGroup( f );;
Reshaped := [ gens[1], gens[2] ];
prg := StraightLineProgram( [ [] ] );
ResultOfStraightLineProgram( prg, [] );
```

The above straight line program **prg** returns – for any list of input generators – an empty list.
Section 8. Straight Line Programs

```
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[4]:= r[3]^-1;
# return value:
r[4]
```

```
gap> IsInternallyConsistent( StraightLineProgramNC( [ [1,2] ] ) );
true
```

```
gap> prg1:= StraightLineProgram( [ [1,1,2,2], [3,3,1,1] ], 2 );;
gap> prg2:= StraightLineProgram( [ [1,1,2,2], [2,3,1,1] ] );;
gap> res1:= ResultOfStraightLineProgram( prg1, gens );
x*y^2*x*y^2*x*y^2*x
```

```
gap> res1 = (x*y^2)^3*x;
true
```

```
gap> res2:= ResultOfStraightLineProgram( prg2, gens );
x*y^2*x*y^2*x*y^2*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 ]
```

```
StringOfResultOfStraightLineProgram( prog, gensnames[, "LaTeX"] )
```

StringOfResultOfStraightLineProgram returns a string that describes the result of the straight line program (see 35.8.1) 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 [, ].
Chapter 35. Associative Words

For two straight line programs \( \text{prog1} \) and \( \text{prog2} \), \( \text{CompositionOfStraightLinePrograms} \) returns a straight line program \( \text{prog} \) with the properties that \( \text{prog1} \) and \( \text{prog} \) have the same number of inputs, and the result of \( \text{prog} \) when applied to given generators \( \text{gens} \) equals the result of \( \text{prog2} \) when this is applied to the output of \( \text{prog1} \) applied to \( \text{gens} \).

(Of course the number of outputs of \( \text{prog1} \) must be the same as the number of inputs of \( \text{prog2} \).)

\[
\text{CompositionOfStraightLinePrograms}( \text{prog2}, \text{prog1} )
\]

For a nonempty dense list \( \text{listofprogs} \) of straight line programs that have the same number \( n \), say, of inputs (see 35.8.4) and for which the results (see 35.8.5) are single elements (i.e., not lists of elements), \( \text{IntegratedStraightLineProgram} \) returns a straight line program \( \text{prog} \) with \( n \) inputs such that for each \( n \)-tuple \( \text{gens} \) of generators, \( \text{ResultOfStraightLineProgram}( \text{prog}, \text{gens} ) \) is equal to the list \( \text{List}( \text{listofprogs}, p \rightarrow \text{ResultOfStraightLineProgram}( \text{p}, \text{gens} ) ) \).

\[
\text{IntegratedStraightLineProgram}( \text{listofprogs} )
\]
35.9 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.

1. \texttt{IsStraightLineProgElm(obj)}
   
   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 \texttt{seeds}, the second component the straight line program. We need to rank higher than default methods.

2. \texttt{StraightLineProgElm(seed, prog)}
   
   Creates a straight line program element for seed \texttt{seed} and program \texttt{prog}.

3. \texttt{StraightLineProgGens(gens[, base])}
   
   Returns a set of straight line program elements corresponding to the generators in \texttt{gens}. If \texttt{gens} is a set of permutations then \texttt{base} can be given which must be a base for the group generated by \texttt{gens}. (Such a base will be used to speed up equality tests.)

4. \texttt{EvalStraightLineProgElm(slpel)}
   
   Evaluates a straight line program element \texttt{slpel} from its seeds.

5. \texttt{StretchImportantSLPElement(elm)}
   
   If \texttt{elm} is a straight line program element whose straight line representation is very long, this operation changes the representation of \texttt{elm} to a straight line program element, equal to \texttt{elm}, whose seed contains the evaluation of \texttt{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.

   \begin{verbatim}
   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, 3, -2, 1, -1, 2, -2, 1, 1, 1 ] ]
   | (1,4,3,2)>
   \end{verbatim}

   See also Section 41.12.
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.

### 36.1 Operations on rewriting systems

1. `IsRewritingSystem( obj )`
   
   This is the category in which all rewriting systems lie.

2. `Rules( rws )`
   
   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.

3. `OrderOfRewritingSystem( rws )`
   
   `OrderingOfRewritingSystem( rws )`  
   
   return the ordering of the rewriting system $rws$.

4. `ReducedForm( rws, u )`
   
   Given an element $u$ in the free (or term) algebra 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 $T$.

5. `IsConfluent( rws )`
   
   `IsConfluent( A )`
   
   return `true` if and only if the rewriting system $rws$ 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 \( \text{rws, ReducedForm}(\text{rws}, u) = \text{ReducedForm}(\text{rws}, v) \) as words in the free algebra \( T \). This element is the unique normal form of the element represented by \( u \).

In its second form, if \( A \) is an algebra with a canonical rewriting system associated with it, \text{IsConfluent} checks whether that rewriting system is confluent.

Also see 44.4.7.

6. \text{ConfluentRws}( \text{rws} )

   Return a new rewriting system defining the same algebra as \( \text{rws} \) which is confluent.

7. \text{IsReduced}( \text{rws} )

   A rewriting system is reduced if for each rule \((l, r)\), \( l \) and \( r \) are both reduced.

8. \text{ReduceRules}( \text{rws} )

   Reduce rules and remove redundant rules to make \( \text{rws} \) reduced.

9. \text{AddRule}( \text{rws}, \text{rule} )

   Add rule \( \text{rule} \) to a rewriting system \( \text{rws} \).

10. \text{AddRuleReduced}( \text{rws}, \text{rule} )

    Add rule to rewriting system \( \text{rws} \). Performs a reduction operation on the resulting system, so that if \( \text{rws} \) is reduced it will remain reduced.

11. \text{MakeConfluent}( \text{rws} )

    Add rules (and perhaps reduce) in order to make \( \text{rws} \) confluent

12. \text{GeneratorsOfRws}( \text{rws} )

    A

13. \text{AddGenerators}( \text{rws}, \text{gens} )

    O

36.2 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 \( A \).

1. \text{ReducedProduct}( \text{rws}, u, v )

   The result is \( w \) where \([u] = [u][v] \) 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).

2. \text{ReducedSum}( \text{rws}, \text{left}, \text{right} )

3. \text{ReducedOne}( \text{rws} )

4. \text{ReducedAdditiveInverse}( \text{rws}, \text{obj} )

5. \text{ReducedComm}( \text{rws}, \text{left}, \text{right} )

6. \text{ReducedConjugate}( \text{rws}, \text{left}, \text{right} )

7. \text{ReducedDifference}( \text{rws}, \text{left}, \text{right} )

8. \text{ReducedInverse}( \text{rws}, \text{obj} )

9. \text{ReducedLeftQuotient}( \text{rws}, \text{left}, \text{right} )

10. \text{ReducedPower}( \text{rws}, \text{obj}, \text{pow} )

11. \text{ReducedQuotient}( \text{rws}, \text{left}, \text{right} )

12. \text{ReducedScalarProduct}( \text{rws}, \text{left}, \text{right} )

13. \text{ReducedZero}( \text{rws} )
36.3 Properties of rewriting systems

The following properties may be used to identify the type of term algebra over which the rewriting system is defined.

1. IsBuiltFromAdditiveMagmaWithInverses( obj )
2. IsBuiltFromMagma( obj )
3. IsBuiltFromMagmaWithOne( obj )
4. IsBuiltFromMagmaWithInverses( obj )
5. IsBuiltFromSemigroup( obj )
6. IsBuiltFromGroup( obj )

36.4 Developing rewriting systems

The key point to note about rewriting systems is that they have properties such as IsConfluent and attributes such as Rules, 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 51), the function CreateKnuthBendixRewritingSystem creating the rewriting system (in kbsemi.gi) uses

```plaintext
kbrvs := 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, whenever IsConfluent is called, the appropriate method to determine confluence is called.
37 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 \( \{1, \ldots, 4\} \).

37.1 Group Elements

Groups in GAP are written multiplicatively. The elements from which a group can be generated must permit
multiplication and multiplicative inversion (see 30.14).

\begin{verbatim}
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
\end{verbatim}

The next example may run into an infinite loop because the given matrix in fact has infinite order.

\begin{verbatim}
gap> Order( [[ 1, 1 ], [ 0, 1 ]] * Indeterminate( Rationals ) );
#I Order: warning, order of <mat> might be infinite
\end{verbatim}

Since groups are domains, the recommended command to compute the order of a group is \texttt{Size} (see 28.3.6).
For convenience, group orders can also be computed with \texttt{Order}.

The operation \texttt{Comm} (see 30.12.3) can be used to compute the commutator of two elements, the operation
\texttt{LeftQuotient} (see 30.12.2) computes the product \( x^{-1}y \).

37.2 Creating Groups

When groups are created from generators, this means that the generators must be elements that can be
multiplied and inverted (see also 30.3). For creating a free group on a set of symbols, see 35.2.1.

\begin{verbatim}
\textbf{Group( \( \text{gen}, \ldots \) )} \quad \text{F} 
\textbf{Group( \( \text{gens} \) )} \quad \text{F} 
\textbf{Group( \( \text{gens}, \text{id} \) )} \quad \text{F}
\end{verbatim}

\textbf{Group( \( \text{gen}, \ldots \) )} is the group generated by the arguments \texttt{gen}, ...
If the only argument \( gens \) is a list that is not a matrix then \( \text{Group}(\ gens\ ) \) is the group generated by the elements of that list.

If there are two arguments, a list \( gens \) and an element \( id \), then \( \text{Group}(\ gens,\ id\ ) \) is the group generated by the elements of \( gens \), with identity \( id \).

Note that the value of the attribute \( \text{GeneratorsOfGroup} \) need not be equal to the list \( gens \) of generators entered as argument. Use \( \text{GroupWithGenerators} \) (see 37.2.2) if you want to be sure that the argument \( gens \) is stored as value of \( \text{GeneratorsOfGroup} \).

\[
gap> g:=\text{Group}(((1,2,3,4),(1,2));
group([ (1,2,3,4), (1,2) ])$$

\( \text{GroupWithGenerators}(\ gens\ ) \) \hspace{1cm} O
\( \text{GroupWithGenerators}(\ gens,\ id\ ) \) \hspace{1cm} O

\( \text{GroupWithGenerators} \) 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 \( \text{GeneratorsOfGroup} \) of \( G \) is equal to \( gens \).

\[
gap> g:=\text{GroupWithGenerators}([[1,2,3,4),(1,2)]);\n\text{Group}([[ 1,2,3,4), (1,2) ]])$$
\[
gap> \text{GeneratorsOfGroup}(g);\n[ (1,2,3,4), (1,2) ]$$

While in this example GAP displays the group via the generating set stored in the attribute \( \text{GeneratorsOfGroup} \), the methods installed for \( \text{View} \) (see 6.2.1) will in general display only some information about the group which may even be just the fact that it is a group.

\( \text{AsGroup}(\ D\ ) \) \hspace{1cm} A

if the elements of the collection \( D \) form a group the command returns this group, otherwise it returns \( \text{fail} \).

\[
gap> \text{AsGroup}([[1,2]]);\n\text{fail}$$
\[
gap> \text{AsGroup}([[\text{()},(1,2)]]);\n\text{Group}([[ (1,2) ]])$$

\( \text{ConjugateGroup}(\ G,\ obj\ ) \) \hspace{1cm} O

returns the conjugate group of \( G \), obtained by applying the conjugating element \( obj \). To form a conjugate (group) by any object acting via \( ^\cdot \), one can use the infix operator \( ^\cdot \).

\[
gap> \text{ConjugateGroup}(g,(1,5));\n\text{Group}([[ (2,3,4,5), (2,5) ]])$$

\( \text{IsGroup}(\ obj\ ) \) \hspace{1cm} C

A group is a magma-with-inverses (see 33.1.4) and associative (see 33.4.7) multiplication.

\( \text{IsGroup} \) tests whether the object \( obj \) fulfills these conditions, it does \( \text{not} \) test whether \( obj \) is a set of elements that forms a group under multiplication; use \( \text{AsGroup} \) (see 37.2.4) if you want to perform such a test. (See 13.3 for details about categories.)

\[
gap> \text{IsGroup}(g);\n\text{true}$$

\( \text{InfoGroup} \) \hspace{1cm} V

is the info class for the generic group theoretic functions (see 7.4).
37.3 Subgroups

For the general concept of parents and subdomains, see 30.7 and 30.8. More functions that construct certain subgroups can be found in the sections 37.10, 37.11, 37.12, and 37.13.

1► \textbf{Subgroup}( G, \textit{gens} ) \hspace{1cm} \textbf{F}
2► \textbf{SubgroupNC}( G, \textit{gens} ) \hspace{1cm} \textbf{F}

creates the subgroup \( U \) of \( G \) generated by \textit{gens}. The \textbf{Parent} of \( U \) will be \( G \). The \textbf{NC} version does not check, whether the elements in \textit{gens} actually lie in \( G \).

\[
\text{gap> } u:=\text{Subgroup}(g,[(1,2,3),(1,2)]);
\text{Group}([ (1,2,3), (1,2) ])
\]

2► \textbf{Index}( G, U ) \hspace{1cm} \textbf{O}
3► \textbf{IndexNC}( G, U ) \hspace{1cm} \textbf{O}

For a subgroup \( U \) of the group \( G \), \textbf{Index} returns the index \([ G : U ] = \frac{|G|}{|U|}\) of \( U \) in \( G \). The \textbf{NC} version does not test whether \( U \) is contained in \( G \).

\[
\text{gap> } \text{Index}(g,u);
4
\]

3► \textbf{IndexInWholeGroup}( G ) \hspace{1cm} \textbf{A}

If the family of elements of \( G \) itself forms a group \( P \), this attribute returns the index of \( G \) in \( P \).

4► \textbf{AsSubgroup}( G, U ) \hspace{1cm} \textbf{O}

creates a subgroup of \( G \) which contains the same elements as \( U \)

\[
\text{gap> } v:=\text{AsSubgroup}(g,\text{Group}([(1,2,3),(1,4)]))
\text{Group}([ (1,2,3), (1,4) ])
\text{gap> } \text{Parent}(v);
\text{Group}([ (1,2,3,4), (1,2) ])
\]

5► \textbf{IsSubgroup}( G, U ) \hspace{1cm} \textbf{F}

\textbf{IsSubgroup} returns \textbf{true} if \( U \) is a group that is a subset of the domain \( G \). This is actually checked by calling \textbf{IsGroup}( U ) and \textbf{IsSubset}( G, U ); note that special methods for \textbf{IsSubset} (see 28.4.1) 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 \textbf{IsSubset}.

\[
\text{gap> } \text{IsSubgroup}(g,u);
\text{true}
\text{gap> } v:=\text{Group}([(1,2,3),(1,2)]);
\text{Group}([ (1,2,3), (1,2) ])
\text{gap> } u=v;
\text{true}
\text{gap> } \text{IsSubgroup}(g,v);
\text{true}
\]

6► \textbf{IsNormal}( G, U ) \hspace{1cm} \textbf{O}

returns \textbf{true} if the group \( G \) normalizes the group \( U \) and \textbf{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 \( G \).
Chapter 37. Groups

7 ▶ IsCharacteristicSubgroup( G, N )
   tests whether N is invariant under all automorphisms of G.
   gap> IsCharacteristicSubgroup(g,u);
   false
8 ▶ ConjugateSubgroup( G, g )
9 ▶ ConjugateSubgroups( G, U )
   returns a list of all images of the group U under conjugation action by G.
10 ▶ IsSubnormal( G, U )
   A subgroup U of the group G is subnormal if it is contained in a subnormal series of G.
   gap> IsSubnormal(g,Group((1,2,3)));
   false
   gap> IsSubnormal(g,Group((1,2)(3,4)));
   true

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) with the parent as an attribute.

37.4 Closures of (Sub)groups

1 ▶ ClosureGroup( G, obj )
   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.
   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 ])

2 ▶ ClosureGroupAddElm( G, elm )
3 ▶ ClosureGroupCompare( G, elm )
4 ▶ ClosureGroupIntest( G, elm )
   These three functions together with ClosureGroupDefault implement the main methods for ClosureGroup (see 37.4.1). 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.

3 ▶ ClosureGroupDefault( G, elm )
   This functions returns the closure of the group G with the element elm. If G has the attribute AsSSortedList then also the result has this attribute. This is used to implement the default method for Enumerator (see 28.2.2) and EnumeratorSorted (see 28.2.3).
Section 5. Expressing Group Elements as Words in Generators

For a group \( G \) that stores a parent group (see 30.7), \texttt{ClosureSubgroup} calls \texttt{ClosureGroup} (see 37.4.1) 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 \texttt{NC} version. As a wrong parent might imply wrong properties this version should be used with care.

### 37.5 Expressing Group Elements as Words in Generators

Using homomorphisms (see chapter 38) is is possible to express group elements as words in given generators: Create a free group (see 35.2.1) 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. The following example shows how to decompose elements of \( S_4 \) in the generators \((1,2,3,4)\) and \((1,2)\):

```gap
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> f:=FreeGroup("x","y");
<free group on the generators [ x, y ]>
gap> hom:=GroupHomomorphismByImagesNC(f,g,GeneratorsOfGroup(f),
  GeneratorsOfGroup(g));
[ x, y ] -> [ (1,2,3,4), (1,2) ]
gap> PreImagesRepresentative(hom,(1,4));
x*y^-1*x^-1
```

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 \((15,16)\) as a word as short as possible in the generators of a particular permutation group of degree 16.

```gap
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 );;
gap> Size( puzzle );
20922789888000
gap> F := FreeGroup( "a", "b", "c", "d" );;
gap> gens := GeneratorsOfGroup( F );;
gap> hom := GroupHomomorphismByImages( F, puzzle, gens, perms );;
gap> word := PreImagesRepresentative( hom, (15,16) );;
gap> Length( word );
113
```

Because of the random methods involved in the function \texttt{GroupHomomorphismByImages} we may even get shorter words for the given permutation if we try it a few more times.

```gap
gap> words := [];
gap> for i in [ 1 .. 10 ] do
  >   hom := GroupHomomorphismByImages( F, puzzle, gens, perms );
  >   word := PreImagesRepresentative( hom, (15,16) );
  >   Add( words, word );
  > od;
gap> List( words, word -> Length( word ) );
[ 33, 61, 53, 33, 33, 33, 49, 53, 75, 33 ]
```
returns a factorization of \( elm \) as word in the generators of \( G \) given in the attribute \texttt{GeneratorsOfGroup}. (The corresponding free generators can be obtained via the family \texttt{fam} of the result as \texttt{CollectionsFamily(fam)}!\!.\texttt{wholeGroup}.)

The algorithm used computes 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 thousand elements. Because of this, one should not call this function within algorithms, but use homomorphisms instead.

\[
\begin{align*}
gap> \text{Factorization( } G, \ elm \ )
\end{align*}
\]

right cosets 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 \texttt{OnLeftInverse}. Of course the action of a larger group \( G \) on right cosets is via \texttt{OnRight}.

\[
\begin{align*}
gap> \text{RightCoset( } U, \ g \ )
\end{align*}
\]

computes a duplicate free list of right cosets \( Ug \) 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 \( G \).

\[
\begin{align*}
\text{37.6 Cosets}
\end{align*}
\]

\[
\begin{align*}
\text{RightCosets( } G, \ U \ ) & \quad \text{F} \\
\text{RightCosetsNC( } G, \ U \ ) & \quad \text{O}
\end{align*}
\]
Section 7. Transversals

3.7 Transversals

A right transversal $t$ is a list of representatives for the set $U \backslash G$ of right cosets (consisting of cosets $Ug$) of $U$ in $G$.

The object returned by $\text{RightTransversal}$ 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 $\text{RightTransversal}$ may return the sorted plain list of coset representatives.

The operation $\text{PositionCanonical}(t,g)$, 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, $\text{Position}$ will return $\text{fail}$ if the element is not equal to the representative.) Functions that implement group actions such as $\text{Action}$ or $\text{Permutation}$ (see Chapter 39) use $\text{PositionCanonical}$, 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.

Note that the elements of a right transversal are not necessarily “canonical” in the sense of $\text{CanonicalRightCosetElement}$ (see 37.6.3), but we may compute a list of canonical coset representatives by calling that function.
gap> List(RightTransversal(g,u),i->CanonicalRightCosetElement(u,i));
[ (), (2,3,4), (1,2,3,4), (3,4) ]

The operation PositionCanonical is described in section 21.16.2.

gap> PositionCanonical(rt,(1,2,3));
1
gap> rt[1];
()

37.8 Double Cosets

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 UgV which is the set of all elements of the form ugv for any u ∈ U, v ∈ 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, and V as RightActingGroup.

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) ]

computes a duplicate free list of all double cosets UgV for g ∈ G. U and V must be subgroups of the group G. The NC version does not check whether U and V are both subgroups of G.

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) ]
Section 9. Conjugacy Classes

5► DoubleCosetRepsAndSizes( G, U, V )
returns a list of double coset representatives and their sizes, the entries are lists of the form [rep, size]. This operation is faster than DoubleCosetsNC because no double coset objects have to be created.

\[
\text{gap> dc:=DoubleCosetRepsAndSizes(g,u,v);}\\
\text{[ [ ()], [ (1,3)(2,4), 6 ], [ (1,4)(2,3), 6 ] ]}
\]

6► InfoCoset
The information function for coset and double coset operations is InfoCoset.

37.9 Conjugacy Classes

1► ConjugacyClass( G, g )
creates the conjugacy class in \( G \) with representative \( g \). This class is an external set, so functions such as Representative (which returns \( g \)), ActingDomain (which returns \( G \)), StabilizerOfExternalSet (which returns the centralizer of \( g \)) and AsList work for it.

A conjugacy class is an external orbit (39.11.9) of group elements with the group acting by conjugation on it. Thus element tests or operation representatives can be computed. The attribute Centralizer gives the centralizer of the representative (which is the same result as StabilizerOfExternalSet). (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.)

2► ConjugacyClasses( G )
returns the conjugacy classes of elements of \( G \) as a list of ConjugacyClasses of \( G \) (see ConjugacyClass (37.9.1) 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 37.9.4). 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 43.17). 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 (37.9.3) below.

action
The classes are computed by action of \( G \) on itself See ConjugacyClassesByOrbits (37.9.4) below.

noaction
Even for small groups ConjugacyClassesByOrbits (37.9.4) is not used as a default. This can be useful if the elements of the group use a lot of memory.

\[
\text{gap> g:=SymmetricGroup(4);;}\\
\text{gap> cl:=ConjugacyClasses(g);}\\
\text{[ ()^G, (1,2)^G, (1,2)(3,4)^G, (1,2,3)^G, (1,2,3,4)^G ]}\\
\text{gap> Representative(cl[3]);Centralizer(cl[3]);}\\
\text{[ (1,2)(3,4) ]}\\
\text{Group([[ (1,2), (1,3)(2,4), (3,4) ]])}\\
\text{gap> Size(Centralizer(cl[5]));}\\
\text{4}\\
\text{gap> Size(cl[2]);}\\
\text{6}
\]

3► ConjugacyClassesByRandomSearch( G )
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 ConjugacyClass.

\textbf{ConjugacyClassesByOrbits(} $G$ \textbf{)} \hspace{1cm} F

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 ConjugacyClass.

\begin{verbatim}
gap> h:=Group((4,6)(5,7),(1,2,4)(3,6,5));;ConjugacyClasses(h:noaction);;time;
190

gap> h:=Group((4,6)(5,7),(1,2,4)(3,6,5));;ConjugacyClasses(h:random);;time;
130

gap> h:=Group((4,6)(5,7),(1,2,4)(3,6,5));;ConjugacyClasses(h:action);;time;
70
\end{verbatim}

\textbf{NrConjugacyClasses(} $G$ \textbf{)} \hspace{1cm} A

returns the number of conjugacy classes of $G$.

\begin{verbatim}
gap> g:=Group((1,2,3,4),(1,2));;
gap> NrConjugacyClasses(g);
5
\end{verbatim}

\textbf{RationalClass(} $G$, $g$ \textbf{)} \hspace{1cm} C

creates the rational class in $G$ with representative $g$. A rational class consists of all elements that are conjugate to $g$ or to a power $g^i$ 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 (39.11.1) of group elements with the group acting by conjugation on it, but not an external orbit.

\textbf{RationalClasses(} $G$ \textbf{)} \hspace{1cm} A

returns a list of the rational classes of the group $G$. (See 37.9.6.)

\begin{verbatim}
gap> RationalClasses(DerivedSubgroup(g));
[ RationalClass( AlternatingGroup( [ 1 .. 4 ] ), () ),
  RationalClass( AlternatingGroup( [ 1 .. 4 ] ), (1,2)(3,4) ),
  RationalClass( AlternatingGroup( [ 1 .. 4 ] ), (1,2,3) ) ]
\end{verbatim}

\textbf{GaloisGroup(} ratcl \textbf{)} \hspace{1cm} A

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 15.2.3), the so-called Galois group of the rational class. The prime residue class group $P_n$ is obtained in GAP as \texttt{Units( Integers mod n )}, the unit group of a residue class ring. The Galois group of a rational class \texttt{rcl} is stored in the attribute \texttt{GaloisGroup(rcl)} as a subgroup of this group.

\textbf{IsConjugate(} $G$, $x$, $y$ \textbf{)} \hspace{1cm} O
\textbf{IsConjugate(} $G$, $U$, $V$ \textbf{)} \hspace{1cm} O

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 \texttt{RepresentativeOperation}.

\begin{verbatim}
gap> IsConjugate(g,Group((1,2,3,4),(1,3)),Group((1,3,2,4),(1,2)));
true
\end{verbatim}

\texttt{RepresentativeAction} (see 39.5.1) can be used to obtain conjugating elements.

\begin{verbatim}
gap> RepresentativeAction(g,(1,2),(3,4));
(1,3)(2,4)
\end{verbatim}
37.10 Normal Structure

For the operations Centralizer and Centre, see Chapter 33.

1. Normalizer( $G$, $U$ )
2. Normalizer( $G$, $g$ )

Computes the normalizer $N_G(U)$, that is the stabilizer of $U$ under the conjugation action of $G$. The second form computes $N_G(\langle g \rangle)$.

```gap
gap> Normalizer(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3), (2,3) ])
```

3. Core( $S$, $U$ )

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 $U$.

```gap
g:=Group((1,2,3,4),(1,2));
gap> Core(g,Subgroup(g,[(1,2,3,4)]));
Group(())
```

4. PCore( $G$, $p$ )

The $p$-core of $G$ is the largest normal $p$-subgroup of $G$. It is the core of a $p$-Sylow subgroup of $G$.

```gap
gap> PCore(g,2);
Group([ (1,4)(2,3), (1,3)(2,4) ])
```

5. NormalClosure( $G$, $U$ )

The normal closure of $U$ in $G$ is the smallest normal subgroup of $G$ which contains $U$.

```gap
gap> NormalClosure(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3), (1,3,4) ])
```

6. NormalIntersection( $G$, $U$ )

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 $U$.

```gap
gap> NormalIntersection(Group((1,2)(3,4),(1,3)(2,4)),Group((1,2,3,4)));
Group([ (1,3)(2,4) ])
```

7. Complementclasses( $G$, $N$ )

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 $U$ of $G$ which intersect trivially with $N$ and together with $N$ generate $G$.

```gap
gap> Complementclasses(g,Group((1,2)(3,4),(1,3)(2,4)));
[ Group([ (3,4), (2,4,3) ] ) ]
```

8. InfoComplement

Info class for the complement routines.


37.11 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 (see 33.4.5).

1\textbf{\textgreater} TrivialSubgroup( \textit{G} )
\begin{verbatim}
gap> TrivialSubgroup(g);
Group(())
\end{verbatim}

2\textbf{\textgreater} CommutatorSubgroup( \textit{G}, \textit{H} )
\begin{verbatim}
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 30.12.3) of elements \(g \in G\) and \(h \in H\), that is the group \(\langle [g,h] | g \in G, h \in H \rangle\).
\end{verbatim}
\begin{verbatim}
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
\end{verbatim}

3\textbf{\textgreater} DerivedSubgroup( \textit{G} )
\begin{verbatim}
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 \(G\).
\end{verbatim}
\begin{verbatim}
gap> DerivedSubgroup(g);
Group([ (1,3,2), (1,4,3) ])
\end{verbatim}

4\textbf{\textgreater} CommutatorLength( \textit{G} )
\begin{verbatim}
returns the minimal number \(n\) such that each element in the derived subgroup (see 37.11.3) of the group \(G\) can be written as a product of (at most) \(n\) commutators of elements in \(G\).
\end{verbatim}
\begin{verbatim}
gap> CommutatorLength( g );
1
\end{verbatim}

5\textbf{\textgreater} FittingSubgroup( \textit{G} )
\begin{verbatim}
The Fitting subgroup of a group \(G\) is its largest nilpotent normal subgroup.
\end{verbatim}
\begin{verbatim}
gap> FittingSubgroup(g);
Group([ (1,3)(2,4), (1,4)(2,3) ])
\end{verbatim}

6\textbf{\textgreater} FrattiniSubgroup( \textit{G} )
\begin{verbatim}
The Frattini subgroup of a group \(G\) is the intersection of all maximal subgroups of \(G\).
\end{verbatim}
\begin{verbatim}
gap> FrattiniSubgroup(g);
Group(())
\end{verbatim}

7\textbf{\textgreater} PrefrattiniSubgroup( \textit{G} )
\begin{verbatim}
returns a Prefrattini subgroup of the finite solvable group \(G\). A factor \(M/N\) of \(G\) is called a Frattini factor if \(M/N \leq \phi(G/N)\) holds. The group \(P\) 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 \(G\).
\end{verbatim}
Section 11. Specific and Parametrized Subgroups

```gap
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([ ])
```

8 ▶ PerfectResiduum( G )

is the smallest normal subgroup of $G$ that has a solvable factor group.

```gap
gap> PerfectResiduum(Group((1,2,3,4,5),(1,2)));
Group([ (1,3,2), (1,4,3), (2,4,5) ])
```

9 ▶ RadicalGroup( G )

is the radical of $G$, i.e., the largest solvable normal subgroup of $G$.

```gap
gap> RadicalGroup(SL(2,5));
<group of 2x2 matrices of size 2 in characteristic 5>
gap> Size(last);
2
```

10 ▶ Socle( G )

The socle of the group $G$ is the subgroup generated by all minimal normal subgroups.

```gap
gap> Socle(g);
Group([ (1,4)(2,3), (1,2)(3,4) ])
```

11 ▶ SupersolvableResiduum( G )

is the supersolvable residuum of the group $G$, that is, its smallest normal subgroup with supersolvable factor group.

```gap
gap> SupersolvableResiduum(g);
Group([ (1,2)(3,4), (1,4)(2,3) ])
```

12 ▶ PRump( G, p )

The $p$-rump of a group $G$ is the subgroup $G'G^p$ for a prime $p$.

@example missing!@
37.12 Sylow Subgroups and Hall Subgroups

1. **SylowSubgroup( G, p )**

   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`.

   ```gap
   gap> g:=SymmetricGroup(4);
   gap> SylowSubgroup(g,2);
   Group([ (1,2), (3,4), (1,3)(2,4) ]) 
   ```

   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 37.12.2).
   2. For each set $P$ of primes dividing the order of $G$, there exists a $P$-Hall subgroup (see 37.12.3).
   3. $G$ has a Sylow system (see 37.12.4).
   4. $G$ has a complement system (see 37.12.5).

2. **SylowComplement( G, p )**

   Returns a $p$-Sylow 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.

   ```gap
   gap> SylowComplement(g,3);
   Group([ (3,4), (1,4)(2,3), (1,3)(2,4) ]) 
   ```

3. **HallSubgroup( G, P )**

   Computes a $P$-Hall subgroup for a set $P$ of primes. This is a subgroup whose order is only divisible by primes in $P$ and whose index is coprime to all primes in $P$. The function computes Hall subgroups via the operation `HallSubgroupOp`. At the moment methods exist only if $G$ is solvable and GAP will issue an error if $G$ is not solvable.

   ```gap
   gap> h:=SmallGroup(60,10);
   gap> u:=HallSubgroup(h,[2,3]);
   Group([ f1, f2, f3 ])
   gap> Size(u);
   12 
   ```

4. **SylowSystem( G )**

   A Sylow system of a group $G$ is a set of Sylow subgroups of $G$ such that every pair of Sylow 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.

   ```gap
   gap> h:=SmallGroup(60,10);
   gap> SylowSystem(h);
   [ Group([ f1, f2 ]), Group([ f3 ]), Group([ f4 ])]
   gap> List(last,Size);
   [ 4, 3, 5 ] 
   ```

5. **ComplementSystem( G )**

   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.

```gap
gap> ComplementSystem(h);
[ Group([ f3, f4 ]), Group([ f1, f2, f4 ]), Group([ f1, f2, f3 ]) ]
gap> List(last,Size);
[ 15, 20, 12 ]
```

6 ▶ `HallSystem(G)`

returns a list containing one Hall-\( P \)-subgroup for each set \( P \) of primes which occur in the order of \( G \). Hall systems exist only for solvable groups. The operation returns `fail` if the group \( G \) is not solvable.

```gap
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 ]
```

37.13 Subgroups characterized by prime powers

1 ▶ `Omega(G, p[, n])`

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 \( n=1 \).@

```gap
gap> h:=SmallGroup(16,10);
<pc group of size 16 with 4 generators>
gap> Omega(h,2);
Group([ f4, f2, f3 ])
```

2 ▶ `Agemo(G, p[, n])`

For a \( p \)-group \( G \), one defines \( \Omega_n(G) = \langle g^{p^n} \mid g \in G \rangle \). The default value for \( n \) is 1.

```gap
gap> Agemo(h,2);Agemo(h,2,2);
Group([ f4 ])
Group([ ])
```

37.14 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.

1 ▶ `IsCyclic(G)`

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` (see 37.21.3).

2 ▶ `IsElementaryAbelian(G)`

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 \( p \).
A group is **nilpotent** if the lower central series (see 37.16.11 for a definition) reaches the trivial subgroup in a finite number of steps.

The nilpotency class of a nilpotent group $G$ is the number of steps in the lower central series of $G$ (see 37.16.11);

If $G$ is not nilpotent an error is issued.

A group is **perfect** if it equals its derived subgroup (see 37.11.3).

A group is **solvable** if the derived series (see 37.16.7 for a definition) reaches the trivial subgroup in a finite number of steps.

For finite groups this is the same as being polycyclic (see 37.14.7), and each polycyclic group is solvable, but there are infinite solvable groups that are not polycyclic.

A group is **supersolvable** if it has a normal series with cyclic factors.

A finite group is **monomial** if every irreducible complex character is induced from a linear character of a subgroup.

A group is **simple** if it has no nontrivial normal subgroups.

For a finite simple group $G$, `IsomorphismTypeInfoFiniteSimpleGroup(G)` returns a record with 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 $F(q, q)$.

"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 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.

\begin{verbatim}
gap> IsomorphismTypeInfoFiniteSimpleGroup(Group((4,5)(6,7),(1,2,4)(3,5,6))); rec( series := "L", parameter := [ 2, 7 ], 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)" )
12 ◄ IsFinitelyGeneratedGroup( G ) P tests whether the group $G$ can be generated by a finite number of generators. (This property is mainly used to obtain finiteness conditions.)
13 ◄ IsSubsetLocallyFiniteGroup( U ) P 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.
14 ◄ IsPGroup( G ) P A $p$-group is a finite group whose order (see 28.3.6) 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.
15 ◄ PrimePGroup( G ) A If $G$ is a nontrivial $p$-group (see 37.14.14), PrimePGroup returns the prime integer $p$; if $G$ is trivial then PrimePGroup returns fail. Otherwise an error is issued.
16 ◄ PClassPGroup( G ) A The $p$-class of a $p$-group $G$ (see 37.14.14) is the length of the lower $p$-central series (see 37.16.13) of $G$. If $G$ is not a $p$-group then an error is issued.
17 ◄ RankPGroup( G ) A For a $p$-group $G$ (see 37.14.14), 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.
\end{verbatim}

Note that the following functions, 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.
IsPSolvable( G, p )

A group is \( p \)-solvable if every chief factor is either not divisible by \( p \) or solvable.

@Currently no method is installed!@

IsPNilpotent( G, p )

A group is \( p \)-nilpotent if it possesses a normal \( p \)-complement.

37.15 Numerical Group Attributes

AbelianInvariants( G )

returns the abelian invariants of the commutator factor group of the group \( G \). They are given as a list of the orders of a set of independent generators of \( G/G' \) (see 37.21.5).

\[ [ 2, 2 ] \]

Exponent( G )

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 \( g \in G \).

\[ 12 \]

EulerianFunction( G, n )

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 an \( n \)-tuple need not be different.

\[ 432 \]

37.16 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.

ChiefSeries( G )

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).
Section 16. Subgroup Series

\begin{verbatim}
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(()) ]

2 ▶ ChiefSeriesThrough( G, l )

is a chief series of the group G going through the normal subgroups in the list l. l 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).

3 ▶ ChiefSeriesUnderAction( H, G )

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).

4 ▶ SubnormalSeries( G, U )

If U is a subgroup of G this operation returns a subnormal series that descends from G to a subnormal subgroup V ≤ U. If U is subnormal, V = U.

\begin{verbatim}
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) ]) ]
\end{verbatim}

5 ▶ CompositionSeries( G )

A composition series is a subnormal series which cannot be refined. This attribute returns one composition series (of potentially many possibilities).

6 ▶ DisplayCompositionSeries( G )

Displays a composition series of G in a nice way, identifying the simple factors.

\begin{verbatim}
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) ]) ]
gap> DisplayCompositionSeries(Group((1,2,3,4,5,6,7),(1,2)));
G (6 gens, size 5040)
  | Z(2)
S (5 gens, size 2520)
  | A(7)
  | 1 (0 gens, size 1)
\end{verbatim}

7 ▶ DerivedSeriesOfGroup( G )

The derived series of a group is obtained by \( U_{i+1} = U_i' \). It stops if \( U_i \) is perfect.

8 ▶ DerivedLength( G )

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.)
gap> List(DerivedSeriesOfGroup(g),Size);
[ 24, 12, 4, 1 ]
gap> DerivedLength(g);
3

ElementaryAbelianSeries( G )
ElementaryAbelianSeriesLargeSteps( G )
ElementaryAbelianSeries( [G, NT1, NT2, ...] )

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 is constructed.

gap> List(ElementaryAbelianSeries(g),Size);
[ 24, 12, 4, 1 ]

InvariantElementaryAbelianSeries( G, morph[, N [, fine]] )

For a (solvable) group $G$ and a list of automorphisms $\text{morph}$ 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 $\text{morph}$.

If a normal subgroup $N$ of $G$ which is invariant under $\text{morph}$ 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 $\text{fine}$ is given as true.

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)]);
Group([ (1,2,3,4), (1,3) ] -> [ (1,4,3,2), (1,4)(2,3) ])

LowerCentralSeriesOfGroup( G )

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.

UpperCentralSeriesOfGroup( G )

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.

PCentralSeries( G, p )

The $p$-central series of $G$ is defined by $U_1 := G$, $U_i := [G, U_{i-1}] U_{p-1}$.

JenningsSeries( G )

For a $p$-group $G$, this function returns its Jennings series. This series is defined by setting $G_1 = G$ and for $i \geq 0$, $G_{i+1} = [G_i, G]G_j$, where $j$ is the smallest integer $\geq i/p$. 
DimensionsLoewyFactors( G )

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 without computing the Loewy series itself.

\begin{verbatim}
gap> G:= SmallGroup( 3^6, 100 );
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([<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, 26, 25, 23, 22, 20, 19, 17, 16, 14, 13, 11, 10,
  8, 7, 5, 4, 2, 1 ]
\end{verbatim}

AscendingChain( G, U )

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.

IntermediateGroup( G, U )

This routine tries to find a subgroup \( E \) of \( G \) such that \( G > E > U \). If \( U \) is maximal, it returns false. This is done by finding minimal blocks for the operation of \( G \) on the right cosets of \( U \).

IntermediateSubgroups( G, U )

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 components 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 subgroup number \( i \) is a maximal subgroup of subgroup number \( j \), the numbers 0 and 1+length(subgroups) are used to denote \( U \) and \( G \) respectively.

Factor Groups

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 should be used instead of Range to get a group isomorphic to the factor group. The NC variant does not check whether \( N \) is normal in \( G \).

FactorGroupNC( G, N )

returns the image of the NaturalHomomorphismByNormalSubgroup(\( G, N \)). The NC version does not test whether \( N \) is normal in \( G \).
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 ])

37.18 Sets of Subgroups

**ConjugacyClassSubgroups** \(( G, U )\)

generates the conjugacy class of subgroups of \( G \) with representative \( U \). This class is an external set, so functions such as `Representative`, (which returns \( U \)), `ActingDomain` (which returns \( G \)), `StabilizerOfExternalSet` (which returns the normalizer of \( U \)), and `AsList` work for it.

(The use the [] list access to select elements of the class is considered obsolescent and will be removed in future versions. Use `ClassElementLattice` instead.)
Section 18. Sets of Subgroups

gap> g:=Group((1,2,3,4),(1,2));;IsNaturalSymmetricGroup(g);
gap> cl:=ConjugacyClassSubgroups(g,Subgroup(g,[1,2]));
group((1,2))

357

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.

3 ConjugacyClassesSubgroups( G )

This attribute returns a list of all conjugacy classes of subgroups of the group G. It also is applicable for
lattices of subgroups (see 37.19.1). 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 (see 28.3.7).

gap> ConjugacyClassesSubgroups(g);
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (1,2) ] )^G,
  Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
  Group( [ (1,2)(3,4), (1,2) ] )^G, Group( [ (1,3)(2,4), (1,4,2,3) ] )^G,
  Group( [ (3,4), (2,4,3) ] )^G, Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3) ] )^G,
  Group( [ (1,3)(2,4), (2,3), (2,4,3), (1,2) ] )^G ]

4 ConjugacyClassesMaximalSubgroups( G )

returns the conjugacy classes of maximal subgroups of G. Representatives of the classes can be computed
directly by MaximalSubgroupClassReps (see 37.18.5).

gap> ConjugacyClassesMaximalSubgroups(g);
[ AlternatingGroup( [ 1 .. 4 ] )^G, Group( [ (1,2), (3,4), (1,3)(2,4) ] )^G, Group( [ (1,2), (3,4), (1,3)(2,4) ] )^G ]

5 MaximalSubgroupClassReps( G )

returns a list of conjugacy representatives of the maximal subgroups of G.

  gap> MaximalSubgroupClassReps(g);
  [ Alt( [ 1 .. 4 ] ), Group([ (1,2,3), (1,2) ]),
    Group([ (1,2), (3,4), (1,3)(2,4) ])]

6 MaximalSubgroups( G )

returns a list of all maximal subgroups of G. This may take up much space, therefore the command should
be avoided if possible. See 37.18.4.

  gap> MaximalSubgroups(Group([1,2,3,1,2]));
  [ Group([ (1,2,3) ]), Group([ (2,3) ]), Group([ (1,2) ]), Group([ (1,3) ])]

7 NormalSubgroups( G )

returns a list of all normal subgroups of G.
Chapter 37. Groups

8 ▶ MaximalNormalSubgroups( G )

is a list containing those proper normal subgroups of the group G that are maximal among the proper normal subgroups.

37.19 Subgroup Lattice

The GAP package XGAP permits a graphical display of the lattice of subgroups in a nice way.

1 ▶ LatticeSubgroups( G )

computes the lattice of subgroups of the group G. This lattice has the conjugacy classes of subgroups as attribute ConjugacyClassesSubgroups (see 37.18.3) and permits one to test maximality/minimality relations.

2 ▶ ClassElementLattice( C, n )

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, AsList(C) can give a different arrangement of the class elements!

3 ▶ MaximalSubgroupsLattice( lat )

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 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 [cls,nr] which means that the nrst subgroup in class number cls is a maximal subgroup of the representative.

The number nr corresponds to access via ClassElementLattice and not necessarily the AsList arrangement! See also 37.19.4.
Section 19. Subgroup Lattice

gap> MaximalSubgroupsLattice(1);
[ [ ], [ [ 1, 1 ] ], [ [ 1, 1 ] ],
  [ [ 7, 1 ] ], [ [ 6, 1 ] ], [ [ 5, 1 ] ],
  [ [ 8, 4 ] ] ]
gap> last[6];
[ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ]
gap> u1:=Representative(ConjugacyClassesSubgroups(1)[6]);
Group([ (1,2)(3,4), (1,2) ])
gap> u2:=ClassElementLattice(ConjugacyClassesSubgroups(1)[3],1);
true
gap> u3:=ClassElementLattice(ConjugacyClassesSubgroups(1)[3],6);
true
gap> u4:=ClassElementLattice(ConjugacyClassesSubgroups(1)[2],3);
true
gap> IsSubgroup(u1,u2);IsSubgroup(u1,u3);IsSubgroup(u1,u4);
true
true
true

MinimalSupergroupsLattice(lat)

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 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 [cls,nr] which means that the nrst subgroup in class number cls is a minimal supergroup of the representative.

The number nr corresponds to access via ClassElementLattice and not necessarily the AsList arrangement! See also 37.19.3.

gap> MinimalSupergroupsLattice(1);
  [ [ 5, 1 ] ], [ [ 6, 2 ] ], [ [ 7, 2 ] ], [ [ 6, 1 ] ], [ [ 8, 2 ] ], [ [ 8, 3 ] ],
  [ [ 8, 1 ] ], [ [ 10, 1 ] ], [ [ 9, 1 ] ], [ [ 9, 2 ] ], [ [ 9, 3 ] ], [ [ 10, 1 ] ],
  [ ] ]
gap> last[3];
[ [ 6, 1 ] ], [ [ 8, 2 ] ], [ [ 8, 3 ] ]
gap> u5:=ClassElementLattice(ConjugacyClassesSubgroups(1)[8],2);
Group([ (1,2), (1,4,2) ])
gap> u6:=ClassElementLattice(ConjugacyClassesSubgroups(1)[8],3);
Group([ (1,2), (1,2,3) ])
gap> IsSubgroup(u5,u2);
true
gap> IsSubgroup(u6,u2);
true

RepresentativesPerfectSubgroups(G)

returns a list of conjugacy representatives of perfect (respectively simple) subgroups of G. This uses the library of perfect groups (see 48.8.2), thus it will issue an error if the library is insufficient to determine all perfect subgroups.
Chapter 37. Groups

gap> m11:=TransitiveGroup(11,6);
M(11)
gap> r:=RepresentativesPerfectSubgroups(m11);
[ Group([ (1,6,8)(3,5,11)(7,9,10), (1,7)(4,9)(5,10)(8,11) ]),
  Group([ (1,5,7)(2,9,11)(3,10,6), (1,7)(4,9)(5,10)(8,11) ]),
  Group([ (1,11)(3,8)(5,6)(9,10), (1,10,7,5)(4,11,9,8) ]),
  Group([ (1,10,11)(2,7,6)(3,8,9), (1,7)(4,9)(5,10)(8,11) ]), M(11),
  Group(()) ]
gap> List(r,Size);
[ 60, 60, 360, 660, 7920, 1 ]

6 ▶ ConjugacyClassesPerfectSubgroups(G) returns a list of the conjugacy classes of perfect subgroups of G. (see 37.19.5.)

gap> ConjugacyClassesPerfectSubgroups(m11);
[ Group( [ ( 1, 6, 8)( 3, 5,11)( 7, 9,10), ( 1, 7)( 4, 9)( 5,10)( 8,11) ])^G,
  Group( [ ( 1, 5, 7)( 2, 9,11)( 3,10, 6), ( 1, 7)( 4, 9)( 5,10)( 8,11) ])^G,
  Group( [ ( 1,11)( 3, 8)( 5, 6)( 9,10), ( 1,10, 7, 5)( 4,11, 9, 8) ])^G,
  Group( [ ( 1,10,11)( 2, 7, 6)( 3, 8, 9), ( 1, 7)( 4, 9)( 5,10)( 8,11) ])^G,
  M(11)^G, Group( () )^G ]

7 ▶ Zuppos(G) 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.

8 ▶ InfoLattice is the information class used by the cyclic extension methods for subgroup lattice calculations.

37.20 Specific Methods for Subgroup Lattice Computations

1 ▶ LatticeByCyclicExtension(G[, func[, noperf]]) 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.

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.

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:
gap> LatticeSubgroups(g);
<subgroup lattice of <permutation group of size 5184 with 9 generators>,
566 classes, 27134 subgroups>

2 ▶ InvariantSubgroupsElementaryAbelianGroup( G, homs[, dims ] )

Let $G$ be an elementary abelian group (that is a vector space) and $homs$ 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 subspaces. If the optional parameter $dims$ is given, only subspaces of this dimension are computed.

```gap
g:=Group((1,2,3),(4,5,6),(7,8,9));
Group([ (1,2,3), (4,5,6), (7,8,9) ])
> 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) ]
u:=InvariantSubgroupsElementaryAbelianGroup(g,hom);
[ Group(()), Group([ (1,3,2)(7,8,9), (1,3,2)(4,5,6) ]),
  Group([ (7,8,9), (4,5,6), (1,2,3) ]) ]
```

3 ▶ SubgroupsSolvableGroup( G[, opt] )

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 $false$ 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 $G$. 

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.

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)]);

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

This function returns a function consider of four arguments that can be used in SubgroupsSolvableGroup (see 37.20.3) for the option consider to compute subgroups whose sizes are divisible by size.

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.

This function returns a function consider of four arguments that can be used in SubgroupsSolvableGroup (see 37.20.3) for the option consider to compute subgroups whose sizes are exactly size.

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.

Information function for the subgroup lattice functions using pcgs.
Section 22. 1-Cohomology

37.21 Special Generating Sets

1 ▶ GeneratorsSmallest( G )
returns a “smallest” generating set for the group G. This is the lexicographically (using GAP's order of group elements) smallest list l of elements of G such that G = ⟨ l ⟩ and li /∈ ⟨ l1, ..., li−1 ⟩ (in particular li is not the one 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.

   gap> g:=SymmetricGroup(4);;
   gap> GeneratorsSmallest(g);
   [ (3,4), (2,3), (1,2) ]

2 ▶ LargestElementGroup( G )
returns the largest element of G with respect to the ordering < of the elements family.

3 ▶ MinimalGeneratingSet( G )
returns a generating set of G of minimal possible length.

   gap> MinimalGeneratingSet(g);
   [ (2,4,3), (1,4,2,3) ]

4 ▶ SmallGeneratingSet( G )
returns a generating set of G which has few elements. As neither irredundancy, nor minimal length is proven it runs much faster than MinimalGeneratingSet. It can be used whenever a short generating set is desired which not necessarily needs to be optimal.

   gap> SmallGeneratingSet(g);
   [ (1,2), (1,2,3,4) ]

5 ▶ IndependentGeneratorsOfAbelianGroup( A )
returns a set of generators g of prime-power order of the abelian group A such that A is the direct product of the cyclic groups generated by the gi.

   gap> g:=AbelianGroup(IsPermGroup,[15,14,22,78]);;
   gap> List(IndependentGeneratorsOfAbelianGroup(g),Order);
   [ 2, 2, 2, 3, 3, 5, 7, 11, 13 ]

37.22 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

\[ Z^1(G/M, M) = \{ \gamma : G/M \to M \mid \forall g_1, g_2 \in G : \gamma(g_1 M \cdot g_2 M) = \gamma(g_1 M)^{g_2} \cdot \gamma(g_2 M) \} \]

and is a GF(p)-vector space.

The group of 1-coboundaries B^1(G/M, M) is defined as

\[ B^1(G/M, M) = \{ \gamma : G/M \to M \mid \exists m \in M \forall g \in G : \gamma(g M) = (m^{-1})^g \cdot m \} \]

It also is a GF(p)-vector space.

Let \( \alpha \) be the isomorphism of M into a row vector space \( W \) and \( (g_1, \ldots, g_l) \) representatives for a generating set of G/M. Then there exists a monomorphism \( \beta \) of \( Z^1(G/M, M) \) in the l-fold direct sum of \( W \), such that \( \beta(\gamma) = (\alpha(\gamma(g_1 M)), \ldots, \alpha(\gamma(g_l M))) \) for every \( \gamma \in Z^1(G/M, M) \).
Chapter 37. Groups

1. \texttt{OneCocycles( G, M )}
2. \texttt{OneCoboundaries( G, M )}

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 the DenominatorOfModuloPcgs(\( mpcgs \)) (see 43.1). 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 operations \texttt{OneCocycles} and \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{oneCocycles} and returns the corresponding complement to \( M \) in \( G \) (with respect to the fixed complement \texttt{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 \texttt{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 GAP but vector spaces.
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( oneCoboundaries := <vector space over GF(2), with 2 generators>,
    oneCocycles := <vector space over GF(2), with 2 generators>,
    generators := [ (3,4), (2,4,3) ], isSplitExtension := true,
    complement := Group([ (3,4), (2,4,3) ]),
    cocycleToList := function( c ) ... end,
    listToCocycle := function( L ) ... end,
    cocycleToComplement := function( c ) ... end,
    factorGens := [ (3,4), (2,4,3) ],
    complementToCocycle := function( K ) ... end )
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)]);
[ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]
gap> oc.cocycleToComplement([ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]);
Group([ (1,2), (1,2,3) ])
gap> oc.cocycleToComplement([ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]);
Group([ (3,4), (1,2,3) ])
gap> oc.complementToCocycle(Group((1,2,4),(1,4)));
[ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ]

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 \texttt{OCOneCocycles}.

3\textbf{ OCOneCocycles( ocr, onlySplit )}

is the more technical function to compute 1-cocycles. It takes an record \( ocr \) as first argument which must contain at least the components \texttt{group} for \( G \) and \texttt{modulePcgs} for a (modulo) pcgs of \( M \). This record will also be returned with components as described under \texttt{OneCocycles} (with the exception of \texttt{isSplitExtension} which is indicated by the existence of a \texttt{complement}) but components such as \texttt{oneCoboundaries} will only be computed if not already present.

If \texttt{onlySplit} is true, \texttt{OCOneCocyclesOC} returns \texttt{false} as soon as possible if the extension does not split.

4\textbf{ ComplementclassesEA( G, N )}

computes \texttt{Complementclasses} to an elementary abelian normal subgroup \( N \) via 1-Cohomology. Normally, a user program should call \texttt{Complementclasses} (see 37.10.6) instead, which also works for a solvable (not necessarily elementary abelian) \( N \).

5\textbf{ InfoCoh}

The info class for the cohomology calculations is \texttt{InfoCoh}.
37.23 Schur Covers and Multipliers

1. ▶ EpimorphismSchurCover( G[, pl] )

   returns an epimorphism $\text{epi}$ 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 of $\text{epi}$. The kernel of $\text{epi}$ 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.

   gap> g:=Group((1,2,3,4),(1,2));;
gap> epi:=EpimorphismSchurCover(g);
[ f1, f2, f3 ] -> [ (3,4), (2,4,3), (1,4)(2,3) ]
gap> Size(Source(epi));
48

   If the group becomes bigger, Schur Cover calculations might become unfeasible.

2. ▶ SchurCover( G )

   returns one (of possibly several) Schur covers of $G$.

   At the moment this cover is represented as a finitely presented group and IsomorphismPermGroup would be needed to convert it to a permutation group.

   gap> g:=Group((1,2,3,4),(1,2));;
gap> epi:=EpimorphismSchurCover(g);
[ f1, f2, f3 ] -> [ (3,4), (2,4,3), (1,4)(2,3) ]
gap> Size(Source(epi));
48

   If the group becomes bigger, Schur Cover calculations might become unfeasible.

   There is another operation which only returns the structure of the Multiplier, and which should work for larger groups as well.

3. ▶ AbelianInvariantsMultiplier( G )

   returns a list of the abelian invariants of the Schur multiplier of $G$.

   gap> AbelianInvariantsMultiplier(g);
[ 2 ]
gap> AbelianInvariantsMultiplier(MathieuGroup(22));
[ 4, 3 ]

   Note that the following example will take some time.

   gap> AbelianInvariantsMultiplier(PSU(6,2));
[ 2, 2, 3 ]

   At the moment, this operation will not give any information about how to extend the multiplier to a Schur Cover.

37.24 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).
Section 24. Tests for the Availability of Methods

1. **CanEasilyTestMembership( grp )**
   This filter indicates whether a group can test membership of elements in \( \text{grp} \) (via the operation \( \text{in} \)) in reasonable time. It is used by the method selection to decide whether an algorithm that relies on membership tests may be used.

2. **CanComputeSize( dom )**
   This filter indicates whether the size of the domain \( \text{dom} \) (which might be \text{infinity} \) can be computed.

3. **CanComputeSizeAnySubgroup( grp )**
   This filter indicates whether \( \text{grp} \) can easily compute the size of any subgroup. (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.)

4. **CanComputeIndex( \( G, H \) )**
   This filter indicates whether the index \([G : H]\) (which might be \text{infinity} \) can be computed. It assumes that \( H \leq G \). (see 37.24.5)

5. **CanComputeIsSubset( \( A, B \) )**
   This filter indicates that GAP can test (via \text{IsSubset} \) whether \( B \) is a subset of \( A \).

6. **KnowsHowToDecompose( \( G \) )**
   **KnowsHowToDecompose( \( G, \text{gens} \) )**
   Tests whether the group \( G \) can decompose elements in the generators \( \text{gens} \). If \( \text{gens} \) is not given it tests, whether it can decompose in the generators given in \text{GeneratorsOfGroup} \).
   This property can be used for example to check whether a \text{GroupHomomorphismByImages} \ can be reasonably defined from this group.
38 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, PreImage, PreImagesRepresentative, KernelOfMultiplicativeGeneralMapping, Source, Range, IsInjective and IsSurjective (see chapter 31, in particular section 31.8) are applicable to them.

Homomorphisms can be used to transfer calculations into isomorphic groups in another representation, for which better algorithms are available. Section 38.5 explains a technique how to enforce this automatically.

Homomorphisms are also used to represent group automorphisms, and section 38.6 explains GAP’s facilities to work with automorphism groups.

The penultimate section of this chapter, 38.9, explains how to make GAP to search for all homomorphisms between two groups which fulfill certain specifications.

38.1 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.

1 GroupHomomorphismByImages( G, H, gens, imgs )

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 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. (There also is the possibility to construct potentially multi-valued mappings with GroupGeneralMappingByImages and to test with IsMapping that they are indeed homomorphisms.)

2 GroupHomomorphismByImagesNC( G, H, gensG, gensH )

GroupHomomorphismByImagesNC creates a homomorphism as GroupHomomorphismByImages 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. Results are unpredictable if the conditions do not hold.

(For creating a possibly multi-valued mapping from G to H that respects multiplication and inverses, GroupGeneralMappingByImages can be used.)
Section 1. Creating Group Homomorphisms

```gap
gap> gens:=([1,2,3,4,1,2]);
gap> g:=Group(gens);
gap> h:=Group((1,2,3),(1,2));
gap> hom:=GroupHomomorphismByImages(g,h,gens,[1,2,3,4],[1,2]);
gap> Image(hom,(1,4));
gap> map:=GroupGeneralMappingByImages(g,h,gens,[1,2,3,4,1,2]);
gap> IsMapping(map);
```

returns a generalized mapping defined by extending the mapping from `gensG` to `gensH` homomorphically. (GroupHomomorphismByImages creates a GroupGeneralMappingByImages and tests whether it IsMapping.)

```gap
gap> hom:=GroupHomomorphismByImages(g,h,gens,[1,2,3,4],[1,2]);
gap> Image(hom,(1,4));
gap> map:=GroupGeneralMappingByImages(g,h,gens,[1,2,3,4,1,2]);
gap> IsMapping(map);
```

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 38.6).

```gap
gap> hom:=GroupHomomorphismByFunction(g,h,
  function(x) if SignPerm(x)=-1 then return (1,2); else return ();fi;end);
```

The third class are epimorphisms from a group onto its factor group. Such homomorphisms can be constructed by NaturalHomomorphismByNormalSubgroup (see 37.17.1).
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 39 and in particular section 39.6.1.

AsGroupGeneralMappingByImages(map)

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 38.10.2.)

gap> AsGroupGeneralMappingByImages(hom);
[ (1,2,3,4), (1,2) ] -> [ (1,2), (1,2) ]

38.2 Operations for Group Homomorphisms

Group homomorphisms are mappings, so all the operations and properties for mappings described in chapter 31 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 can be used to compute the kernel of a group homomorphism.

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.

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> hom:=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(hom); # 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

For an automorphism aut, Inverse 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,
Section 3. Efficiency of Homomorphisms

the operation `InverseGeneralMapping` should be used instead. (See 30.10.8 for a further discussion of this problem.)

```gap
> iso := IsomorphismPcGroup(g);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]) -> [ f1, f2, f3, f4 ]
> Inverse(iso);
#I The mapping must be bijective and have source=range
#I You might want to use 'InverseGeneralMapping'
fail
> InverseGeneralMapping(iso);
[ f1, f2, f3, f4 ] -> Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
```

38.3 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 41) or Pc groups (see 44) 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 `SetIsMapping`) (or to create it directly with `GroupHomomorphismByImagesNC`).

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.

Mappings given on generators (`GroupHomomorphismByImages`, `GroupGeneralMappingByImages`)

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`, see 37.24.6 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` should be used if the mapping is known to be a homomorphism.

Action homomorphisms (`ActionHomomorphism`)

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 `<` to compare elements of the domain is available.

Once the images of a generating set are computed, computing preimages (which is done via the `AsGroupGeneralMappingByImages`) and computing the kernel bahaves the same as for a `GroupHomomorphismByImages` in 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.

Mappings given by functions (`GroupHomomorphismByFunction`, `GroupGeneralMappingByFunctions`)

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.
Chapter 38. Group Homomorphisms

Testing whether a GroupGeneralMappingByFunctions is a homomorphism would require mapping all products of elements and thus should be avoided.

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 (respectively ImagesSource) 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 groups this can be simplified:

\begin{verbatim}
1 ImagesSmallestGenerators( map )
\end{verbatim}

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.)

38.4 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 PreImageRepresentative) 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) (otherwise the range is set to be the full symmetric group) However do not compute Range or Image, 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 by

\begin{verbatim}
function(p,g) return p^g;end;
\end{verbatim}

The action will be the same, but as the action function is not OnPoints, the extra processing for special cases is not triggered.
38.5 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 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 an ActionHomomorphism or a GroupHomomorphismByFunction (see also section 38.3).

1 ▶ IsHandledByNiceMonomorphism( obj )  
If this property is true, high-valued methods that translate all calculations in obj in the image under the NiceMonomorphism become available for obj.

2 ▶ NiceMonomorphism( obj )  
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.

3 ▶ NiceObject( obj )  
The NiceObject of obj is the image of obj under its NiceMonomorphism.

A typical example are finite matrix groups, which use a faithful action on vectors to translate all calculations in a permutation group.

```gap
    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)
```

4 ▶ IsCanonicalNiceMonomorphism( nhom )  
A NiceMonomorphism nhom is canonical if the image set will only depend on the set of group elements but not on the generating set and < comparison of group elements translates through the nice monomorphism. This implies that equal objects will always have equal NiceObjects. In some situations however this condition would be expensive to achieve, therefore it is not guaranteed for every nice monomorphism.

38.6 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.

1 ▶ ConjugatorIsomorphism( G, g )  
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 (see 38.6.2).
Chapter 38. Group Homomorphisms

2. ConjugatorAutomorphism\((G, g)\)  
   ConjugatorAutomorphismNC\((G, g)\)

   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 (see 38.6.1).

   ConjugatorAutomorphismNC does the same as ConjugatorAutomorphism, except that the check is omitted whether \(g\) normalizes \(G\).

3. InnerAutomorphism\((G, g)\)  
   InnerAutomorphismNC\((G, g)\)

   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 (see 38.6.1) or with ConjugatorAutomorphism (see 38.6.2).

   InnerAutomorphismNC does the same as InnerAutomorphism, except that the check is omitted whether \(g \in G\).

4. IsConjugatorIsomorphism\((\text{hom})\)  
   IsConjugatorAutomorphism\((\text{hom})\)  
   IsInnerAutomorphism\((\text{hom})\)

   Let \(\text{hom}\) be a group general mapping (see 31.8.4) with source \(G\), say. IsConjugatorIsomorphism returns **true** if \(\text{hom}\) 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 41), a matrix group (see Chapter 42), a finitely presented group (see Chapter 45), or a group given w.r.t. a polycyclic presentation (see Chapter 44). In all other cases, IsConjugatorIsomorphism may return **false** if \(\text{hom}\) is induced by conjugation but is not an inner automorphism.

   If IsConjugatorIsomorphism returns **true** for \(\text{hom}\) then an element \(g\) that induces \(\text{hom}\) can be accessed as value of the attribute ConjugatorOfConjugatorIsomorphism (see 38.6.5).

   IsConjugatorAutomorphism returns **true** if \(\text{hom}\) is an automorphism (see 31.12.3) that is regarded as a conjugator isomorphism by IsConjugatorIsomorphism, and **false** otherwise.

   IsInnerAutomorphism returns **true** if \(\text{hom}\) is a conjugator automorphism such that an element \(g\) inducing \(\text{hom}\) can be chosen in \(G\), and **false** otherwise.

5. ConjugatorOfConjugatorIsomorphism\((\text{hom})\)

   For a conjugator isomorphism \(\text{hom}\) (see 38.6.1), ConjugatorOfConjugatorIsomorphism returns an element \(g\) such that mapping under \(\text{hom}\) is induced by conjugation with \(g\).

   To avoid problems with IsInnerAutomorphism, it is guaranteed that the conjugator is taken from the source of \(\text{hom}\) if possible.
Section 7. Groups of Automorphisms

Group automorphism can be multiplied and inverted and thus it is possible to form groups of automorphisms.

1. `IsGroupOfAutomorphisms( G )` indicates whether \( G \) consists of automorphisms of another group \( H \). The group \( H \) can be obtained from \( G \) via the attribute `AutomorphismDomain`.

2. `AutomorphismDomain( G )` returns \( H \) if \( G \) consists of automorphisms of \( H \).

3. `AutomorphismGroup( obj )` returns the full automorphism group of the object \( obj \). The automorphisms act on the domain by the caret operator \( ^{\cdot} \). The automorphism group often stores a "NiceMonomorphism" (see 38.5.2) to a permutation group, obtained by the action on a subset of \( obj \).

4. `IsAutomorphismGroup( G )` indicates whether \( G \) is the full automorphism group of another group \( H \), this group is given as `AutomorphismDomain` of \( G \).

5. `InnerAutomorphismsAutomorphismGroup( autgroup )` For an automorphism group \( autgroup \) of a group this attribute stores the subgroup of inner automorphisms (automorphisms induced by conjugation) of the original group.
Chapter 38. Group Homomorphisms

Let \( \text{aut} \) be an automorphism of a group \( G \) and \( \text{epi}: G \to H \) an homomorphism such that \( \ker \text{epi} \) is fixed under \( \text{aut} \). Let \( U \) be the image of \( \text{epi} \). This command returns the automorphism of \( U \) induced by \( \text{aut} \) via \( \text{epi} \), that is the automorphism of \( U \) which maps \( g^{\text{epi}} \) to \((g^{\text{aut}})^{\text{epi}}, \) for \( g \in G \).

```gap
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));
^f2
\end{verbatim}

38.8 Calculating with Group Automorphisms

Usually the best way to calculate in a group of automorphisms is to go 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 a NiceMonomorphism (see 38.5.2.)

Once a group knows to be a group of automorphisms (this can be achieved by testing or setting the property IsGroupOfAutomorphisms (see 38.7.1), GAP will try itself to find such a nice monomorphism once calculations in the automorphism group are done.

```gap
1 AssignNiceMonomorphismAutomorphismGroup( autgrp, group )
computes a nice monomorphism for autgroup acting on group and stores it as NiceMonomorphism in autgrp.
If the centre of AutomorphismDomain of autgrp is trivial, the operation will first try to represent all automorphisms by conjugation (in group or 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 and does not return a value.
If a good domain for a faithful permutation action is known already, a homomorphism for the action on it can be created using NiceMonomorphismAutomGroup. It might be stored by SetNiceMonomorphism (see 38.5.2).

2 NiceMonomorphismAutomGroup( autgrp, elms, elmsgens )
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 groups original generators.) It does not yet assign it as NiceMonomorphism.
Another nice way of representing automorphisms as permutations has been described in [Sim97]. It it not yet available in GAP, a description however can be found in section 8.3 of “Extending GAP”.
```
Section 9. Searching for Homomorphisms

38.9 Searching for Homomorphisms

1. ▶ IsomorphismGroups( G, H )
   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 (see 48.7.5) or the random isomorphism test
   (see 44.10.1).

   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

2. ▶ GQuotients( F, G )
   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).

   gap> g:=Group((1,2,3,4),(1,2));
   Group([ (1,2,3,4), (1,2) ])
   gap> h:=Group((3,4),(1,2));;
   gap> quo:=GQuotients(g,h);
   [ [ (1,2,3,4), (1,2) ] -> [ (2,3), (1,2) ] ]

3. ▶ IsomorphicSubgroups( G, H )
   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 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).

   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) ] -> [ (3,4), (1,2) ],
     [ (3,4), (1,2) ] -> [ (1,3)(2,4), (1,2)(3,4) ] ]

4. ▶ MorClassLoop( range, classes, params, action )
   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.
Chapter 38. Group Homomorphisms

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 components

classes
  A 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 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 \([\text{word}, \text{order}]\) 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.

action is a number whose bit-representation indicates the requirements which are enforced on the element tuples found:

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 section V.5 of [Hul96].
38.10 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.

1. `IsGroupGeneralMappingByImages(map)`
   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 `MappingGeneratorImages` is used to store generators and their images.

2. `IsGroupGeneralMappingByAsGroupGeneralMappingByImages(map)`
   Representation for mappings that delegate work on a `GroupHomomorphismByImages`.

3. `IsPreimagesByAsGroupGeneralMappingByImages(map)`
   Representation for mappings that delegate work for preimages to a `GroupHomomorphismByImages`.

4. `IsPermGroupGeneralMappingByImages(map)`
   Is the representation for mappings that map from a perm group

5. `IsToPermGroupGeneralMappingByImages(map)`
   Is the representation for mappings that map to a perm group

6. `IsGroupGeneralMappingByPcgs(map)`
   Is the representation for mappings that map a pcgs to images and thus may use exponents to decompose generators.

7. `IsPcGroupGeneralMappingByImages(map)`
   Is the representation for mappings from a pc group

8. `IsToPcGroupGeneralMappingByImages(map)`
   Is the representation for mappings to a pc group

9. `IsFromFpGroupGeneralMappingByImages(map)`
   Is the representation of mappings from an fp group.

10. `IsFromFpGroupStdGensGeneralMappingByImages(map)`
    Is the representation of mappings from an fp group that give images of the standard generators.
39  Group Actions

A **group action** is a triple \((G, \Omega, \mu)\), where \(G\) is a group, \(\Omega\) a set and \(\mu: \Omega \times G \to \Omega\) a function (whose action is compatible with the group arithmetic). We call \(\Omega\) the **domain** of the action.

In **GAP**, \(\Omega\) 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 21.17.4).

The acting function \(\mu\) is a **GAP** function of the form

\[
\text{actfun}(\text{pnt}, g)
\]

that returns the image \(\mu(\text{pnt}, g)\) for a point \(\text{pnt} \in \Omega\) and a group element \(g \in G\).

Groups always acts from the right, that is \(\mu(\mu(\text{pnt}, g), h) = \mu(\text{pnt}, gh)\).

**GAP** does not test whether an acting function **actfun** 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, 39.1, 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 39.2.

The sections 39.6 and 39.7 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 39.11 describes the concept of “external sets” which represent the concept of a **G-set** and underly the actions mechanism.

### 39.1 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** for the orbits of the group \(G\) on the domain \(\Omega\) in the following ways:

\[
\text{OrbitsDomain}(G, \Omega[, \text{actfun}])
\]

The acting function **actfun** is optional. If it is not given, the built-in action **OnPoints** (which defines an action via the caret operator \(^{\cdot}\)) is used as a default.

\[
\text{OrbitsDomain}(G, \Omega, \text{gens[, acts[, actfun]}])
\]

This second version (of **OrbitsDomain**) permits one to implement an action induced by a homomorphism:

If \(H\) acts on \(\Omega\) via \(\varphi: G \to H\) is a homomorphism, \(G\) acts on \(\Omega\) via \(\mu'(\omega, g) = \mu(\omega, g^\varphi)\):

Here \(\text{gens}\) must be a set of generators of \(G\) and \(\text{acts}\) the images of \(\text{gens}\) under a homomorphism \(\varphi: G \to H\). **actfun** is the acting function for \(H\), the call to **ExampleActionFunction** implements the induced action of \(G\). Again, the acting function **actfun** is optional and **OnPoints** is used as a default.

The advantage of this notation is that **GAP** does not need to construct this homomorphism \(\varphi\) 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 \( \text{gens} \mapsto \text{acts} \) actually induces a homomorphism and the results are unpredictable if this is not the case.

\texttt{OrbitsDomain(\textit{extset}) \ A}

A third variant is to call the operation with an external set (which then provides \( G, \Omega \) and \( \text{actfun} \). You will find more about external sets in section 39.11.

For operations like \texttt{Stabilizer} of course the domain must be replaced by an element of \( \Omega \) which is to be acted on.

### 39.2 Basic Actions

GAP already provides acting functions for the more common actions of a group. For built-in operations such as \texttt{Stabilizer} special methods are available for many of these actions.

1. \texttt{OnPoints( \textit{pnt}, \textit{g} )}
   
   returns \( \textit{pnt} \cdot \textit{g} \). 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 \texttt{OnPoints} and \texttt{OnRight} (see 39.2.2).

2. \texttt{OnRight( \textit{pnt}, \textit{g} )}
   
   returns \( \textit{pnt} \ast \textit{g} \). 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 \texttt{OnPoints} (see 39.2.1) and \texttt{OnRight}.

3. \texttt{OnLeftInverse( \textit{pnt}, \textit{g} )}
   
   returns \( g^{-1} \ast \textit{pnt} \). Forming the inverse is necessary to make this a proper action, as in GAP groups always act from the right.

   (\texttt{OnLeftInverse} is used for example in the representation of a right coset as an external set (see 39.11), that is a right coset \( Ug \) is an external set for the group \( U \) acting on it via \texttt{OnLeftInverse}.)

4. \texttt{OnSets( \textit{set}, \textit{g} )}
   
   Let \( \textit{set} \) be a proper set (see 21.19). \texttt{OnSets} returns the proper set formed by the images \texttt{OnPoints( \textit{pnt}, \textit{g} )} of all points \( \textit{pnt} \) of \( \textit{set} \).

   \texttt{OnSets} is for example used to compute the action of a permutation group on blocks.

   (\texttt{OnTuples} is an action on lists that preserves the ordering of entries, see 39.2.5.)

5. \texttt{OnTuples( \textit{tup}, \textit{g} )}
   
   Let \( \textit{tup} \) be a list. \texttt{OnTuples} returns the list formed by the images \texttt{OnPoints( \textit{pnt}, \textit{g} )} for all points \( \textit{pnt} \) of \( \textit{tup} \).

   (\texttt{OnSets} is an action on lists that additionally sorts the entries of the result, see 39.2.4.)

6. \texttt{OnPairs( \textit{tup}, \textit{g} )}
   
   is a special case of \texttt{OnTuples} (see 39.2.5) for lists \( \textit{tup} \) of length 2.

7. \texttt{OnSetsSets( \textit{set}, \textit{g} )}
   
   Action on sets of sets; for the special case that the sets are pairwise disjoint, it is possible to use \texttt{OnSetsDisjointSets} (see 39.2.8).

8. \texttt{OnSetsDisjointSets( \textit{set}, \textit{g} )}
   
   Action on sets of pairwise disjoint sets (see also 39.2.7).
Chapter 39. Group Actions

9. OnSetsTuples( set, g )
Action on sets of tuples.

10. OnTuplesSets( set, g )
Action on tuples of sets.

11. OnTuplesTuples( set, g )
Action on tuples of tuples

```gap
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), (1,4,4), (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 ] ], [ [ 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 ] ] ]
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 ] ] ]
```

12. OnLines( vec, g )
Let vec be a normed row vector, that is, its first nonzero entry is normed to the identity of the relevant field, OnLines returns the row vector obtained from normalizing OnRight( vec, g ) by scalar multiplication from the left. This action corresponds to the projective action of a matrix group on 1-dimensional subspaces.

```gap
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) ])
```

13. OnIndeterminates( poly, perm )
A permutation perm acts on the multivariate polynomial poly by permuting the indeterminates as it permutes points.

14. Permutated( list, perm )
The following example demonstrates Permutated being used to implement a permutation action on a domain:
Section 3. Orbits

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" ] ]

15 ▶ OnSubspacesByCanonicalBasis( bas, mat )

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.)

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)

For example one could define the following function that acts on pairs of polynomials via OnInteremminates:

OnIndeterminatesPairs:=function(polypair,g)
  return [OnIndeterminates(polypair[1],g),
           OnIndeterminates(polypair[2],g)];
end;

39.3 Orbits

If G acts on Omega the set of all images of \( \omega \in \Omega \) under elements of G is called the orbit of \( \omega \). The set of orbits of G is a partition of Omega.

1 ▶ Orbit( G[, Omega[, pnt[, gens[, acts[, ]]]] act )

The orbit of the point pnt is the list of all images of pnt under the action.
(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.

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 39.2 for information about specific actions.)

2 ▶ Orbits( G, seeds[, gens, acts][, act] )

returns a duplicate-free list of the orbits of the elements in seeds under the action act of G
(Note that the arrangement of orbits or of points within one orbit is not defined by the operation.)

3 ▶ OrbitsDomain( G, Omega[, gens, acts][, act] )

returns a list of the orbits of G on the domain Omega (given as lists) under the action act.
This operation is often faster than Orbits. 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.)

```gap
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 ]], [ 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]],[[Z(2)^0,Z(2)^0],[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> ] ]
```

(See Section 39.2 for information about specific actions.)

4 ▶ OrbitLength( G, Omega, pnt, [gens, acts, ] act )
computes the length of the orbit of pnt.

5 ▶ OrbitLengths( G, seeds[, gens, acts][, act] )
6 ▶ OrbitLengthsDomain( G, Omega[, gens, acts][, act] )
computes the lengths of all the orbits of the elements in seeds under the action act of G.
computes the lengths of all the orbits of G on Omega.

This operation is often faster than OrbitLengths. The domain Omega must be closed under the action of G, otherwise an error can occur.

```gap
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 ]
```

39.4 Stabilizers

The Stabilizer of an element \( \omega \) is the set of all those \( g \in G \) which fix \( \omega \).

1 ▶ OrbitStabilizer( G, [Omega, ] pnt, [gens, acts, ] act )
computes the orbit and the stabilizer of pnt simultaneously in a single Orbit-Stabilizer algorithm.
The stabilizer must have G as its parent.

2 ▶ Stabilizer( G [, Omega], pnt [, gens, acts] [, act] )
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.
The stabilizer of a set or tuple of points can be computed by specifying an action of sets or tuples of points.

```gap
gap> g:=Group((1,3,2),(2,4,3));;
gap> Stabilizer(g,4);
Group([ (1,3,2) ])
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 39.2 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.

### 39.5 Elements with Prescribed Images

```gap
RepresentativeAction( G [, Omega], d, e [, gens, acts] [, act] )
```
Chapter 39. Group Actions

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 39.2 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.

39.6 The Permutation Image of an Action

If \( G \) acts on a domain \( \Omega \), an enumeration of \( \Omega \) yields a homomorphism of \( G \) into the symmetric group on \( \{1, \ldots, |\Omega|\} \). In GAP, the enumeration of the domain \( \Omega \) is provided by the Enumerator of \( \Omega \) (see 28.2.2) which of course is \( \Omega \) itself if it is a list.

\[ \text{ActionHomomorphism}( G, \Omega, [\text{gens}, \text{acts}] [, \text{act}] [, \text{"surjective"}] ) \]

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 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 (see 39.6.2).

(See Section 39.2 for information about specific actions.)

\[ \text{Action}( G, \Omega, [\text{gens}, \text{acts}] [, \text{act}] ) \]

returns the Image group of ActionHomomorphism 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 39.2 for information about specific actions.) The following code shows for example how to create the regular action of a group:

```gap
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) ])
```

The operation \( \text{SparseActionHomomorphism} \) computes the \( \text{ActionHomomorphism}(G,\text{dom},\text{gens},\text{acts},\text{act}) \), where \( \text{dom} \) is the union of the orbits \( \text{Orbit}(G,\text{pnt},\text{gens},\text{acts},\text{act}) \) for all points \( \text{pnt} \) from \( \text{start} \). If \( G \) acts on a very large domain \( \Omega \) not surjectively this may yield a permutation image of substantially smaller degree than by action on \( \Omega \).

The operation \( \text{SparseActionHomomorphism} \) will only use \( = \) comparisons of points in the orbit. Therefore it can be used even if no good \( < \) comparison method exists. However the image group will depend on the generators \( \text{gens} \) of \( G \).

The operation \( \text{SortedSparseActionHomomorphism} \) in contrast will sort the orbit and thus produce an image group which is not dependent on these generators.

```gap
gap> h:=Group(Z(3)*[[[1,1],[0,1]]]);
gap> hom:=ActionHomomorphism(h,GF(3)^2,OnRight);;
gap> Image(hom);
Group([ (2,3)(4,9,13,10,4)(2,8,14,11,5)(3,7,15,12,6),
       (1,7)(2,8)(3,9)(5,6)(10,11)(14,15) ])
gap> hom:=SparseActionHomomorphism(h,[Z(3)*[1,0]],OnRight);;
gap> Image(hom);
Group([ (1,2,3,4,5,6) ])
```

For an action homomorphism, the operation \( \text{UnderlyingExternalSet} \) will return the external set on \( \Omega \) which affords the action.

### 39.7 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 \( \text{ActionHomomorphism} \) 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:

1. \( \text{FactorCosetAction}(G, U, \lbrack N \rbrack) \)

This command computes the action of \( G \) on the right cosets of the subgroup \( U \). If the normal subgroup \( N \) is given, it is stored as kernel of this action.

```gap
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) ])
```

A special case is the regular action on all elements:
Chapter 39. Group Actions

2 ▶ RegularActionHomomorphism( G )
returns an isomorphism from G onto the regular permutation representation of G.

3 ▶ AbelianSubfactorAction( G, M, N )
Let G be a group and M ≥ 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 might be faster.

\begin{verbatim}
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);;
[ [ 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 )
    2 ), function( e ) ... end, function( r ) ... end ),
  Pcgs([ (2,8,3,9)(4,10,5,11), (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)]);
(2,8,3,9)(4,10,5,11)

gap> e in c[3]; e in c[4];
true false
\end{verbatim}

39.8 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.

1 ▶ Permutation( g, Omega [, gens, acts][, act] )

\begin{verbatim}
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 of this external set (see Section 39.11). 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 fail is returned.
\end{verbatim}

2 ▶ PermutationCycle( g, Omega, pnt [, act] )

\begin{verbatim}
computes the permutation that represents the cycle of pnt under the action of the element g.
\end{verbatim}
Tests for Actions

3 ▶ Cycle(  g, Omega, pnt [, act] )
returns a list of the points in the cycle of pnt under the action of the element g.

4 ▶ CycleLength(  g, Omega, pnt [, act] )
returns the length of the cycle of pnt under the action of the element g.

5 ▶ Cycles(  g, Omega [, act] )
returns a list of the cycles (as lists of points) of the action of the element g.

6 ▶ CycleLengths(  g, Omega, [, act] )
returns the lengths of all the cycles under the action of the element g on Omega.

39.9 Tests for Actions

1 ▶ IsTransitive(  G, Omega[, gens, acts][, act] )
  IsTransitive(  xset )
returns true if the action implied by the arguments is transitive, or false otherwise.
An action is transitive if the whole domain forms one orbit.

2 ▶ Transitivity(  G, Omega[, gens, acts][, act] )
  Transitivity(  xset )
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.

Note: 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 \( \langle (2,3), (2,3) \rangle \) is transitive (on 3 points).
Chapter 39. Group Actions

3► RankAction( G, Omega[, gens, acts][, act] )
4► RankAction( xset )

returns the rank of a transitive action, i.e. the number of orbits of the point stabilizer.

\[
\text{gap> RankAction(} g, \text{Combinations([1..4],2),OnSets);}
\]
\[
4
\]

4► IsSemiRegular( G, Omega[, gens, acts][, act] )
5► IsSemiRegular( xset )

returns true if the action implied by the arguments is semiregular, or false otherwise.

An action is semiregular if the stabilizer of each point is the identity.

\[
\text{gap> IsSemiRegular(} g, \text{Arrangements([1..4],3),OnTuples);}
\]
\[
\text{true}
\]

\[
\text{gap> IsRegular(} g, \text{Arrangements([1..4],3),OnTuples);}
\]
\[
\text{false}
\]

6► Earns( G, Omega[, gens, acts][, act] )
7► Earns( xset )

returns a list of the elementary abelian regular (when acting on Omega) normal subgroups of G.

8► IsPrimitive( G, Omega[, gens, acts][, act] )
9► IsPrimitive( xset )

returns true if the action implied by the arguments is primitive, or false otherwise.

An action is primitive if it is both semiregular (see 39.9.4) and transitive (see 39.9.1). In this case every point pnt of Omega defines a one-to-one correspondence between G and Omega.

\[
\text{gap> IsPrimitive(} g, \text{Orbit(g,(1,2)(3,4)));}
\]
\[
\text{true}
\]

39.10 Block Systems

A block system (system of imprimitivity) for the action of G on Omega is a partition of Omega which – as a partition – remains invariant under the action of G.

1► Blocks( G, Omega[, seed][, gens, acts][, act] )
2► Blocks( xset[, seed] )

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.
Section 10. Block Systems

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 39.2 for information about specific actions.)

2 ▶ MaximalBlocks( G, Omega [, seed] [, gens, acts] [, act] )
  ▶ MaximalBlocks( xset [, seed] )

returns a block system that is maximal with respect to inclusion. maximal with respect to inclusion) for the
action of G on Omega. If seed is given, a block system in which seed is the subset of one block is computed.

gap> MaximalBlocks(g,[1..8]);
[ [ 1, 2, 3, 8 ], [ 4, 5, 6, 7 ] ]

3 ▶ RepresentativesMinimalBlocks( G, Omega[, gens, acts][, act] )
  ▶ RepresentativesMinimalBlocks( xset )

computes a list of block representatives for all minimal (i.e. blocks are minimal with respect to inclusion) nontrivial block systems for the action.

gap> RepresentativesMinimalBlocks(g,[1..8]);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 1, 7 ], [ 1, 8 ] ]

4 ▶ AllBlocks( G )

computes a list of representatives of all block systems for a permutation group G acting transitively on the
points moved by the group.

gap> AllBlocks(g);
[ [ 1, 8 ], [ 1, 2, 3, 8 ], [ 1, 4, 5, 8 ], [ 1, 6, 7, 8 ], [ 1, 3 ],
  [ 1, 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 (see 39.2.4):

gap> Stabilizer(g,[1,8],OnSets);
Group([ (1,8)(2,3)(4,5)(6,7) ])

If bs is a partition of omega, given as a set of sets, the stabilizer under the action OnSetsDisjointSets
(see 39.2.8) returns the largest subgroup which preserves bs as a block system.

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) ])
39.11 External Sets

When considering group actions, sometimes the concept of a $G$-set is used. This is the set $\Omega$ endowed with an action of $G$. The elements of the $G$-set are the same as those of $\Omega$, however concepts like equality and equivalence of $G$-sets do not only consider the underlying domain $\Omega$ but the group action as well.

This concept is implemented in GAP via external sets.

An external set specifies an action $\text{act} : \Omega \times G \to \Omega$ of a group $G$ on a domain $\Omega$. The external set knows the group, the domain and the actual acting function. Mathematically, an external set is the set $\Omega$, which is endowed with the action of a group $G$ via the group action $\text{act}$. For this reason GAP treats external sets as a domain whose elements are the elements of $\Omega$. An external set is always a union of orbits. Currently the domain $\Omega$ must always be finite. If $\Omega$ is not a list, an enumerator for $\Omega$ is automatically chosen.

**Example**

```gap
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: <enumerator of perm group>>
gap> Orbits(e);
[ [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3), (2,3,4), (1,2,4), (1,3,2), (1,4,3), (2,4,3), (1,2,3), (1,3,4), (2,3,4), (1,3,2), (1,4,3), (1,2,4) ] ]
```

The following three attributes of an external set hold its constituents.

**Example**

```gap
gap> ActingDomain(e);
Group([ (1,2,3), (2,3,4) ])
gap> FunctionAction(e)=OnRight;
true
gap> HomeEnumerator(e);
[ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3), (2,3,4), (1,2,4), (1,3,2), (1,4,3), (2,4,3), (1,2,3), (1,3,4), (1,4,2) ]
```

Most operations for actions are applicable as an attribute for an external set.
Section 11. External Sets

6 ▶ IsExternalSubset( obj )

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.

The most prominent external subsets are orbits:

7 ▶ ExternalSubset( G, xset, start, [gens, acts, ] act )

color=O
constructs the external subset of xset on the union of orbits of the points in start.

8 ▶ IsExternalOrbit( obj )

An external orbit is an external subset consisting of one orbit.

9 ▶ ExternalOrbit( G, Omega, pnt, [gens, acts, ] act )

color=O
constructs the external subset on the orbit of pnt. The Representative of this external set is pnt.

gap> e:=ExternalOrbit(g,g,(1,2,3));
(1,2,3)^G

Many subsets of a group, such as conjugacy classes or cosets (see 37.9.1 and 37.6.1) are implemented as external orbits.

10 ▶ StabilizerOfExternalSet( xset )

computes the stabilizer of Representative(xset) The stabilizer must have the acting group G of xset as its parent.

gap> Representative(e);
(1,2,3)
gap> StabilizerOfExternalSet(e);
Group([ (1,2,3) ])

11 ▶ ExternalOrbits( G, Omega[, gens, acts][, act] )
▶ ExternalOrbits( xset )

computes a list of ExternalOrbits that give the orbits of G.

gap> ExternalOrbits(g,AsList(g));
[ ()^G, (2,3,4)^G, (2,4,3)^G, (1,2)(3,4)^G ]

12 ▶ ExternalOrbitsStabilizers( G, Omega[, gens, acts][, act] )
▶ ExternalOrbitsStabilizers( xset )

In addition to ExternalOrbits, 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 first and the stabilizers afterwards.)

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) ])

13 ▶ CanonicalRepresentativeOfExternalSet( xset )

The canonical representative of an external set may only depend on G, Omega, act 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 every external set to compute a canonical representative . See 39.11.14.
returns a function that takes as arguments the acting group and the point. It returns a list of length 3: [canonrep, stabilizer, canonrep, conjugating elm]. (List components 2 and 3 are optional and do not need to be bound.) An external set is only guaranteed to be able to compute a canonical representative if it has a CanonicalRepresentativeDeterminatorOfExternalSet.

returns an element mapping Representative(xset) to CanonicalRepresentativeOfExternalSet(xset) under the given action.

 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)

External sets also are implicitly underlying action homomorphisms:

The underlying set of an action homomorphism is the external set on which it was defined.

 gap> g:=Group((1,2,3),(1,2));;
 gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples);;
 gap> UnderlyingExternalSet(hom);
 [ [ 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 ] ]

returns an action homomorphism for xset which is surjective. (As the Image of this homomorphism has to be computed to obtain the range, this may take substantially longer than ActionHomomorphism.)

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 and “Operations for Xyz”.

The old names still exist as synonyms, however we do not guarantee that they will remain indefinitely in the system and strongly discourage their further use. They are listed here mainly as a help to users who are still accustomed to the old names:

Obsolete synonym, see 39.5.1.

Obsolete synonym, see 39.6.2.
Section 12. Legacy Operations

3. OperationHomomorphism(args)
   Obsolete synonym, see 39.6.1.

4. FunctionOperation(args)
   Obsolete synonym, see 39.11.4.
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 less than $2^{28} - 1$, and the image of a point $i$ under a permutation $p$ is written $i^p$, which is expressed as $i^p$ in GAP. (This action is also implemented by the function `OnPoints`, see 39.2.1.) If $i^p \neq i$, 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 30.12.

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 called **IsPerm**.

```gap
1 ▶ IsPerm( obj ) C
2 ▶ IsPermCollection( obj ) C
  ▶ IsPermCol1Coll( obj ) C
```

are the categories for collections of permutations and collections of collections of permutations, respectively.

3 ▶ **PermutationsFamily** V

is the family of all permutations.

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 either all stored as 16-bit integers or all as 32-bit integers (actually as GAP immediate integers less than $2^{28}$), depending on whether $d \leq 65536$ or not. This means that the identity permutation $(\cdot)$ takes $4m$ bytes if it was calculated as $(1, \ldots, m) \cdot (1, \ldots, m)^{-1}$. It can take even more because the internal list has sometimes room for more than $d$ images. For example, the maximal degree of any permutation in GAP is $m = 2^{22} - 1024 = 4,193,280$, because bigger permutations would have an internal list with room for more than $2^{22}$ images, requiring more than $2^{24}$ bytes. $2^{24}$, however, is the largest possible size of an object that the GAP storage manager can deal with.

Permutations do not belong to a specific group. That means that one can work with permutations without defining a permutation group that contains them.

```gap
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** (see 21.20.15) can be used to permute the entries of a list according to a permutation.
Section 2. Moved Points of Permutations

40.1 Comparison of Permutations

Two permutations are equal if they move the same points and all these points have the same images under both permutations.

The permutation $p_1$ is smaller than $p_2$ if $p_1 \neq p_2$ and $i^{p_1} < i^{p_2}$ where $i$ is the smallest point with $i^{p_1} \neq i^{p_2}$. Therefore the identity permutation is the smallest permutation. (see also 30.11)

Permutations can be compared with certain other GAP objects, see 4.11 for the details.

\[
gap> (1,2,3) = (2,3,1); \quad \text{true}
\]
\[
gap> (1,2,3) * (2,3,4) = (1,3)(2,4); \quad \text{true}
\]
\[
gap> (1,2,3) < (1,3,2); \quad \# 1^{(1,3,2)} = 2 < 3 = 1^{(1,2,3)} \quad \text{true}
\]
\[
gap> (1,3,2,4) < (1,3,4,2); \quad \# 2^{(1,3,2,4)} = 4 > 1 = 2^{(1,3,4,2)} \quad \text{false}
\]

\[\text{SmallestGeneratorPerm( } perm \text{ )} \]

is the smallest permutation that generates the same cyclic group as the permutation $perm$. This is very efficient, even when $perm$ has large order.

\[
gap> \text{SmallestGeneratorPerm}( (1,4,3,2) ); \quad (1,2,3,4)
\]

40.2 Moved Points of Permutations

\[\text{SmallestMovedPoint( } perm \text{ )} \quad \text{A}
\]
\[\text{SmallestMovedPoint( } C \text{ )} \quad \text{A}
\]

is the smallest positive integer that is moved by $perm$ if such an integer exists, and $\infty$ if $perm = ()$. For $C$ a collection or list of permutations, the smallest value of \text{SmallestMovedPoint} for the elements of $C$ is returned (and $\infty$ if $C$ is empty).

\[\text{LargestMovedPoint( } perm \text{ )} \quad \text{A}
\]
\[\text{LargestMovedPoint( } C \text{ )} \quad \text{A}
\]

For a permutation $perm$, this attribute contains the largest positive integer which is moved by $perm$ if such an integer exists, and 0 if $perm = ()$. For $C$ a collection or list of permutations, the largest value of \text{LargestMovedPoint} for the elements of $C$ is returned (and 0 if $C$ is empty).

\[\text{MovedPoints( } perm \text{ )} \quad \text{A}
\]
\[\text{MovedPoints( } C \text{ )} \quad \text{A}
\]

is the proper set of the positive integers moved by at least one permutation in the collection $C$, respectively by the permutation $perm$.

\[\text{NrMovedPoints( } perm \text{ )} \quad \text{A}
\]
\[\text{NrMovedPoints( } C \text{ )} \quad \text{A}
\]

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 \text{MovedPoints}, see 40.2.3)
Chapter 40. Permutations

40.3 Sign and Cycle Structure

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.

is the cycle structure (i.e. the numbers of cycles of different lengths) of \( perm \). 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.

40.4 Creating Permutations

is a list \( list \) that contains the images of the positive integers under the permutation \( perm \). That means that \( list[i] = i^\text{perm} \), where \( i \) lies between 1 and the largest point moved by \( perm \) (see 40.2.2).

is the permutation \( perm \) that moves points as described by the list \( list \). That means that \( i^\text{perm} = list[i] \) if \( i \) lies between 1 and the length of \( list \), and \( i^\text{perm} = i \) if \( i \) is larger than the length of the list \( list \). It will signal an error if \( list \) does not define a permutation, i.e., if \( list \) is not a list of integers without holes, or if \( list \) contains an integer twice, or if \( list \) contains an integer not in the range \([1..\text{Length}(list)]\).

Let \( src \) and \( dst \) be lists of positive integers of the same length, such that neither may contain an element twice. \( \text{MappingPermListList} \) returns a permutation \( perm \) such that \( src[i]^\text{perm} = dst[i] \). \( perm \) 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 \( \text{MappingPermListList} \) returns.
Section 4. Creating Permutations

4. RestrictedPerm( perm, list )

RestrictedPerm returns the new permutation new 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. 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.

```gap
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)
```
A permutation group is a group of permutations on a finite set $\Omega$ of positive integers. GAP does not require the user to specify the operation domain $\Omega$ when a permutation group is defined.

```gap
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 37) can be applied to them. In many cases special methods are installed for permutation groups that make computations more effective.

### 41.1 The Natural Action

The functions `MovedPoints`, `NrMovedPoints`, `LargestMovedPoint`, and `SmallestMovedPoint` are defined for arbitrary collections of permutations (see 40.2), in particular they can be applied to permutation groups.

```gap
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`). Therefore all action functions can be applied (see the Chapter 39), for example `Orbit`, `Stabilizer`, `Blocks`, `IsTransitive`, `IsPrimitive`.

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 $\cdot$, the following functions are provided for this purpose.

```gap
1 ▶ OrbitPerms( perms, pnt )
returns the orbit of the positive integer `pnt` under the group generated by the permutations in the list `perms`.

2 ▶ OrbitsPerms( perms, D )
returns the list of orbits of the positive integers in the list `D` under the group generated by the permutations in the list `perms`.

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 ] ]
```

Similarly, several functions concerning the natural action of permutation groups address stabilizer chains (see 41.5) rather than permutation groups themselves, for example `BaseStabChain` (see 41.9.1).
Section 3. Symmetric and Alternating Groups

41.2 Computing a Permutation Representation

\[ \text{IsomorphismPermGroup}( \ G \ ) \]
returns an isomorphism \( \varphi \) from the group \( G \) onto a permutation group \( P \) which is isomorphic to \( G \). The method will select a suitable permutation representation.

\[
g := \text{SmallGroup}(24,12);
gap> iso := \text{IsomorphismPermGroup}(g);
gap> \text{Image}(iso,g.3*g.4);
\]

\[ (1,4)(2,3)(5,8)(6,7)(9,12)(10,11)(13,16)(14,15)(17,20)(18,19)(21,24)(22,23) \]

In many cases the permutation representation constructed by \text{IsomorphismPermGroup} is regular.

\[ \text{SmallerDegreePermutationRepresentation}( \ G \ ) \]
Let \( G \) be a permutation group that acts transitively on its moved points. \text{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 \( G \).

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 \text{SmallerDegreePermutationRepresentation}.

\[
gap> image := \text{Image}(iso);\gap\
24
\]
\[
gap> small := \text{SmallerDegreePermutationRepresentation}(image);\gap\
\gap> \text{Image}(small);
\]
\[ \text{Group}([ (1,4)(2,3), (1,3)(2,4), (2,3), (2,3,4) ]) \]

41.3 Symmetric and Alternating Groups

The commands \text{SymmetricGroup} and \text{AlternatingGroup} (see 48.1) 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 \text{natural} symmetric or alternating group, respectively.

\[ \text{IsNaturalSymmetricGroup}( \ group \ ) \]
A group is a natural symmetric group if it is a permutation group acting as symmetric group on its moved points.

\[ \text{IsNaturalAlternatingGroup}( \ group \ ) \]
A group is a natural alternating group if it is a permutation group acting as alternating group 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.
The following functions can be used to check whether a given group (not necessarily a permutation group) is isomorphic to a symmetric or alternating group.

There are no methods yet for IsSymmetricGroup and IsAlternatingGroup!

3► IsSymmetricGroup( group )

is true if the group group is isomorphic to a natural symmetric group.

4► IsAlternatingGroup( group )

Such a group is a group isomorphic to a natural alternating group.

5► SymmetricParentGroup( grp )

For a permutation group grp this function returns the symmetric group that moves the same points as grp does.

gap> SymmetricParentGroup( Group( (1,2), (4,5), (7,8,9) ) );
Sym( [ 1, 2, 4, 5, 7, 8, 9 ] )

41.4 Primitive Groups

1► ONanScottType( G )

returns the type of G 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:

1 Affine.
2 Almost simple.
3a Diagonal, Socle consists of two normal subgroups.
3b Diagonal, Socle is minimal normal.
4a Product action with the first factor primitive of type 3a.
4b Product action with the first factor primitive of type 3b.
4c Product action with the first factor primitive of type 2.
5 Twisted wreath product

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.

2► SocleTypePrimitiveGroup( G )

returns the socle type of a primitive permutation group. 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, see 37.14.11) and width for the number of direct factors.

If G does not have a faithful primitive action, the result is undefined.
41.5 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 8.1 in “Extending GAP”.

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 length 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 of the form $g = r_n \cdots r_1$ 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 $r T[p]^{-1}$ 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 41.7, we explain in 41.6 the idea of using non-deterministic algorithms; this is necessary for understanding the options available for the construction of stabilizer chains. After that, in 41.8 it is explained how a stabilizer chain is stored in GAP, 41.9 lists operations for stabilizer chains, and 41.10 lists low level routines for manipulating stabilizer chains.

### 41.6 Randomized Methods for Permutation Groups

For most computations with permutation groups, it is crucial to construct stabilizer chains efficiently. Sims’s original construction [Sim70] is deterministic, and is called the Schreier-Sims algorithm, because it is based on Schreier’s Lemma (p. 96 in [Hal59]): 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. 41.7).

If `random = 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 $1 \leq x \leq 999$ 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 $R(m)R(m-1)\ldots R(1)$ 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` command in 41.7.1).

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= 1000` (i.e., guaranteed correct output) is about the same as the running time of randomized verification with guarantee of at least 90% correctness. Therefore, we suggest to use the default value `random= 1000`. Possible uses of `random< 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 illustrate these ideas in two examples.

```gap
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. (The `StabChain` command is described in 41.7.1.)
Now randomized verification will be used. We require that the result is guaranteed correct with probability 90%. 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, see 41.7.1.)

First the group is created anew.

```gap
gap> h:= Group( GeneratorsOfGroup( op ) );;
gap> StabChain( h );; time;
1120

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.

```gap
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 = 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 $O^{(i)}$.

```gap
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 $< 1000$, this is noted in the group $G$, by setting of the record component random in the value of the attribute StabChainOptions for $G$ (see 41.7.2). As errors induced by the random methods might propagate, any group or homomorphism created from $G$ inherits a random component in its StabChainOptions 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.
41.7 Construction of Stabilizer Chains

These commands compute a stabilizer chain for the permutation group \( G \); additionally, \texttt{StabChainMutable} is also an attribute for the group homomorphism \( \text{permhomom} \) whose source is a permutation group.

\texttt{StabChainOp} is an operation with two arguments \( G \) and \( \text{options} \), the latter being a record which controls some aspects of the computation of a stabilizer chain (see below); \texttt{StabChainOp} returns a \texttt{mutable} stabilizer chain. \texttt{StabChainMutable} is a \texttt{mutable} attribute for groups or homomorphisms, its default method for groups is to call \texttt{StabChainOp} with empty \texttt{options} record. \texttt{StabChainImmutable} is an attribute with \texttt{immutable} values; its default method dispatches to \texttt{StabChainMutable}.

\texttt{StabChain} is a function with first argument a permutation group \( G \), and optionally a record \texttt{options} as second argument. If the value of \texttt{StabChainImmutable} for \( G \) is already known and if this stabilizer chain matches the requirements of \texttt{options}, \texttt{StabChain} simply returns this stored stabilizer chain. Otherwise \texttt{StabChain} calls \texttt{StabChainOp} and returns an immutable copy of the result; additionally, this chain is stored as \texttt{StabChainImmutable} value for \( G \). If no \texttt{options} argument is given, its components default to the global variable \texttt{DefaultStabChainOptions} (see 41.7.3). If \texttt{base} is a list of positive integers, the version \texttt{StabChain( G, base \text{ record(base:= base) })}.

If given, \texttt{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 \texttt{DefaultStabChainOptions} (see 41.7.3). The following options are supported.

\begin{itemize}
  \item \texttt{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 \texttt{reduced} component of \texttt{options} is \texttt{true} then those points of \texttt{base} with nontrivial basic orbits form the initial segment of \( B \), if the \texttt{reduced} component is \texttt{false} then \texttt{base} itself is the initial segment of \( B \). Repeated occurrences of points in \texttt{base} are ignored. If a stabilizer chain for \( G \) is already known then the stabilizer chain is computed via a base change.
  
  \item \texttt{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 \texttt{knownBase} bears no relation to the \texttt{base} component, you may specify a known base \texttt{knownBase} and a desired base \texttt{base} independently.
  
  \item \texttt{reduced} (default \texttt{true})
    If this is \texttt{true} the resulting stabilizer chain \( S \) is reduced, i.e., the case \( G^{(i)} = G^{(i+1)} \) does not occur. Setting \texttt{reduced} to \texttt{false} makes sense only if the component \texttt{base} (see above) is also set; in this case all points of \texttt{base} will occur in the base \( B \) of \( S \), even if they have trivial basic orbits. Note that if \texttt{base} is just an initial segment of \( B \), the basic orbits of the points in \( B \setminus \text{base} \) are always nontrivial.
  
  \item \texttt{tryPcgs} (default \texttt{true})
    If this is \texttt{true} and either the degree is at most 100 or the group is known to be solvable, \texttt{GAP} will first try to construct a pcgs (see Chapter 43) for \( G \) which will succeed and implicitly construct a
Section 8. Stabilizer Chain Records

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.

\[ \text{random (default 1000)} \]

If the value is less than 1000, the resulting chain is correct with probability at least \( \text{random}/1000 \). The random option is explained in more detail in 41.6.

\[ \text{size (default Size( } G \text{ ) if this is known, i.e., if HasSize( } G \text{ ) is true)} \]

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, \( \text{GAP} \) does a non-deterministic construction until the size is correct.

\[ \text{limit (default Size( Parent( } G \text{ ) ) or StabChainOptions( Parent( } G \text{ ) ).limit if this is present)} \]

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.

2 ▶ StabChainOptions( \( G \) )

is a record that stores the options with which the stabilizer chain stored in StabChainImmutable has been computed (see 41.7.1 for the options that are supported).

3 ▶ DefaultStabChainOptions

are the options for StabChain which are set as default.

4 ▶ StabChainBaseStrongGenerators( \( base \), \( sgs \), \( one \) )

If a base \( base \) for a permutation group \( G \) and a strong generating set \( sgs \) for \( G \) with respect to \( base \) are given, \( one \) must be the appropriate One (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.

5 ▶ MinimalStabChain( \( G \) )

returns the reduced stabilizer chain corresponding to the base \([1, 2, 3, 4, . . .] \).

41.8 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

de the identity element of the current stabilizer.

labels

ea 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 is this list is always the identity, for reasons explained below under translabels. Note that \( \text{GAP} \) tries to arrange things so that the labels components are identical (i.e., the same \( \text{GAP} \) 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).
Chapter 41. Permutation Groups

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 \( b_i^G \), ordered in such a way that the base point \( b_i \) is first in the list.

translabels

The factorized inverse transversal that was found during the orbit algorithm carried out with the inverses of labels{ genlabels }, starting from the base point. The directed edge from the point \( i \) to the point \( j = i^g \) in the Schreier tree is represented by labels[ translabels[ j ] ] = \( g \) in the stabilizer chain (note that \( i \) can be reconstructed as \( jg^{-1} \) from this information). The entries in the translabels component are thus just integers giving the positions of the appropriate labels in the labels component. The base point itself (i.e., the root of the Schreier tree) corresponds to the entry 1 in translabels. This makes it possible to assign transversal{ orbit } := labels{ translabels{ orbit } } (see 21.4.2 in the Reference Manual and transversal below). The translabels component pays off handsomely in the stabilizer chain conjugation (see 8.1 in “Extending GAP”).

transversal

Is another representation of the factorized inverse transversal: as a list with transversal[ j ] = labels[ translabels[ j ] ] (the base point itself has entry 0). To look up an entry in this list is faster than to evaluate the expression on the right hand side, and since this operation appears most often in permutation group code (e.g., in the function InverseRepresentative below), the speed-up is really worth this extra record component.

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 possibly 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.

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( reduced := true, random := 1000, tryPcgs := true )
Section 9. Operations for Stabilizer Chains

41.9 Operations for Stabilizer Chains

1. BaseStabChain( S )
   returns the base belonging to the stabilizer chain S.

2. BaseOfGroup( G )
   returns a base of the permutation group G. There is no guarantee that a stabilizer chain stored in G corresponds to this base!

3. SizeStabChain( S )
   returns the product of the orbit lengths in the stabilizer chain S, that is, the order of the group described by S.

4. StrongGeneratorsStabChain( S )
   returns a strong generating set corresponding to the stabilizer chain S.

5. GroupStabChain( [G, ] S )
   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.

6. OrbitStabChain( S, pnt )
   returns the orbit of pnt under the group described by the stabilizer chain S.

7. IndicesStabChain( S )
   returns a list of the indices of the stabilizers in the stabilizer chain S.

8. ListStabChain( S )
   returns a list that contains at position i the stabilizer of the first i – 1 base points in the stabilizer chain S.

9. ElementsStabChain( S )
   returns a list of all elements of the group described by the stabilizer chain S.

10. InverseRepresentative( S, pnt )
    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.

11. SiftedPermutation( S, g )
    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.

12. MinimalElementCosetStabChain( S, g )
    Let G be the group described by the stabilizer chain S. This function returns a permutation h such that Gg = G h (that is, g/h ∈ 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.
Chapter 41. Permutation Groups

13 ▶ LargestElementStabChain( S, id )

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).

14 ▶ ApproximateSuborbitsStabilizerPermGroup( G, pnt )

returns an approximation of the orbits of \( \text{Stabilizer}( G, \text{pnt} ) \) on all points of the orbit \( \text{Orbit}( G, \text{pnt} ) \), 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.

41.10 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.)

1 ▶ CopyStabChain( S )

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.

2 ▶ CopyOptionsDefaults( G, options )

sets components in a stabilizer chain options record \( \text{options} \) according to what is known about the group \( G \). This can be used to obtain a new stabilizer chain for \( G \) quickly.

3 ▶ ChangeStabChain( S, base [, reduced] )

changes or reduces a stabilizer chain \( S \) to be adapted to the base \( \text{base} \). The optional argument \( \text{reduced} \) is interpreted as follows.

\[ \text{reduced} = \text{false} : \]
\[ \quad \text{change the stabilizer chain, do not reduce it,} \]
\[ \text{reduced} = \text{true} : \]
\[ \quad \text{change the stabilizer chain, reduce it.} \]

4 ▶ ExtendStabChain( S, base )

extends the stabilizer chain \( S \) so that it corresponds to base \( \text{base} \). The original base of \( S \) must be a subset of \( \text{base} \).

5 ▶ ReduceStabChain( S )

changes the stabilizer chain \( S \) to a reduced stabilizer chain by eliminating trivial steps.

6 ▶ RemoveStabChain( S )

\( S \) 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.

7 ▶ EmptyStabChain( labels, id [, pnt] )

constructs a stabilizer chain for the trivial group with \( \text{identity} = \text{id} \) and \( \text{labels} = \{ \text{id} \} \cup \text{labels} \) (but of course with \( \text{genlabels} = [ ] \) and \( \text{generators} = [ ] \)). If the optional third argument \( \text{pnt} \) is present, the only stabilizer
of the chain is initialized with the one-point basic orbit \([ \text{pnt} ]\) and with \text{translabels} and \text{transversal} components.

8 ▶ InsertTrivialStabilizer( \(S, \text{pnt}\) )

\text{InsertTrivialStabilizer} initializes the current stabilizer with \text{pnt} as \text{EmptyStabChain} did, but assigns the original \(S\) to the new \(S.\text{stabilizer}\) component, such that a new level with trivial basic orbit (but identical \text{labels} and \text{ShallowCopied genlabels and generators}) is inserted. This function should be used only if \text{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 \text{AddGeneratorsExtendSchreierTree}.

9 ▶ IsFixedStabilizer( \(S, \text{pnt}\) )

returns \text{true} if \text{pnt} is fixed by all generators of \(S\) and \text{false} otherwise.

10 ▶ AddGeneratorsExtendSchreierTree( \(S, \text{new}\) )

adds the elements in \text{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).

### 41.11 Backtrack

A main use for stabilizer chains is in backtrack algorithms for permutation groups. \text{GAP} implements a partition-backtrack algorithm as described in [Leo91] and refined in [The97].

1 ▶ SubgroupProperty( \(G, \text{Pr}[\text{, } L]\) )

\text{Pr} must be a one-argument function that returns \text{true} or \text{false} for elements of \(G\) and the subset of elements of \(G\) that fulfill \text{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 fulfilling \text{Pr} and can be given if known in order to speed up the calculation.

2 ▶ ElementProperty( \(G, \text{Pr}[\text{, } L[\text{, } R]]\) )

\text{ElementProperty} returns an element \(\pi\) of the permutation group \(G\) such that the one-argument function \(\text{Pr}\) returns \text{true} for \(\pi\). It returns \text{fail} if no such element exists in \(G\). The optional arguments \(L\) and \(R\) are subgroups of \(G\) such that the property \(\text{Pr}\) has the same value for all elements in the cosets \(Lg\) and \(gR\), respectively.

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 \(R := C_G(b)\).

```gap
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 40 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 \text{OnPoints, OnSets, OnTuples} and \text{OnSetSets}. Some of these methods use more specific refinements than \text{SubgroupProperty} or \text{ElementProperty}. For the definition of refinements, and how one can define refinements, see Section 8.2 in “Extending \text{GAP}”. 
Chapter 41. Permutation Groups

31. TwoClosure( G )

The 2-closure of a transitive permutation group \( G \) on \( n \) points is the largest subgroup of \( S_n \) 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 \( G \).

\[
gap> \text{TwoClosure(\text{Group}((1,2,3),(2,3,4)))}; \\
\text{Sym( [ 1 .. 4 ] )}
\]

41.12 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 \[\text{Sim90}\]) 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.) It is therefore beneficial to tell a group of larger degree, which is known to be solvable, that it is, using \text{SetIsSolvableGroup}(G,\text{true}).

The second aspect concerns memory usage. A permutation on more than 65536 points requires 4 byte 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 35.9).

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.

\[
gap> \text{Read("largeperms")}; \quad \text{# read generators in from file} \\
\text{gap> gens:=StraightLineProgGens(permutationlist);;} \\
\text{gap> g:=\text{Group}(gens);} \\
\text{gap> \# use random algorithm (faster, but result is monte carlo)} \\
\text{gap> StabChainOptions(g).random:=1;;} \\
\text{gap> Size(g); \# enforce computation of a stabilizer chain} \\
3529698298145066075557232833758234188056080273649172207877011796336000
\]

Without straight line program elements, the same calculation runs into memory problems after a while even with 512MB of workspace:

\[
gap> h:=\text{Group}(\text{permutationlist});; \\
\text{gap> StabChainOptions(h).random:=1;;} \\
\text{gap> Size(h);} \\
\text{exceeded the permitted memory (‘-o’ command line option) at mlimit := 1;} \\
\text{called from SCRMakeStabStrong(S.stabilizer, [ g ], param, orbits, where, basesize, base, correct, missing, false);} \text{called from SCRMakeStabStrong(S.stabilizer, [ g ], param, orbits, where, basesize, ...}
\]

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
Section 12. Working with large degree permutation groups

line program elements (see StraightLineProgGens). 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:

```gap
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, > 3529698298145066075557232833758234188056080273649172207877011796336000);
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.
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.

```gap
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) ] ]])
```

For most operations, GAP only provides methods for finite matrix groups. Many calculations in finite matrix groups are done via a `NiceMonomorphism` (see 38.5) that represents a faithful action on vectors.

### 42.1 Attributes and Properties for Matrix Groups

1. **DimensionOfMatrixGroup** *(mat-grp)*
   - The dimension of the matrix group.

2. **DefaultFieldOfMatrixGroup** *(mat-grp)*
   - Is a field containing all the matrix entries. It is not guaranteed to be the smallest field with this property.

3. **FieldOfMatrixGroup** *(matgrp)*
   - 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` to get (for example) the characteristic.

   ```gap
gap> DimensionOfMatrixGroup(m);
3

gap> DefaultFieldOfMatrixGroup(m);
GF(3)
```

4. **TransposedMatrixGroup** *(matgrp)*
   - returns the transpose of the matrix group `matgrp`. The transpose of the transpose of `matgrp` is identical to `matgrp`. 
Section 3. GL and SL

42.2 Actions of Matrix Groups

The basic operations for groups are described in Chapter 39, special actions for matrix groups mentioned there are OnLines, OnRight, and OnSubspacesByCanonicalBasis.

For subtleties concerning multiplication from the left or from the right, see 42.6.

1. ProjectiveActionOnFullSpace( G, F, n )

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.

2. ProjectiveActionHomomorphismMatrixGroup( G )

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.)

3. BlowUpIsomorphism( matgrp, B )

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 BlowupMat(A, B), see 24.12.3.

42.3 GL and SL

1. IsGeneralLinearGroup( grp ) P

P = IsGL( grp )

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.

2. IsNaturalGL( matgrp ) P

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.
Chapter 42. Matrix Groups

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.

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.

gap> IsNaturalGL(m);
false

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.

(See also section 48.2.)

42.4 Invariant Forms

This attribute contains a bilinear form that is invariant under matgrp. The form is given by a record with the component matrix which is a matrix $m$ such that for every generator $g$ of $m$ the equation $g \cdot m \cdot g^\text{tr}$ holds.

This property tests, whether a matrix group matgrp is the full subgroup of GL or SL (the property IsSubgroupSL determines which it is) respecting the InvariantBilinearForm of matgrp.

This attribute contains a sesquilinear form that is invariant under matgrp. The form is given by a record with the component matrix which is is a matrix $m$ such that for every generator $g$ of $m$ the equation $g \cdot m \cdot (g^\text{tr})^F$ holds, where $F$ is the FrobeniusAutomorphism of the FieldOfMatrixGroup of $G$.

This property tests, whether a matrix group matgrp is the full subgroup of GL or SL (the property IsSubgroupSL determines which it is) respecting the InvariantSesquilinearForm of matgrp.

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 $q(v) = vQv^\text{tr}$, and the invariance under matgrp is given by the equation $q(v) = q(vM)$ for all $v \in V$ and $M$ in matgrp.

This property tests, whether the matrix group matgrp is the full subgroup of GL or SL (the property IsSubgroupSL determines which it is) respecting the InvariantQuadraticForm value of matgrp.
42.5 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.

1► IsCyclotomicMatrixGroup( G )
   tests whether all matrices in \( G \) have cyclotomic entries.

2► IsRationalMatrixGroup( G )
   tests whether all matrices in \( G \) have rational entries.

3► IsIntegerMatrixGroup( G )
   tests whether all matrices in \( G \) have integer entries.

4► IsNaturalGLnZ( G )
   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.)
   
   gap> IsNaturalGLnZ( GL( 2, Integers ) );
   true

5► IsNaturalSLnZ( G )
   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.)

\[
gap> g := \text{Sp}( 2, 3 );;
gap> m := \text{InvariantBilinearForm}( g ).\text{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> \text{IsFullSubgroupGLorSLRespectingBilinearForm}( g );
true
gap> g := \text{SU}( 2, 4 );;
gap> m := \text{InvariantSesquilinearForm}( g ).\text{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> \text{IsFullSubgroupGLorSLRespectingSesquilinearForm}( g );
true
gap> g := \text{GO}( 1, 2, 3 );;
gap> m := \text{InvariantBilinearForm}( g ).\text{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 := \text{InvariantQuadraticForm}( g ).\text{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> \text{IsFullSubgroupGLorSLRespectingQuadraticForm}( g );
true
gap> IsNaturalSLnZ( SL( 2, Integers ) );
true

6► InvariantLattice( G )

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 \texttt{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.

7► NormalizerInGLnZ( G )

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 \texttt{Normalizer( GL( n, Integers ), G )}.

8► CentralizerInGLnZ( G )

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 \texttt{Centralizer( GL( n, Integers ), G )}.

9► ZClassRepsQClass( G )

The conjugacy class in \( GL_n(\mathbb{Q}) \) of the finite integer matrix group \( G \) splits into finitely many conjugacy classes in \( GL_n(\mathbb{Z}) \). \texttt{ZClassRepsQClass( G )} returns representative groups for these.

10► IsBravaisGroup( G )

test whether \( G \) coincides with its Bravais group (see 42.5.11).

11► BravaisGroup( G )

returns the Bravais group of a finite integer matrix group \( G \). If \( C \) is the cone of positive definite quadratic forms \( Q \) invariant under \( g \mapsto g * Q * g^\text{tr} \) 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 \( g \mapsto g^\text{tr} * Q * g \) 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.

12► BravaisSubgroups( G )

returns the subgroups of the Bravais group of \( G \), which are themselves Bravais groups.

13► BravaisSupergroups( G )

returns the subgroups of \( GL_n(\mathbb{Z}) \) that contain the Bravais group of \( G \) and are Bravais groups themselves.

14► NormalizerInGLnZBravaisGroup( G )

returns the normalizer of the Bravais group of \( G \) in the appropriate \( GL_n(\mathbb{Z}) \).
Section 6. Acting OnRight and OnLeft

42.6 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.

1 CrystGroupDefaultAction

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.

2 SetCrystGroupDefaultAction(action)

allows to set the value of the global variable CrystGroupDefaultAction. Only the arguments RightAction and LeftAction are allowed. Initially, the value of CrystGroupDefaultAction is RightAction.
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 a **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 and hence **GAP** tries to use them whenever possible.

In **GAP 3** these methods have been available for **AgGroups** only; that is, for groups defined via a power-commutator presentation, see Chapter 44 for the **GAP 4** analogon. This has changed in **GAP 4** where these methods can be applied to many types of groups. For example, the methods can be applied to permutation groups or matrix groups which 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 methods for polycyclic groups can only be applied to finite groups. However, a more general implementation is planned.

### 43.1 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 $g_iC_{i+1}$ 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 $g$. 
43.2 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 45.13.

Note also that a pcgs behaves like a list.

1 $\triangleright$ Pcgs( $G$ )
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 $G$!

2 $\triangleright$ IsPcgs( $obj$ )
The category of pcgs.

3 $\triangleright$ CanEasilyComputePcgs( $grp$ )
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.

43.3 Defining a Pcgs Yourself

In a number of situations it might be useful to supply a pcgs to a group.

1 $\triangleright$ PcgsByPcSequence( $fam$, $pcs$ )
2 $\triangleright$ PcgsByPcSequenceNC( $fam$, $pcs$ )
constructs a pcgs for the elements family $fam$ from the elements in the list $pcs$. The elements must lie in the family $fam$. PcgsByPcSequenceNC will always create a new pcgs which is not induced by any other pcgs.
Chapter 43. Polycyclic Groups

 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 ]

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 SetRelativeOrders( pcgs, orders ). See also 43.4.3.

### 43.4 Elementary Operations for a Pcgs

1. **RelativeOrders( pcgs )**
   - Returns the list of relative orders of the pcgs `pcgs`

2. **IsFiniteOrdersPcgs( pcgs )**
   - Tests whether the relative orders of `pcgs` are all finite.

3. **IsPrimeOrdersPcgs( pcgs )**
   - Tests whether the relative orders of `pcgs` are prime numbers. Many algorithms require a pcgs to have this property. The operation IsomorphismRefinedPcGroup (see 44.4.8) can be of help here.

4. **PcSeries( pcgs )**
   - Returns the subnormal series determined by `pcgs`.

5. **GroupOfPcgs( pcgs )**
   - The group generated by `pcgs`.

6. **OneOfPcgs( pcgs )**
   - The identity of the group generated by `pcgs`.

```gap
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()) ]```
Section 5. Elementary Operations for a Pcgs and an Element

43.5 Elementary Operations for a Pcgs and an Element

1. \texttt{RelativeOrderOfPcElement( pcgs, elm )}
   
   The relative order of \textit{elm} with respect to the prime order \textit{pcgs} \textit{pcgs}.

2. \texttt{ExponentOfPcElement( pcgs, elm, pos )}
   
   returns the \textit{pos}-th exponent of \textit{elm} with respect to \textit{pcgs}.

3. \texttt{ExponentsOfPcElement( pcgs, elm )}
   
   \texttt{ExponentsOfPcElement( pcgs, elm, posran )}
   
   returns the exponents of \textit{elm} with respect to \textit{pcgs}. The second form returns the exponents in the positions given in \textit{posran}.

4. \texttt{DepthOfPcElement( pcgs, elm )}
   
   returns the depth of the element \textit{elm} with respect to \textit{pcgs}.

5. \texttt{LeadingExponentOfPcElement( pcgs, elm )}
   
   returns the leading exponent of \textit{elm} with respect to \textit{pcgs}.

6. \texttt{PcElementByExponents( pcgs, list )}
   
   \texttt{PcElementByExponentsNC( pcgs, list )}
   
   \texttt{PcElementByExponentsNC( pcgs, basisind, list )}
   
   returns the element corresponding to the exponent vector \textit{list} with respect to \textit{pcgs}. The exponents in \textit{list} must be in the range of permissible exponents for \textit{pcgs}. \textit{It is not guaranteed that PcElementByExponents will reduce the exponents modulo the relative orders}. (You should use the operation \texttt{LinearCombinationPcgs} 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 third version gives exponents only w.r.t. the generators in \textit{pcgs} indexed by \textit{basisind}.

7. \texttt{LinearCombinationPcgs( pcgs, list [ , one ] )}
   
   returns the product $\prod pcgs[i]^{list[i]}$. In contrast to \texttt{PcElementByExponents} this permits negative exponents. \textit{pcgs} might be an list of group elements, in this case, an appropriate \textit{one} must be given. if \textit{list} can be empty.
   
   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 ]

8. \texttt{SiftedPcElement( pcgs, elm )}
   
   sifts \textit{elm} through \textit{pcgs}, reducing it if the depth is the same as the depth of one of the generators in \textit{pcgs}. Thus the identity is returned if \textit{elm} lies in the group generated by \textit{pcgs}. \textit{pcgs} must be an induced \textit{pcgs} and \textit{elm} must lie in the span of the parent of \textit{pcgs}.

9. \texttt{CanonicalPcElement(ipcgs, elm )}
   
   reduces \textit{elm} at the induces \textit{pcgs} \textit{ipcgs} such that the exponents of the reduced result \textit{r} are zero at the depths for which there are generators in \textit{ipcgs}. Elements, whose quotient lies in the group generated by \textit{ipcgs} yield the same canonical element.

10. \texttt{ReducedPcElement( pcgs, x, y )}
    
    reduces the element \textit{x} by dividing off (from the left) a power of \textit{y} such that the leading coefficient of the result with respect to \textit{pcgs} becomes zero. The elements \textit{x} and \textit{y} therefore have to have the same depth.
returns an element in the span of \( pgs \) whose exponents for indices 1 to \( \text{dep} - 1 \) with respect to \( pgs \) 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 43.9.

The difference to HeadPcElementByNumber (see 43.5.12) is that HeadPcElementByNumber is guaranteed to zero out trailing coefficients while CleantedTailPcElement will only do this if it can be done cheaply.

returns an element in the span of \( pgs \) whose exponents for indices 1 to \( \text{dep} - 1 \) with respect to \( pgs \) 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.

### 43.6 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.)

#### ExponentsConjugateLayer

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 pcs of \( mpcgs \) and \( mpcgs \) must be the modulo pcs 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 pcs, the exponents can be computed directly by looking up exponents without having to compute in the group and having to collect a potential tail.

The second class are exponents of products of the generators which make up the pcs. If the pcs used is a FamilyPcgs these exponents can be looked up and do not need to be computed.

#### ExponentsOfRelativePower

For \( p = pgs[i] \) this function returns the exponent vector with respect to \( pgs \) of the element \( p^e \) where \( e \) is the relative order of \( p \) in \( pgs \). For the family pcs or pcs induced by it, this might be faster than computing the element and computing its exponent vector.

#### ExponentsOfConjugate

returns the exponents of \( pgs[i]^{\text{mpg}[j]} \) with respect to \( pgs \). For the family pcs or pcs induced by it, this might be faster than computing the element and computing its exponent vector.

#### ExponentsOfCommutator

returns the exponents of the commutator \( \text{Comm}(pgs[i], pgs[j]) \) with respect to \( pgs \). For the family pcs or pcs induced by it, this might be faster than computing the element and computing its exponent vector.
43.7 Subgroups of Polycyclic Groups - Induced Pcsgs

Let $U$ be a subgroup of $G$ and let $P$ be a pcsgs 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 $u_i \in (U \cap C_i) \setminus (U \cap C_{i+1})$ whenever this factor is non-trivial, then we obtain a pcsgs $Q = (u_1, \ldots, u_m)$ of $U$. We say that $Q$ is an induced pcsgs with respect to $P$. The pcsgs $P$ is the parent pcsgs to the induced pcsgs $Q$.

Note that the pcsgs $Q$ is induced with respect to $P$ if and only if the matrix of exponent vectors of the elements $u_i$ with respect to $P$ is in upper triangular form. Thus $Q$ is not unique in general.

In particular, the elements of an induced pcsgs do not necessarily have leading coefficient 1 relative to the inducing pcsgs. The attribute LeadCoeffsIGS (see 43.7.7) holds the leading coefficients in case they have to be renormed in an algorithm.

Each induced pcsgs is a pcsgs and hence allows all elementary operations for pcsgs. On the other hand each pcsgs could be transformed into an induced pcsgs for the group defined by the pcsgs, but note that an arbitrary pcsgs is in general not an induced pcsgs for technical reasons.

An induced pcsgs is “compatible” with its parent.

1. IsInducedPcsgs( pcs )
   The category of induced pcsgs. This a subcategory of pcsgs.

2. InducedPcsgsByPcSequence( pcs, pcs )
   InducedPcsgsByPcSequenceNC( pcs, pcs )
   InducedPcsgsByPcSequenceNC( pcs, pcs, depths )
   If $pcs$ is a list of elements that form an induced pcsgs with respect to $pcs$ this operation returns an induced pcsgs with these elements.
   In the third version, the depths of $pcs$ with respect to $pcs$ can be given (they are computed anew otherwise).

3. ParentPcsgs( pcs )
   returns the pcsgs by which $pcs$ was induced. If $pcs$ was not induced, it simply returns $pcs$.

   gap> G := Group( (1,2,3,4),(1,2) );;
   gap> P := Pcgs(G);;
   gap> K := InducedPcsgsByPcSequence( P, [ (1,2,3,4),(1,3)(2,4) ] );
   Pcsgs( [ (1,2,3,4), (1,3)(2,4) ] )
   gap> ParentPcsgs( K );
   Pcsgs( [ (1,2,3,4), (1,3)(2,4), (1,4)(2,3), (1,3)(2,4) ] )
   gap> IsInducedPcsgs( K );
   true

In [LNS84] a “non-commutative gauss” algorithm is described to compute an induced pcsgs of a subgroup $U$ from a generating set of $U$. This can be called in GAP via one of the following commands.

4. InducedPcsgs( pcs, grp )
   computes a pcsgs for $grp$ which is induced by $pcs$. If $pcs$ has a parent pcsgs, then the result is induced with respect to this parent pcsgs.

5. InducedPcsgsByGenerators( pcs, gens )
   InducedPcsgsByGeneratorsNC( pcs, gens )
   returns an induced pcsgs with respect to $pcs$ for the subgroup generated by $gens$. 

In [LNS84] a “non-commutative gauss” algorithm is described to compute an induced pcsgs of a subgroup $U$ from a generating set of $U$. This can be called in GAP via one of the following commands.
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.

\[
gap> G := \text{Group}( (1,2,3,4),(1,2) );; \quad P := \text{Pcgs}(G);;
\]
\[
gap> I := \text{InducedPcgsByGenerators( } P, [[1,2,3,4]] );
\]
\[
gap> J := \text{InducedPcgsByPcSequenceAndGenerators( } P, I, [[(1,2)] ];
\]

\[
\text{Pcgs([ (1,2,3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])}
\]

\[
\text{LeadCoeffsIGS( } igs \text{ )}
\]
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 \text{InducedPcgsByPcSequenceNC} as the permutation group methods call it from within the postprocessing, before this postprocessing however no coefficients may be computed.)

\[
\text{ExtendedPcgs( } N, gens \text{ )}
\]
extends the pcgs \( N \) (induced wrt. \text{home}) to a new induced pcgs by prepending \( gens \). No checks are performed that this really yields an induced pcgs.

To create a subgroup generated by an induced pcgs such that the induced pcgs gets stored automatically there is the following operation.

\[
\text{SubgroupByPcgs( } G, pcgs \text{ )}
\]

\subsection{43.8 Subgroups of Polycyclic Groups - Canonical Pcgs}

The induced pcgs \( Q \) of \( U \) is called \text{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.

\[
\text{IsCanonicalPcgs( } pcgs \text{ )}
\]
An induced pcgs is canonical if the matrix of the exponent vectors of the elements of \( pcgs \) with respect to \text{ParentPcgs}(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.

\[
\text{CanonicalPcgs( } pcgs \text{ )}
\]
returns the canonical pcgs corresponding to the induced pcgs \( pcgs \).

\[
gap> G := \text{Group}( (1,2,3,4),(5,6,7) );
\]
\[
\text{Group([[ (1,2,3,4), (5,6,7) ]])}
\]
\[
\text{Pcgs([[ (5,6,7), (1,2,3,4), (1,3)(2,4) ]])}
\]
\[
\text{InducedPcgsByPcSequence(P, [[(5,6,7)*((1,3)(2,4)),(1,3)(2,4)]] );}
\]
\[
\text{Pcgs([[ (1,3)(2,4)(5,6,7), (1,3)(2,4) ]])}
\]
\[
\text{CanonicalPcgs(I);}
\]
\[
\text{Pcgs([[ (5,6,7), (1,3)(2,4) ]])}
\]
43.9 Factor Groups of Polycyclic Groups - Modulo Pcs

Let \(N\) be a normal subgroup of \(G\) such that \(G/N\) is polycyclic with pcs \((h_1N, \ldots, h_rN)\). Then we call the sequence of preimages \((h_1, \ldots, h_r)\) a modulo pcs of \(G/N\). \(G\) is called the numerator of the modulo pcs and \(N\) is the denominator of the modulo pcs.

Modulo pcs 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 pcs, see Sections 43.4 and 43.5, apply to modulo pcs as well. However, it is in general not possible to compute induced pcs with respect to a modulo pcs.

\[
\text{ModuloPcgs}(G, N) \quad \text{returns a modulo pcs for the factor } G/N \text{ which must be solvable, which } N \text{ may be insolvable.}
\]

\[
\text{IsModuloPcgs}(\text{obj}) \quad \text{The category of modulo pcs. Note that each pcs is a modulo pcs for the trivial subgroup. Additionally there are two more elementary operations for modulo pcs.}
\]

\[
\text{NumeratorOfModuloPcgs}(\text{pcs}) \quad \text{returns a generating set for the numerator of the modulo pcs pcs.}
\]

\[
\text{DenominatorOfModuloPcgs}(\text{pcs}) \quad \text{returns a generating set for the denominator of the modulo pcs pcs.}
\]

\[
\text{gap> G := Group( (1,2,3,4,5),(1,2) );}
\]
\[
\text{Group( [ (1,2,3,4,5), (1,2) ] )}
\]
\[
\text{gap> P := ModuloPcgs(G, DerivedSubgroup(G)); Pcgs([ (4,5) ]);}
\]
\[
\text{gap> NumeratorOfModuloPcgs(P); [ (1,2,3,4,5), (1,2) ]}
\]
\[
\text{gap> DenominatorOfModuloPcgs(P); [ (1,3,2), (2,4,3), (2,3)(4,5) ]}
\]
\[
\text{gap> RelativeOrders(P); [ 2 ]}
\]
\[
\text{gap> ExponentsOfPcElement( P, (1,2,3,4,5) ); [ 0 ]}
\]
\[
\text{gap> ExponentsOfPcElement( P, (4,5) ); [ 1 ]}
\]

Modulo Pcs can also be built from compatible induced pcs. Let \(G\) be a group with pcs \(P\) and let \(I\) be an induced pcs of a normal subgroup \(N\) of \(G\). (Respectively: \(P\) and \(I\) are both induced with respect to the same Pcs.) Then we can compute a modulo pcs of \(G\) mod \(N\) by

\[
\text{P mod I}
\]

Note that in this case we obtain the advantage that the NumeratorOfModuloPcgs and the DenominatorOfModuloPcgs are just \(P\) and \(I\), respectively, and hence are unique.

The resulting modulo pcs will consist of a subset of \(P\) and will be “compatible” with \(P\) (or its parent).
Chapter 43. Polycyclic Groups

```gap
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) ])
```

6 ▶ CorrespondingGeneratorsByModuloPcgs( mpcgs, imgs )

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.

7 ▶ CanonicalPcgsByGeneratorsWithImages( pcgs, gens, imgs )

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.

43.10 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 on a modulo pcgs, see 44.5.1).

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.

1 ▶ ProjectedPcElement( pcgs, fpcgs, elm )

returns the image in H of an element elm of G.

2 ▶ ProjectedInducedPcgs( pcgs, fpcgs, ipcs )

ipcs 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 ipcs.

3 ▶ LiftedPcElement( pcgs, fpcgs, elm )

returns a preimage in G of an element elm of H.

4 ▶ LiftedInducedPcgs( pcgs, fpcgs, ipcs, ker )

ipcs must be an induced pcgs with respect to fpcgs. This operation returns an induced pcgs with respect to pcgs consisting of the preimages of ipcs, appended by the elements in ker (assuming there is a bijection of pcgs mod ker to fpcgs). ker might be a simple element list.
43.11 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 43.13 for a more explicit possibility.

1▶ IsPcgsElementaryAbelianSeries( pcgs )
returns true if the pcgs pcgs refines an elementary abelian series. IndicesNormalSteps gives the indices in the Pcgs, at which the normal subgroups of this series start.

2▶ PcgsElementaryAbelianSeries( G )
computes a pcgs for G that refines an elementary abelian series. IndicesNormalSteps 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 N1, N2, etc.

3▶ IsPcgsCentralSeries( pcgs )
returns true if the pcgs pcgs refines a central series. IndicesNormalSteps gives the indices in the Pcgs, at which the normal subgroups of this series start.

4▶ PcgsCentralSeries( G )
computes a pcgs for G that refines a central series. IndicesNormalSteps gives the indices in the Pcgs, at which normal subgroups start. If G is not nilpotent, the result is undefined.

5▶ IsPcgsPCentralSeriesPGroup( pcgs )
6▶ PcgsPCentralSeriesPGroup( G )
computes a pcgs for G that refines the lower p-central series. IndicesNormalSteps gives the indices in the Pcgs, at which normal subgroups start. If G is not a p-group, the result is undefined.

7▶ PcgsChiefSeries( G )
computes a pcgs for G that refines a chief series. IndicesNormalSteps gives the indices in the Pcgs, at which normal subgroups start.

Let P = (g1,...,gn) be a pcgs such that the pc series determined by P refines a normal series of one of the above types. To obtain the subgroups of this series, the following attributes can be used:

8▶ NormalSeriesByPcgs( pcgs )
returns the series of all normal subgroups contained in the pc series determined by pcgs.

9▶ IndicesNormalSteps( pcgs )
Let pcgs be a pcgs obtained as corresponding to a series of normal subgroups with elementary abelian factors (for example PcgsElementaryAbelianSeries, PcgsChiefSeries or PcgsCentralSeries). Then IndicesNormalSteps 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, (g1,...,gi) 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.)

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and IndicesNormalSteps is not guaranteed to work in this situation.
Typically, \texttt{IndicesNormalSteps} is set by calls to operations that compute a particular pcgs such as \texttt{PcgsElementaryAbelianSeries}. (Note, however, that in an existing pcgs this attribute might have been set already differently by a previous calculation. In this case the pcgs is never \texttt{IsPcgsElementaryAbelianSeries}, even though it could be.)

```gap
g:=Group((1,2,3,4),(1,2));;
p:=PcgsElementaryAbelianSeries(g);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IndicesNormalSteps(p);
[ 1, 2, 3, 5 ]
g:=Group((1,2,3,4),(1,5)(2,6)(3,7)(4,8));;
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> IndicesNormalSteps(p);
[ 1, 2, 4, 5, 6 ]
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> IndicesNormalSteps(q);
[ 1, 3, 5, 6 ]
```

### 43.12 Sum and Intersection of Pcgs

\texttt{SumFactorizationFunctionPcgs( parentpcgs, n, u, kerpcgs )}

computes the sum and intersection of the lists \( n \) and \( u \) whose elements form modulo pcgs induced by \( \text{parentpcgs} \) for two subgroups modulo a kernel given by \( \text{kerpcgs} \). If \( \text{kerpcgs} \) is a tail if the \( \text{parent-pcgs} \) it is sufficient to give the starting depth, this can be more efficient than to construct an explicit pcgs. The factor group modulo \( \text{kerpcgs} \) 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 \texttt{sum} a record with components \( u \) and \( n \) that give its decomposition.

The record components \texttt{sum} and \texttt{intersection} are not pcgs but only lists of pc elements (to avoid unnecessary creation of InducedPcgs).

### 43.13 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 \textbf{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 $F_i = P_{i,1} \cdots P_{i,r}$. 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 $\pi$ the elements of the pcgs with relative order in $\pi$ form a pcgs of a Hall $\pi$-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 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.

```
1  IsSpecialPcgs(  obj  )          P
   tests whether obj is a special pcgs.

2  SpecialPcgs(  pcgs  )          A
   computes a special pcgs for the group defined by pcgs or for G.

3  LGWeights(  pcgs  )           A
   returns the LGWeights of the special pcgs pcgs.

4  LGLayers(  pcgs  )            A
   returns the layers of the special pcgs pcgs.

5  LGFirst(  pcgs  )             A
   returns the first indices for each layer of the special pcgs pcgs.

6  LGLength(  G  )               A
   returns the Length of the LG-series of the group G, if G is solvable and fail otherwise.
```
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 \( f_1, f_2, f_3, f_4, f_5, f_6 \) and the Sylow 3-subgroup is generated by \( f_5 \). The lower 2-central series of the Sylow 2-subgroup has length 2 and the second subgroup in this series is generated by \( f_6 \).

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 \( f_6 \). The subgroup generated by \( f_1, \ldots, f_5 \) 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 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 44.5.3.

43.14 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 following functions.

43.14.1 VectorSpaceByPcgsOfElementaryAbelianGroup ( \( m \text{pcgs}, \text{fld} \) ) 
returns the vector space over \( \text{fld} \) corresponding to the modulo pcgs \( m \text{pcgs} \). Note that \( m \text{pcgs} \) has to define an elementary abelian \( p \)-group where \( p \) is the characteristic of \( \text{fld} \).
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.

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.

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 following functions.

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.

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.
43.15 Orbit Stabilizer Methods for Polycyclic Groups

If a 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.

1 ▶ StabilizerPcgs( pcgs, pt )

2 ▶ PcgsOrbitStabilizer( pcgs, domain, pnt, oprs, opr )

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.

43.16 Operations which have Special Methods for Groups with Pcgs

For the following methods there are special operations for groups with pcgs installed: IsNilpotent, IsSupersolvable, Size, CompositionSeries, ConjugacyClasses, Centralizer, FrattiniSubgroup, PrefrattiniSubgroup, MaximalSubgroups and related operations, HallSystem and related operations, MinimalGeneratingSet, Centre, Intersection, AutomorphismGroup, IrreducibleModules.

43.17 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.

@The syntax of this function may change in a future rewrite!@

1 ▶ ClassesSolvableGroup( G, mode [, opt] )

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 algorithms behaviour:

pcgs
is a pcgs that will be used for the calculation. In the case of the calculation of rational classes, it must be a pcgs refining a central series. The attribute NormalSeriesByPcgs must return an appropriate series of normal subgroups with elementary abelian factors among them. The algorithm will step down this series. By default an ElementaryAbelianSeries is used.

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.

2► CentralizerSizeLimitConsiderFunction(sz) returns a function (of the form func(fhome,rep,cenp,K,L) )that can be used in ClassesSolvableGroup 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 (37.20.3).
PcGroups are polycyclic groups that use the polycyclic presentation for element arithmetic. This presentation gives them a “natural” pegs, the FamilyPcgs (see 44.1.1) with respect to which pcgs operations as described in chapter 43 are particularly efficient.

Let $G$ be a polycyclic group with pegs $P = (g_1, \ldots, g_n)$ 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.

$$g_i^{r_i} = g_{i+1}^{a(i,i+1)} \cdots g_n^{a(i,n)}$$

for $1 \leq i \leq n$ and $i \in I$

$$g_i^{-1} g_j g_i = g_{i+1}^{a(i,j,i+1)} \cdots g_n^{a(i,j,n)}$$

for $1 \leq i < j \leq n$

For infinite groups we need additionally

$$g_i^{-1} g_j^{-1} g_i = g_{i+1}^{b(i,j,i+1)} \cdots g_n^{b(i,n)}$$

for $1 \leq i < j \leq n$ and $i \notin I$

$$g_i g_j g_i^{-1} = g_{i+1}^{c(i,j,i+1)} \cdots g_n^{c(i,n)}$$

for $1 \leq i < j \leq n$ and $j \notin I$

$$g_i g_j^{-1} g_i^{-1} = g_{i+1}^{d(i,j,i+1)} \cdots g_n^{d(i,n)}$$

for $1 \leq i < j \leq n$ and $i, j \notin I$

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 pegs $P$ 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 level of implementation GAP can only deal with finite pc groups. This will be extended in near future.

Algorithms for pc groups use the methods for polycyclic groups described in chapter 43.
44.1 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.

\[
\begin{align*}
1 & \quad \text{FamilyPcgs( } \text{grp} \text{ )} \\
2 & \quad \text{IsFamilyPcgs( } \text{pcgs} \text{ )} \\
3 & \quad \text{InducedPcgsWrtFamilyPcgs( } \text{grp} \text{ )} \\
4 & \quad \text{IsParentPcgsFamilyPcgs( } \text{pcgs} \text{ )}
\end{align*}
\]

This property indicates that the pcgs \( \text{pcgs} \) is induced with respect to a family pcgs.

In \textsc{GAP} 3 the family pcgs had been the only pcgs allowed for a pc group. Note that this has changed in \textsc{GAP} 4 where a pc group may have several independent polycyclic generating sequences.

However, 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 43.11.2) or even a special pcgs (see 43.13). If the family pcgs has the required properties, it will be used for these purposes, if not \textsc{GAP} has to work with respect to a new pcgs which is\textbf{not} the family pcgs and thus takes longer for elementary calculations like \texttt{ExponentsOfPcElement}.

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 \texttt{Range} of the isomorphism obtained by \texttt{IsomorphismSpecialPcGroup} (see 44.5.3).

44.2 Elements of pc groups

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 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.

\[
\begin{align*}
1 & \quad \text{pcword} = \text{pcword} \\
\text{pcword} < \text{pcword}
\end{align*}
\]

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 to elements in a pc group, we first concatenate the corresponding words and then use an algorithm called \texttt{collection} to transform the new word into a word in normal form.

\[
\begin{align*}
gap> g := \text{FamilyPcgs( SmallGroup(24, 12) )}; \\
Pcgs([f1, f2, f3, f4]) \\
gap> g[4] * g[1]; \\
f1*f3 \\
gap> (g[2] * g[3])^-1; \\
f2^-2*f3*f4
\end{align*}
\]
44.3 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.

1 ▶ IsPcGroup( G )

tests whether G is a pc group.

gap> G := SmallGroup( 24, 12 );
<pc group of size 24 with 4 generators>
gap> IsPcGroup( G );
true

gap> IsFpGroup( G );
falso

Note that it is possible to convert a pc group to a fp group in GAP. The following function 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.

2 ▶ IsomorphismFpGroupByPcgs( pcgs, str )


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, g4^2, g4^-1*g3^-1*g4*g3, g4^2 ]

44.4 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, see below 44.5.1.) One way is to define a finitely presented group with a pc presentation in GAP and then convert this presentation into a pc group. Note that this does not work for arbitrary presentations of polycyclic groups, see Chapter 45.13 for further information.

For performance reasons it is beneficial to enforce a “syllable” representation in the free group (see 35.6).

1 ▶ PcGroupFpGroup( G )

creates a PcGroup P from an FpGroup (see Chapter 45) 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.
Section 4. Constructing Pc Groups

439

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)];
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>

Equivalently to the above method one can initiate a collector of a pc group by hand and use it to define a
pc group. In GAP there are different collectors for different collecting strategies; at the moment, there are
two collectors to choose from: the single collector for finite pc groups and the combinatorial collector for
finite p-groups. See [Sim94] for further information on collecting strategies.

A collector is initiated by underlying free group to the pc presented 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.

2► SingleCollector( fgrp, reorders )
3► CombinatorialCollector( fgrp, reorders )

Then the right hand sides of the pc presentation have to be declared. Let \( f_1, \ldots, f_n \) be the generators of the
underlying free group \( fgrp \).

4► SetConjugate( coll, i, j, w )

set the conjugate \( f_i^j \) to equal \( w \) where \( w \) is a word in \( f_{i+1}, \ldots, f_n \) and \( i < j \).

4► SetCommutator( coll, i, j, w )

set the commutator of \( f_j \) and \( f_i \) to equal \( w \) where \( w \) is a word in \( f_{i+1}, \ldots, f_n \) and \( i < j \).

5► SetPower( coll, i, w )

set the power \( f_i^w \) to equal \( w \) where \( w \) is a word in \( f_{i+1}, \ldots, f_n \).

Finally, the collector has to be converted to a group.

6► GroupByRws( coll )
7► GroupByRwsNC( coll )

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.

7► IsConfluent( G )

checks whether the pc group \( G \) has been build from a collector with a confluent power-commutator presenta-
tion.

\[
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
Chapter 44. Pc Groups

With the above 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. Hence the following functions are useful.

8 ▶ IsomorphismRefinedPcGroup( G )
returns an isomorphism from G onto an isomorphic PC group whose family pcgs is a prime order pcgs.

9 ▶ RefinedPcGroup( G )
returns the range of IsomorphismRefinedPcGroup(G).

44.5 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.

1 ▶ PcGroupWithPcgs( mpcgs )
creates a new Pc group G whose family pcgs is isomorphic to the (modulo) pcgs mpcgs.

2 ▶ IsomorphismPcGroup( G )
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.
Section 8. 2-Cohomology and Extensions

```
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 ])
```

3. IsomorphismSpecialPcGroup( G )

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.

44.6 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.

```
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
```

44.7 Operations for Pc Groups

All the operations described in Chapters 37 and 43 apply to a pc group. Nearly all methods for pc groups are methods for groups with pcgs as described in Chapter 43. 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 be guaranteed within pc groups.

44.8 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. For further information on meataxe modules see Chapter 67. Note that the matrices defining the module must correspond to the pcgs of the group $G$.

```
gap> TwoCoboundaries( G, M )
```

returns the group of 2-coboundaries of a pc group $G$ by the $G$-module $M$. The generators of $M$ must correspond to Pcgs($G$). The group of coboundaries is given as vector space over the field underlying $M$. 

TwoCocycles\((G, M)\)
returns the group of 2-cocycles of a pc group \(G\) by the \(G\)-module \(M\). The generators of \(M\) must correspond to \(\text{Pcgs}(G)\). The group of cocycles is given as vector space over the field underlying \(M\).

TwoCohomology\((G, M)\)
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 \(G\).

Extensions\((G, M)\)
returns all extensions of \(G\) by the \(G\)-module \(M\) up to equivalence as pc groups.

Extension\((G, M, c)\)
ExtensionNC\((G, M, c)\)
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.

SplitExtension\((G, M)\)
returns the split extension of \(G\) by the \(G\)-module \(M\).

ModuleOfExtension\((E)\)
returns the module of an extension \(E\) of \(G\) by \(M\). This is the normal subgroup of \(E\) which corresponds to \(M\).
Section 8. 2-Cohomology and Extensions

<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>

\[
gap\text{List(last, IdSmallGroup);}\]
\[
\begin{array}{c}
[ [ 8, 5 ], [ 8, 2 ], [ 8, 3 ], [ 8, 3 ], [ 8, 2 ], [ 8, 3 ], [ 8, 4 ] \]
\end{array}
\]

Note that the extensions returned by \texttt{Extensions} are computed up to equivalence, but not up to isomorphism.

There exists an action of the subgroup of compatible pairs in $\text{Aut}(G) \times \text{Aut}(M)$ which acts on the second cohomology group. 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.

8 ▶ \texttt{CompatiblePairs( G, M [, D] )}

returns the group of compatible pairs of the group $G$ with the $G$-module $M$ as subgroup of the direct product of $\text{Aut}(G) \times \text{Aut}(M)$. Here $\text{Aut}(M)$ is considered as subgroup of a general linear group. The optional argument $D$ should be a subgroup of $\text{Aut}(G) \times \text{Aut}(M)$. If it is given, then only the compatible pairs in $D$ are computed.

9 ▶ \texttt{ExtensionRepresentatives( G, M, P )}

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 $M$.

\[
gap\text{G := SmallGroup( 4, 2 );;}
gap\text{mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );;}
gap\text{M := GModuleByMats( mats, GF(2) );;}
gap\text{A := AutomorphismGroup( G );;}
gap\text{B := GL( 1, 2 );;}
gap\text{D := DirectProduct( A, B );}
gap\text{P := CompatiblePairs( G, M, D );}
gap\text{<group with 6 generators>}
gap\text{P := ExtensionRepresentatives( G, M, P );}
\begin{array}{c}
[ <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}>,
 <\text{pc group of size 8 with 3 generators}> ]
gap\text{Extensions( G, M );}
\end{array}
\]

See also the forthcoming \texttt{GAP} package on Group Construction Methods.
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.

10 ▶ \texttt{SplitExtensions( } G, \texttt{ aut, } N \texttt{ )}

returns the split extensions of the pc group $G$ by the pc group $N$. \texttt{aut} should be a homomorphism from $G$ into $\text{Aut}(N)$.

In the following example we construct the holomorph of $Q_8$ as split extension of $Q_8$ by $S_4$.

\begin{verbatim}
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 ]), Pcgs([ f1, f2, f3 ]), Pcgs([ f1, f2, f3 ]), Pcgs([ f1, f2, f3 ]) ]
gap> K := SplitExtension( G, inv, N );
<pc group of size 192 with 7 generators>
\end{verbatim}

44.9 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 48.7.

1 ▶ \texttt{CodePcgs( } pcgs \texttt{ )}
returns the code corresponding to \texttt{pcgs}.

2 ▶ \texttt{CodePcGroup( } G \texttt{ )}
returns the code for a pcgs of $G$.

3 ▶ \texttt{PcGroupCode( } code, size \texttt{ )}
returns a pc group of size \texttt{size} corresponding to \texttt{code}.

4 ▶ \texttt{PcGroupCodeRec( } rec \texttt{ )}
Here \texttt{rec} needs to have entries .\texttt{code} and .\texttt{order}. Then \texttt{PcGroupCodeRec} returns a pc group of size \texttt{order} corresponding to \texttt{code}.
Section 10. Random Isomorphism Testing

44.10 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.

This method takes a list of groups and a non-negative integer as input. The output is a sublist of the input list where only isomorphic copies have been removed. The integer 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 the integer 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.

\[ \text{RandomIsomorphismTest}( \text{list}, n ) \]

Here \( \text{list} \) must be a list of pc groups and \( n \) a non-negative integer. Returns a sublist of \( \text{list} \) where isomorphic copies detected by the probabilistic test have been removed.
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.

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 35.2.1 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

\[ \langle a, b \mid a^2, b^3, (a \cdot b)^5 \rangle \]

you can use the following commands:

```gap
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.

```gap
gap> GeneratorsOfGroup( g );
[ a, b ]
gap> a;
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 (see 35.2.5).

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.
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 45.5 and 45.15.

A consequence of our convention is that elements of finitely presented groups are not printed in a unique
way.

1. IsSubgroupFpGroup( H )
   returns true if H is a finitely presented group or a subgroup of a finitely presented group.

2. IsFpGroup( G )
   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 44 for details.

3. InfoFpGroup
   The info class for functions dealing with finitely presented groups is InfoFpGroup.

45.1 Creating Finitely Presented Groups

1. F/rels
   creates a finitely presented group given by the presentation (gens | rels) where gens are the 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/(a*b).

2. FactorGroupFpGroupByRels( G, elts )
   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.
45.2 Comparison of Elements of Finitely Presented Groups

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.

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 \texttt{FpElmComparisonMethod} is used to obtain a comparison function for a family of FpGroup elements.

3\textbullet\quad \texttt{FpElmComparisonMethod}( \textit{fam} )

If \textit{fam} is the elements family of a finitely presented group this attribute returns a function \texttt{smaller(left, right)} that will be used to compare elements in \textit{fam}.

45.3 Preimages in the Free Group

\texttt{FreeGroupOfFpGroup}( \textit{G} )

returns the underlying free group for the finitely presented group \textit{G}. This is the group generated by the free generators provided by \texttt{FreeGeneratorsOfFpGroup}(\textit{G}).

\texttt{FreeGeneratorsOfFpGroup}( \textit{G} )

\texttt{FreeGeneratorsOfWholeGroup}( \textit{U} )

\texttt{FreeGeneratorsOfFpGroup} returns the underlying free generators corresponding to the generators of the finitely presented group \textit{G} which must be a full fp group.

\texttt{FreeGeneratorsOfWholeGroup} also works for subgroups of an fp group and returns the free generators of the full group that defines the family.

\texttt{RelatorsOfFpGroup}( \textit{G} )

returns the relators of the finitely presented group \textit{G} as words in the free generators provided by \texttt{FreeGeneratorsOfFpGroup}(\textit{G}).

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.
4 ▶ UnderlyingElement( elm )

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.

```gap
gap> w := g.1*g.2;
a*b
gap> IsWord( w );
false
gap> ue := UnderlyingElement( w );
a*b
gap> IsWord( ue );
true
```

5 ▶ ElementOfFpGroup( fam, word )

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.

```gap
gap> ge := ElementOfFpGroup( FamilyObj( g.1 ), f.1*f.2 );
a*b
gap> ge in f;
false
gap> ge in g;
true
```

45.4 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 45.5 for details.

The command IsomorphismPermGroup can be used to obtain a faithful permutation representation.

```gap
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) ])
```
45.5 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.

\[
\text{CosetTable}( \ G, \ H \ )
\]

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 \( l[i] = j \) 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 \text{PermList} to each generator list (see 40.4.2).

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 \text{CosetTableDefaultLimit} and \text{CosetTableDefaultMaxLimit} described below and that the options described under \text{CosetTableFromGensAndRelS} are recognized.

\[
gap> \text{tab} := \text{CosetTable}( \ g, \ \text{Subgroup}( \ g, [ g.1, g.2*g.1*g.2*g.1*g.2^-1 ] ) )
\]

\[
\left[ \left[ 1, 4, 5, 2, 3 \right], \left[ 1, 4, 5, 2, 3 \right], \left[ 2, 3, 1, 4, 5 \right], \left[ 3, 1, 2, 4, 5 \right] \right]
\]

\[
gap> \text{List}( \ \text{last}, \ \text{PermList} )
\]

\[
\left[ (2,4)(3,5), (2,4)(3,5), (1,2,3), (1,3,2) \right]
\]

\[
gap> \text{PrintArray}( \ \text{TransposedMat}( \ \text{tab} ) )
\]

\[
\left[ \left[ 1, 1, 2, 3 \right], \left[ 4, 4, 3, 1 \right], \left[ 5, 5, 1, 2 \right], \left[ 2, 2, 4, 4 \right], \left[ 3, 3, 5, 5 \right] \right]
\]

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 45.6 for a description on the way the numbers are assigned.)

\[
\text{TracedCosetFpGroup}( \ \text{tab}, \ \text{word}, \ \text{pt} ) \quad F
\]

Traces the coset number \( \text{pt} \) under the word \( \text{word} \) through the coset table \( \text{tab} \). (Note: \( \text{word} \) must be in the free group, use \text{UnderlyingElement} if in doubt.)

\[
gap> \text{TracedCosetFpGroup}( \text{tab}, \ \text{UnderlyingElement}( g.1 ), 2 )
\]

\[
4
\]

\[
\text{FactorCosetAction}( \ \text{G}, \ \text{H} )
\]

\[
\text{FactorCosetOperation}( \ \text{G}, \ \text{H} )
\]

returns the action of \( G \) on the cosets of the subgroup \( H \) of \( G \).
Section 5. Coset Tables and Coset Enumeration

```gap
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) ]
```

returns a coset table for the action of $G$ on the cosets of $H$. The columns of the table correspond to the \texttt{GeneratorsOfGroup}(G).

## 5. \texttt{CosetTableBySubgroup}(G, H)

is an internal function which is called by the functions \texttt{CosetTable}, \texttt{CosetTableInWholeGroup} and others. It is, in fact, the proper working horse that performs a Todd-Coxeter coset enumeration. \texttt{fgens} must be a set of free generators and \texttt{grels} a set of relators in these generators. \texttt{fsgens} are subgroup generators expressed as words in these generators. The function returns a coset table with respect to \texttt{fgens}.

\texttt{CosetTableBySubgroup} will call \texttt{TCENUM.CosetTableFromGensAndRels}. This makes it possible to replace the built-in coset enumerator with another one by assigning \texttt{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 \texttt{GAPTCENUM.CosetTableFromGensAndRels} even if other coset enumerators are installed.

The expected parameters are

- \texttt{fgens} generators of the free group $F$
- \texttt{grels} relators as words in $F$
- \texttt{fsgens} subgroup generators as words in $F$.

\texttt{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 \texttt{CosetTableDefaultMaxLimit}. (Due to the algorithm the actual limit used can be a bit higher than the number given.)

- \texttt{silent}
  if set to \texttt{true} the algorithm will not raise the error mentioned under option \texttt{max} but silently return \texttt{fail}. This can be useful if an enumeration is only wanted unless it becomes too big.

## 6. \texttt{CosetTableDefaultMaxLimit}

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 \texttt{GAP} has a “safety stop” built in. This is controlled by the global variable \texttt{CosetTableDefaultMaxLimit}.

If this number of cosets is reached, \texttt{GAP} will issue an error message and prompt the user to either continue the calculation or to stop it. The default value is 256000.

See also the description of the options to \texttt{CosetTableFromGensAndRels}. 

Chapter 45. Finitely Presented Groups

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 256000 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 512000 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 6.3) has been entered. The line beginning Error tells you why this occurred. The next seven lines, occur if OnBreak has its default value of Where (see 6.3.3) and explains, in this case, how GAP came to be doing a coset enumeration. Then you are give 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 (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

    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;).

7 ▶ CosetTableDefaultLimit

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), 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.

8 ▶ MostFrequentGeneratorFpGroup( G )

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.

9 ▶ IndicesInvoluntaryGenerators( G )

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.
Section 6. Standardization of coset tables

45.6 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 $w_1, w_2, ...$ are length-plus-lexicographic ordered (as defined in [Sim94]), for short: lenlex.

We would like to warn you that this standardization of coset tables is different from the concept that we have used in earlier GAP versions. That old concept ignored the columns that correspond to inverse generators and hence only considered words in the generators of $G$. We will call the old standard the semilenlex standard as it would also work in the case of semigroups where no inverses of the generators are known.

We have changed the convention from the semilenlex standard to the lenlex standard because the definition of a standard coset table in Sims' book tends to become a kind of international standard. However, for reasons of upward compatibility GAP still offers the possibility to switch back to the old convention by just changing the value of the global variable CosetTableStandard from its default value "lenlex" to "semilenlex". Then all implicit standardizations of coset tables will follow the old convention. Setting the value of CosetTableStandard back to "lenlex" again means switching back to the new convention.

1 ▶ CosetTableStandard

 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".

Independent of the current value of CosetTableStandard there is the possibility to standardize (or re-standardize) a coset table at any time using the following function.

2 ▶ StandardizeTable( table, standard )

 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 instead. Note that the function alters the given table, it does not create a copy.

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 ] ]
45.7 Coset tables for subgroups in the whole group

\[ \text{CosetTableInWholeGroup( } H \text{ )} \]

\[ \text{TryCosetTableInWholeGroup( } H \text{ )} \]

is equivalent to \( \text{CosetTable}(G, H) \) where \( G \) is the (unique) finitely presented group such that \( H \) is a subgroup of \( G \). It overrides a silent option (see 45.5.5) with false.

The variant \( \text{TryCosetTableInWholeGroup} \) 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 \( \text{CosetTableInWholeGroup} \).

\[ \text{SubgroupOfWholeGroupByCosetTable( } \text{fpfam, tab} \text{ )} \]

takes a family of an fp group and a coset table \( \text{tab} \) and returns the subgroup of \( \text{fam!}.\text{wholeGroup} \) defined by this coset table.

See also \( \text{CosetTableBySubgroup} \) (45.5.4).

45.8 Augmented Coset Tables and Rewriting

\[ \text{AugmentedCosetTableInWholeGroup( } H \text{ [, gens]} \text{ )} \]

For a subgroup \( H \) of a finitely presented group, this function returns an augmented coset table. If a generator set \( \text{gens} \) is given, it is guaranteed that \( \text{gens} \) 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.

\[ \text{AugmentedCosetTableMtc( } G, H, \text{ type, string} \text{ )} \]

is an internal function used by the subgroup presentation functions described in 46.3. It applies a Modified Todd-Coxeter coset representative enumeration to construct an augmented coset table (see 46.3) for the given subgroup \( H \) of \( G \). The subgroup generators will be named \( \text{string1, string2, ...} \).

The function accepts the options max and silent as described for the function \( \text{CosetTableFromGensAndRels} \) (see 45.5.5).

\[ \text{AugmentedCosetTableRrs( } G, \text{ table, type, string} \text{ )} \]

is an internal function used by the subgroup presentation functions described in 46.3. 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 \( \text{table} \). The new subgroup generators will be named \( \text{string1, string2, ...} \).

\[ \text{RewriteWord( } \text{aug, word} \text{ )} \]

RewriteWord rewrites \( \text{word} \) (which must be a word in the full group with respect to which the augmented coset table \( \text{aug} \) is given) in the subgroup generators given by the augmented coset table \( \text{aug} \). It returns a Tietze-type word (i.e. a list of integers), referring to the primary and secondary generators of \( \text{aug} \).

If \( \text{word} \) is not contained in the subgroup, fail is returned.
45.9 Low Index Subgroups

\( \text{LowIndexSubgroupsFpGroup}( G, [H, \text{index}[\text{excluded}]) \)

returns a list of 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 \( \text{index} \).

If the optional argument \text{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 \text{LowIndexSubgroupsFpGroup} returns only those subgroups of index at most \( \text{index} \) 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 function \text{LowIndexSubgroupsFpGroup} finds the requested subgroups by systematically running through a tree of all potential coset tables of \( G \) of length at most \( \text{index} \) (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 \( \text{index} \). So you should be careful with the choice of \( \text{index} \).

\[
\text{gap> li:=LowIndexSubgroupsFpGroup( g, TrivialSubgroup( g ), 10 );}
\[
[ \text{Group(<fp, no generators known>), Group(<fp, no generators known>),}
\\text{Group(<fp, no generators known>), Group(<fp, no generators known>) } ]
\]

By default, the algorithm computes no generating sets for the subgroups. This can be enforced with \text{GeneratorsOfGroup}:

\[
\text{gap> GeneratorsOfGroup(li[2]);}
\[
[ a, b*a*b^-1 ]
\]

As an example for an application of the optional parameter \text{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 \( y^2 \).)

\[
\text{gap> F := FreeGroup( "x", "y", "z" );;}
\text{gap> x := F.1;; y := F.2;; z := F.3;;}
\text{gap> G := F / [ x^2, y^4, z^3, (xy)^3, (yz)^2, (xz)^3 ];;}
\text{gap> torsion := [ x, y, y^2, z, x*y, x*z, y*z ];;}
\text{gap> SetInfoLevel( InfoFpGroup, 2 );}
\text{gap> lis := LowIndexSubgroupsFpGroup( G, TrivialSubgroup( G ), 24, torsion );;}
\#I LowIndexSubgroupsFpGroup called
\#I class 1 of index 24 and length 8
\#I class 2 of index 24 and length 24
\#I class 3 of index 24 and length 24
\#I class 4 of index 24 and length 24
\#I class 5 of index 24 and length 24
\#I LowIndexSubgroupsFpGroup returns 5 classes
\text{gap> SetInfoLevel( InfoFpGroup, 0 );}
\]

If a particular image group is desired, the operation \text{GQuotients} (see 45.13) can be useful as well.
45.10 Converting Groups to Finitely Presented Groups

1. \textbf{IsomorphismFpGroup} \((G)\)

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.

\[
\begin{align*}
g & := \text{Group}( (2,3,4,5), (1,2,5) );; \\
iso & := \text{IsomorphismFpGroup}( g ); \\
[ (2,5,4,3), (1,2,3,4,5), (1,3,2,4,5) ] & \rightarrow [ F1, F2, F3 ] \\
gp & := \text{Image}( iso ); \\
<\text{fp group of size } 120 \text{ on the generators } [ F1, F2, F3 ]> \\
\text{RelatorsOfFpGroup}( fp ); \\
\end{align*}
\]

2. \textbf{IsomorphismFpGroupByGenerators} \((G, gens[, string ])\)

\textbf{IsomorphismFpGroupByGeneratorsNC} \((G, gens, string )\)

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 \(G\).

\[
\begin{align*}
gp & := \text{MathieuGroup}( 12 ); \\
gp & := \text{GeneratorsOfGroup}( M12 );; \\
iso & := \text{IsomorphismFpGroupByGenerators}( M12, gens );; \\
#I & \text{the image group has } 3 \text{ gens and } 29 \text{ rels of total length } 1221
\end{align*}
\]

The main task of the function \textbf{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.
Section 10. Converting Groups to Finitely Presented Groups

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 to 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.

```gap
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 175 rels of total length 8149
```

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` 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.

```gap
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 13 rels of total length 168
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 ],
   [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1 ] ] ]->[ F1, F2 ]>
gap> ConstituentsCompositionMapping(iso);
[ [action isomorphism>, [ (1,7,6)(2,9)(4,5,10), (2,3,4,5)(6,9,8,7) ] ->
```

```
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` as separate GAP objects.

### 45.11 New Presentations and Presentations for Subgroups

`IsomorphismFpGroup` is also used to compute a new finitely presented group that is isomorphic to the subgroup of a given finitely presented group. (This is typically the only method to compute with subgroups of a finitely presented group.)

```gap
gap> f:=FreeGroup(2);;
gap> g:=f/[f.1^2,f.2^3,(f.1*f.2)^5];
gap> u:=Subgroup(g,[g.1*g.2]);
gap> hom:=IsomorphismFpGroup(u);
gap> new:=Range(hom);
gap> List(GeneratorsOfGroup(new),i->PreImagesRepresentative(hom,i));
```

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 `MappingGeneratorImages` and displayed when `Viewing` the homomorphism) as well as preimages under the homomorphism are stored in the form of straight line program elements (see 35.9). These will behave like ordinary words and no extra treatment should be necessary.

```gap
gap> r:=Range(hom).1^10;
gap> p:=PreImagesRepresentative(hom,r);
```

If desired, it also is possible to convert these underlying words using `EvalStraightLineProgElm`:

```gap
gap> r:=EvalStraightLineProgElm(UnderlyingElement(p));
gap> p:=ElementOfFpGroup(FamilyObj(p),r);
```

(If you are only interested in a finitely presented group isomorphic to $U$, but not in the isomorphism, you may also use the functions `PresentationViaCosetTable` and `FpGroupPresentation` (see 46.1).)

Homomorphisms can also be used to obtain an isomorphic finitely presented group with a (hopefully) simpler presentation.

```gap
1
```

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.
Section 12. Preimages under Homomorphisms from an FpGroup

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.

```gap
gap> f:=FreeGroup(3);;
gap> g:=f/[f.1^2,f.2^3,(f.1*f.2)^5,f.1/f.3];
group on the generators [ f1, f2, f3 ]
gap> hom:=IsomorphismSimplifiedFpGroup(g);  
[ f1, f2, f3 ] -> [ f1, f2, f1 ]
gap> Range(hom);
group on the generators [ f1, f2 ]
gap> RelatorsOfFpGroup(Range(hom));
[ f1^2, f2^3, f1*f2*f1*f2*f1*f2*f1*f2*f1*f2 ]
gap> RelatorsOfFpGroup(g);
[ f1^2, f2^3, f1*f2*f1*f2*f1*f2*f1*f2*f1*f2, f1*f3^-1 ]
```

(IsomorphismSimplifiedFpGroup uses Tietze transformations to simplify the presentation, see 46.2.1.)

45.12 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.

1► SubgroupOfWholeGroupByQuotientSubgroup( fpfam, Q, U )

takes a fp group family fpfam, a finitely generated group $Q$ such that the fp generators of fam can be mapped by an epimorphism $\phi$ onto GeneratorsOfGroup($Q$) and a subgroup $U$ of $Q$. It returns the subgroup of $\text{fam!.wholeGroup}$ which is the full preimage of $U$ under $\phi$.

2► IsSubgroupOfWholeGroupByQuotientRep( G )
is the representation for subgroups of an fp group, given by a quotient subgroup. The components $G!.quot$ and $G!.sub$ hold quotient, respectively subgroup.

3► AsSubgroupOfWholeGroupByQuotient( U )
returns the same subgroup in the representation AsSubgroupOfWholeGroupByQuotient.
See also SubgroupOfWholeGroupByCosetTable (45.7.2) and CosetTableBySubgroup (45.5.4).
This technique is used by GAP for example to represent the derived subgroup, which is obtained from the quotient $G'/G$.

```gap
gap> f:=FreeGroup(2);;
gap> g:=f/[f.1^6,f.2^6,(f.1*f.2)^6];;
gap> d:=DerivedSubgroup(g);
gap> Index(g,d);
36
```

4► DefiningQuotientHomomorphism( U )
if $U$ is a subgroup in quotient representation (IsSubgroupOfWholeGroupByQuotientRep), this function returns the defining homomorphism from the whole group to $U!.quot$. 

459
Chapter 45. Finitely Presented Groups

### 45.13 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.

**MaximalAbelianQuotient**(fpgroup)

As defined for general groups, this attribute returns the largest abelian quotient of fpgroup.

```
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
```

**PQuotient**(F, p[, c][, logord][, ctype])

F 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 45.13.2.

For a group G define the exponent-p central series of G inductively by \( P_1(G) = G \) and \( P_{i+1}(G) = [P_i(G), G]P_{i+1}(G)^p \). The factor groups modulo the terms of the lower exponent-p central series are p-groups.

The group G has p-class c if \( P_c(G) \neq P_{c+1}(G) = 1 \).

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 \( p^{\text{logord}} \). 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 \( p^{\text{logord}} \), 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.

**EpimorphismQuotientSystem**(quotsys)

For a quotient system quotsys obtained from the function 45.13.1, this operation returns an epimorphism \( F \rightarrow P \) where \( F \) is the finitely presented group of which quotsys is a quotient system and \( P \) is a PcGroup isomorphic to the quotient of \( F \) determined by quotsys.

Different calls to this operation will create different groups \( P \), each with its own family.

```
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 ] ]
```

**EpimorphismPGroup**(fpgrp, p)

**EpimorphismPGroup**(fpgrp, p, cl)

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.
Section 14. Abelian Invariants for Subgroups

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

EpimorphismNilpotentQuotient( fggrp [, n] )
returns an epimorphism on the class n finite nilpotent quotient of the finitely presented group fggrp. If n is omitted, the largest finite nilpotent quotient (of \( p \)-class up to 1000) is taken.

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 \texttt{GQuotients}, which computes all epimorphisms from a (finitely presented) group \( F \) onto a given (finite) group \( G \), see 38.9.2.

gap> GQuotients(fp,Group((1,2,3),(1,2)));
[ [ f1, f2 ] -> [ (1,3), (1,2) ], [ f1, f2 ] -> [ (1,2), (1,2,3) ],
 [ f1, f2 ] -> [ (1,2,3), (1,2) ] ]

4.5.14 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 \texttt{AbelianInvariants} 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.

\begin{itemize}
\item \texttt{AbelianInvariantsSubgroupFpGroup( G, H )}
\item \texttt{AbelianInvariantsSubgroupFpGroupMtc( G, H )}
\item \texttt{AbelianInvariantsSubgroupFpGroupRrs( G, H )}
\item \texttt{AbelianInvariantsNormalClosureFpGroup( G, H )}
\item \texttt{AbelianInvariantsNormalClosureFpGroupRrs( G, H )}
\end{itemize}

is a synonym for \texttt{AbelianInvariantsSubgroupFpGroupRrs(G,H)}.

uses the Modified Todd-Coxeter method to compute the abelian invariants of a subgroup \( H \) of a finitely presented group \( G \).

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 \texttt{table} in \( G \) may be given as second argument.

is a synonym for \texttt{AbelianInvariantsNormalClosureFpGroupRrs(G,H)}.

uses the Reduced Reidemeister-Schreier method to compute the abelian invariants of the normal closure of a subgroup \( H \) of a finitely presented group \( G \).
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 $B'_1/B''_1 \cong \mathbb{Z}_2^2 \times \mathbb{Z}^3$ and $B'_0/B''_0 \cong \mathbb{Z}_2^2 \times \mathbb{Z}^7$ as stated in Proposition 5 in [FJNT95].

```gap
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 ]
```

### 45.15 Testing Finiteness of Finitely Presented Groups

As a consequence of the algorithmic insolubilities 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 a finitely presented group for `IsFinite` can be problematic. What GAP actually does upon a call of `IsFinite` (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 \texttt{LowIndexSubgroupsFpGroup}) a subgroup $U$ such that $[U : U']$ is infinite. If $U$ has finite index, this can be checked by the operation \texttt{AbelianInvariants} (see section 45.14 for an example).

Another method is based on $p$-group quotients:

\begin{verbatim}
11\textbf{NewmanInfinityCriterion}( G, p )
\end{verbatim}

Let $G$ be a finitely presented group and $p$ a prime that divides the order of $G/G'$. This function applies an infinity criterion due to M.F. Newman [New90] to $G$. (See chapter 16 of [Joh97] for a more explicit description.) It returns \texttt{true} if the criterion succeeds in proving that $G$ is infinite and \texttt{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 (\texttt{IsomorphismSimplifiedFpGroup}) therefore might produce an isomorphic group, for which the criterion will work better.

\begin{verbatim}
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> AbelianInvariants(k);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
gap> NewmanInfinityCriterion(Kernel(hom),5);
true
\end{verbatim}

This proves that the subgroup $k$ (and thus the whole group $g$) is infinite. (This is the original example from [New90].)
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 46.6 to 46.9).

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` already does what you want (see 45.11).

### 46.1 Creating Presentations

1. `PresentationFpGroup( G [, printlevel] )`
   - 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.

   ```
   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>
   ```

   Most of the functions creating presentations and all functions performing Tietze transformations on them sort the relators by increasing lengths. The function `PresentationFpGroup` is an exception because it is intended to reflect the relators that were used to define the involved FpGroup. You may use the following command to sort the presentation.

2. `TzSort( P )`
   - 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.

3. `GeneratorsOfPresentation( P )`
   - returns a list of free generators that is a `ShallowCopy` of the current generators of the presentation `P`. 
Section 1. Creating Presentations

4► FpGroupPresentation( \( P \), nam )

constructs an FpGroup group as defined by the given Tietze presentation \( P \).

```gap
gap> h := FpGroupPresentation( p );
<fp group on the generators [ a, b ]>
```

5► PresentationViaCosetTable( \( G \) )

This 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 in contrast uses a multiple-stage algorithm using a composition 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 \( \text{words} \), where \( F \) is a free group with the same number of generators as \( G \), and \( \text{words} \) 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 \( G \).

```gap
gap> G := GeneralLinearGroup( 2, 7 );
GL(2,7)
```

```gap
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
gap> Size( G );
2016
```

```gap
gap> P := PresentationViaCosetTable( G );
<presentation with 2 gens and 5 rels of total length 46>
```

```gap
gap> TzPrintRelators( P );
#I 1. f2^-3
#I 2. f1^-6
#I 3. f1*f2*f1^-1*f2*f1^-1*f2^-1*f1^-1*f2*f1^-1*f2^-1
#I 4. f1*f2*f1^-1*f2*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1
#I 5. f1^-3*f2*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*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.

```gap
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
gap> F := FreeGroup( "a", "b", "c" );
<free group on the generators [ a, b, c ]>
```

```gap
gap> words := [ F.1, F.2 ];
[ a, b ]
```

```gap
gap> P := PresentationViaCosetTable( M12, F, words );
<presentation with 3 gens and 10 rels of total length 97>
```

```gap
gap> G := FpGroupPresentation( P );
<fp group on the generators [ a, b, c ]>
```
Chapter 46. Presentations and Tietze Transformations

Before it is returned, the resulting presentation is being simplified by appropriate calls of the function \texttt{SimplifyPresentation} (see 46.7), 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 \texttt{SimplifyPresentation}, now without this restriction, once more yourself.

If you apply the function \texttt{FpGroupPresentation} to the resulting presentation you will get a finitely presented group isomorphic to $G$. Note, however, that the function \texttt{IsomorphismFpGroup} (see 45.10.1) is recommended for this purpose.

### 46.2 SimplifiedFpGroup

\begin{verbatim}
SimplifiedFpGroup( G ) -> F
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 \texttt{IsomorphismSimplifiedFpGroup} (see 45.11.1) should be used instead.
\end{verbatim}

In fact, the command

\begin{verbatim}
H := SimplifiedFpGroup( G );
\end{verbatim}

is an abbreviation of the command sequence

\begin{verbatim}
P := PresentationFpGroup( G, 0 );
SimplifyPresentation( P );
H := FpGroupPresentation( P );
\end{verbatim}

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 46.7).
46.3 Subgroup Presentations

1. \texttt{PresentationSubgroup( G, H [, string] )}

is a synonym for \texttt{PresentationSubgroupRrs(G, H [, string])}.

2. \texttt{PresentationSubgroupRrs( G, H [, string] )}

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 \texttt{string1}, \texttt{string2}, ..., the default string is "x". You may access the \texttt{i}-th of these generators by \texttt{P!i}.

Alternatively to the subgroup \( H \), its coset table \texttt{table} in \( G \) may be given as second argument.

```
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 := PresentationSubgroupRrs( 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.

```
gap> gens[1] = g1;
false
```

```
gap> g1 := gens[1];
60
```

The Reduced Reidemeister-Schreier algorithm is a modification of the Reidemeister-Schreier algorithm of George Havas [Hav74]. It was proposed by Joachim Neub"auer 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 \texttt{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 46.7) or by the command DecodeTree (see 46.11.1) 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 the PresentationNormalClosureRrs command below).

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 of the resulting presentation.

3► `PrimaryGeneratorWords( P )`

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 $P$.

```gap
gap> PrimaryGeneratorWords( p );
[ a, b^-1*a*b ]
```

4► `PresentationSubgroupMtc( G, H [, string] [, print level] )`

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 $string1$, $string2$, ... , 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 will be suppressed.

```gap
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 45.5), 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
Section 3. Subgroup Presentations

$H$ 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` 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 46.11.1 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`. 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` described above (see there for details), the function `PresentationSubgroupMtc` returns a list of the primary subgroup generators of $H$ in the attribute `PrimaryGeneratorWords` of $P$. In fact, this list is not very exciting here because it is just a shallow copy of the attribute value `GeneratorsOfPresentation(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 46.7).

```
5 ★ PresentationNormalClosureRrs( G, H [, string] )
```

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 `string1, string2, ...`, the default string is "_x". You may access the $i$-th of these generators by $P!i$.

```
6 ★ PresentationNormalClosure( G, H [, string] )
```

is a synonym for `PresentationNormalClosureRrs(G, H [, string])`. 
46.4 Relators in a Presentation

In order to speed up the Tietze transformation routines, each relator in a presentation $P$ 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.

1. \texttt{TietzeWordAbstractWord( word, fgens )}

assumes \texttt{fgens} to be a list of free group generators and \texttt{word} to be an abstract word in these generators. It converts \texttt{word} into a Tietze word, i.e., a list of positive or negative generator numbers.

This function simply calls \texttt{LetterRepAssocWord}.

2. \texttt{AbstractWordTietzeWord( word, fgens )}

assumes \texttt{fgens} to be a list of free group generators and \texttt{word} to be a Tietze word in these generators, i.e., a list of positive or negative generator numbers. It converts \texttt{word} to an abstract word.

This function simply calls \texttt{AssocWordByLetterRep}.

\begin{verbatim}
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
\end{verbatim}

46.5 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.

1. \texttt{TzPrintGenerators( P [, list] )}

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.

\begin{verbatim}
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
\end{verbatim}

2. \texttt{TzPrintRelators( P[, list] )}

prints the relators of the given Tietze presentation $P$. The optional second argument \texttt{list} can be used to specify the numbers of the relators to be printed. Default: all relators are printed.
Section 5. Printing Presentations

gap> TzPrintRelators( P );
#I 1. f3^-2
#I 2. f2^-4
#I 3. f2*f3*f2*f3
#I 4. f1^-5
#I 5. f1^-2*f2*f1*f2^-1
#I 6. f1*f2^-2*f3*f1*f3*f1^-1*f3

3▶ TzPrintLengths( P ) F
prints just a list of all relator lengths of the given presentation P.

gap> TzPrintLengths( P );
[ 2, 4, 4, 5, 5, 8 ]

4▶ TzPrintStatus( P [, norepeat ] ) F
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.

gap> TzPrintStatus( P );
#I there are 3 generators and 6 relators of total length 28

5▶ TzPrintPresentation( P ) F
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( P ), TzPrintRelators( P ), and TzPrintStatus( P ).

6▶ TzPrint( P [, list] ) F
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 and TzPrintPresentation do). The optional second argument can be used to specify the numbers of the relators to be printed. Default: all relators are printed.

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, -2, -2, 3, 1, 3, -1, 3 ]

7▶ TzPrintPairs( P [, n] ) F
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.

The function TzPrintPairs is useful in the context of Tietze transformations which introduce new generators by substituting words in the current generators (see 46.9). It gives some evidence for an appropriate choice of a word of length 2 to be substituted.

gap> TzPrintPairs( P, 3 );
#I 1. 3 occurrences of f2^-1 * f3
#I 2. 2 occurrences of f2 * f3
#I 3. 2 occurrences of f1^-1 * f3

Finally, there is a function TzPrintOptions. It is described in section 46.12.
46.6 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.

1 ▶ AddGenerator( P )

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 \( \text{GeneratorsOfPresentation}(P)[\text{Length}(\text{GeneratorsOfPresentation}(P))] \).

2 ▶ TzNewGenerator( P )

is an internal function which defines a new abstract generator and adds it to the presentation \( P \). It is called by AddGenerator 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, if needed.

3 ▶ AddRelator( P, word )

adds the relator \( \text{word} \) to the presentation \( P \), probably changing the group defined by \( P \). \( \text{word} \) must be an abstract word in the generators of \( P \).

4 ▶ RemoveRelator( P, n )

removes the \( n \)-th relator from the presentation \( P \), probably changing the group defined by \( P \).

46.7 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]).

1 ▶ TzGo( P [, silent] ) 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 46.12) has a value less than 2.

2 ▶ SimplifyPresentation( P ) is a synonym for TzGo(P).

\[
\text{gap> F2 := FreeGroup( "a", "b" );;}
\text{gap> G := F2 / [ F2.1^9, F2.2^2, (F2.1^F2.2)^4, (F2.1^2*F2.2)^3 ];;}
\text{gap> a := G.1;; b := G.2;;}
\text{gap> H := Subgroup( G, [ (a*b)^2, (a^-1*b)^2 ] );;}
\text{gap> Index( G, H ); 408}
\text{gap> P := PresentationSubgroup( G, H );}
\text{<presentation with 8 gens and 36 rels of total length 111>}
\text{gap> PrimaryGeneratorWords( P ); [ b, a*b*a ]}
\text{gap> TzOptions( P ),protected := 2; 2}
\text{gap> TzOptions( P ),printLevel := 2; 2}
\text{gap> SimplifyPresentation( P );}
\text{#I eliminating _x7 = _x5^-1}
\text{#I eliminating _x5 = _x4}
\text{#I eliminating _x18 = _x3}
\text{#I eliminating _x8 = _x3}
\text{#I there are 4 generators and 8 relators of total length 21}
\text{#I there are 4 generators and 7 relators of total length 18}
\text{#I eliminating _x4 = _x3^-1_* _x2^-1}
\text{#I eliminating _x3 = _x2*_* _x1^-1}
\text{#I there are 2 generators and 4 relators of total length 14}
\text{#I there are 2 generators and 4 relators of total length 13}
\text{#I there are 2 generators and 3 relators of total length 9}
\text{gap> TzPrintRelators( P );}
\text{#I 1. _x1^2}
\text{#I 2. _x2^3}
\text{#I 3. _x2*_* _x1* _x2* _x1}
\]

Roughly speaking, TzGo consists of a loop over a procedure which involves two phases: In the search phase it calls TzSearch and TzSearchEqual described below which try to reduce the relator lengths by substituting common subwords of relators, in the elimination phase it calls the command TzEliminate described below (or, more precisely, a subroutine of TzEliminate in order to save some administrative overhead) which tries to eliminate generators that can be expressed as words in the remaining generators.
If \texttt{TzGo} 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 \texttt{TzSearchEqual} involved has changed the presentation such that another call of \texttt{TzGo} 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 \texttt{TzGoGo} instead).

3 \texttt{\textbackslash TzGoGo( P )}\n
The result of the Tietze transformations can be affected substantially by the options parameters (see 46.12). To demonstrate the effect of the \texttt{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 \texttt{TzGoGo} for different values of the parameter \texttt{eliminationsLimit} (including the default value 100). In fact, we also alter the \texttt{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 \texttt{TzGoGo} again, i.e., they are the best results which we can get without substituting new generators.

```gap
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,
> (F3.2*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\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 \texttt{saveLimit} parameter by just continuing the preceding example for some different values of the \texttt{saveLimit} parameter (including its default value 10), but without changing the \texttt{eliminationsLimit} parameter which keeps its default value 100.
Section 8. Elementary Tietze Transformations

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

46.8 Elementary Tietze Transformations

1. TzEliminate( P )
2. TzEliminate( P, gen )
3. TzEliminate( P, n )

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 46.12). The default value of that parameter is infinity, 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 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.

2. TzSearch( P )

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 46.12 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.

3 TzSearchEqual( P )

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. 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. However, TzSearchEqual does not use the parameter saveLimit: The loop over the relators is executed exactly once.

4 TzFindCyclicJoins( P )

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 (see 46.9.2).

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 Euclidian algorithm to express \( a \) as a power of \( b \), and then it eliminates \( a \).
46.9 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 \texttt{TzSubstitute} and \texttt{TzSubstituteCyclicJoins} which introduce new generators.

\begin{verbatim}
   TzSubstitute( P, word )
   TzSubstitute( P [, n [, eliminate ]] )
\end{verbatim}

In the first form \texttt{TzSubstitute} expects \(P\) to be a presentation and \textit{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, \texttt{TzSubstitute} creates a new abstract generator, \(g\) say, and adds it to the presentation, then it adds a new relator \(g^{-1} \cdot \text{word}\).

In its second form, \texttt{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 \textit{eliminate} are optional. If you specify arguments for them, then \(n\) is expected to be a positive integer, and \textit{eliminate} is expected to be 0, 1, or 2. The default values are \(n = 1\) and \textit{eliminate} = 0. \texttt{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. \texttt{TzSubstitute} defines a new abstract generator \(x_i\) named "\(x_i^n" and adds it to \(P\) (see \texttt{AddGenerator}). 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 \textit{eliminate}:

If \textit{eliminate} is zero, \texttt{TzSubstitute} just calls the function \texttt{TzEliminate}. 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 \texttt{TzSubstitute} with \textit{eliminate} = 0 must not increase the total length of the relators.

Otherwise, if \textit{eliminate} is 1 or 2, \texttt{TzSubstitute} eliminates the respective factor of the substituted word \(ab\), i. e., it eliminates \(a\) if \textit{eliminate} = 1 or \(b\) if \textit{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 \texttt{TzSubstitute} which should be mentioned. If, for instance, \textit{word} is an abstract word, a call

\begin{verbatim}
   TzSubstitute( P, word );
\end{verbatim}

is more or less equivalent to

\begin{verbatim}
   AddGenerator( P );
   g := GeneratorsOfPresentation(P)[Length(GeneratorsOfPresentation(P))];
   AddRelator( P, g^-1 * word );
\end{verbatim}

However, there is a difference: If you are tracing generator images and preimages of \(P\) through the Tietze transformations applied to \(P\) (see 46.10), then \texttt{TzSubstitute}, as a Tietze transformation of \(P\), will update and save the respective lists, whereas a call of the function \texttt{AddGenerator} (which does not perform a Tietze transformation) will delete these lists and hence terminate the tracing.
Chapter 46. Presentations and Tietze Transformations

```gap
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 $J_1$.

```gap
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> TzGo( P );
#I there are 2 generators and 46 relators of total length 2568
```

Here we do not get any more progress without substituting a new generator.

```gap
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
Section 9. Tietze Transformations that introduce new Generators

the arguments \( n \) and \textit{eliminate}. For this purpose it may be helpful to print out a list of the most frequently occurring squarefree relator subwords of length 2.

```gap
gap> TzPrintPairs( P );
#I 1. 504 occurrences of \( _x6 \cdot _x23^{-1} \)
#I 2. 504 occurrences of \( _x6^{-1} \cdot _x23 \)
#I 3. 448 occurrences of \( _x6 \cdot _x23 \)
#I 4. 448 occurrences of \( _x6^{-1} \cdot _x23^{-1} \)
```

If we save a copy of the current presentation, then later we will be able to restart the computation from the current state.

```gap
gap> P1 := ShallowCopy( P );
<presentation with 2 gens and 10 rels of total length 101>
```

Just for demonstration we make an inconvenient choice:

```gap
gap> TzSubstitute( P, 3, 1 );
#I substituting new generator \( _x31 \) defined by \( _x23 \cdot _x30 \)
#I eliminating \( _x23 = _x31 \cdot _x30^{-1} \)
#I there are 2 generators and 10 relators of total length 122
```

This presentation is worse than the one we have saved, so we restart from that presentation again.
Chapter 46. Presentations and Tietze Transformations

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*_x31^-1*_x32^-1*_x31^-1*_x32^-1
#I 4. _x31*_x32*_x31^-1*_x32*_x31^-1*_x32*_x31^-1*_x32^-2
#I 5. _x31^-1*_x32^-2*_x31^-1*_x32^-2*_x31^-1*_x32^-1*_x31^-1*_x32^-2

2 ▶ TzSubstituteCyclicJoins( P )

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.

46.10 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, $\varphi$ 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 $\varphi$. 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.

1 ▶ TzInitGeneratorImages( P )

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 and TzPreImagesNewGens 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 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 described in section 46.6, this new generator cannot be traced in the old generators. Therefore AddGenerator will terminate the tracing of the generator images and preimages and delete the respective lists whenever it is called.

2 ▶ OldGeneratorsOfPresentation( P )

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 $P$. 

Section 10. Tracing generator images through Tietze transformations

3 ▶ TzImagesOldGens( \( P \) )

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) generators \( \text{GeneratorsOfPresentation}(P) \) of \( P \) such that the \( i \)-th word \( l[i] \) represents the \( i \)-th old generator \( \text{OldGeneratorsOfPresentation}(P)[i] \) of \( P \).

4 ▶ TzPreImagesNewGens( \( P \) )

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 \( \text{OldGeneratorsOfPresentation}(P) \) of \( P \) such that the \( i \)-th word \( l[i] \) represents the \( i \)-th (current) generator \( \text{GeneratorsOfPresentation}(P)[i] \) of \( P \).

5 ▶ TzPrintGeneratorImages( \( P \) )

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.

```
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
```
46.11 DecodeTree

Assumes that \( P \) is a subgroup presentation provided by the Reduced Reidemeister-Schreier or by the Modified Todd-Coxeter method (see \texttt{PresentationSubgroupRrs}, \texttt{PresentationNormalClosureRrs}, \texttt{PresentationSubgroupMtc} in section 46.3). It eliminates the secondary generators of \( P \) (see 46.3) by applying the so-called “decoding tree” procedure.

\texttt{DecodeTree} is called automatically by the command \texttt{PresentationSubgroupMtc} (see 46.3.4) 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 46.3. 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 relations 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 \texttt{DecodeTree} and hence by the command \texttt{PresentationSubgroupMtc}.

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 (\texttt{PresentationSubgroupRrs} or \texttt{PresentationNormalClosureRrs}) just returns \( P_0 \).

As mentioned in section 46.3, 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 or TzGoGo. 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 $P!\text{.tree}$ 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 the DecodeTree command 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 46.7 apply to the DecodeTree command as well. Hence, in case of inadequate values of these parameters, it may happen that the DecodeTree routine 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 the DecodeTree command 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 the DecodeTree command it is convenient to further simplify the resulting presentation by suitable Tietze transformations.

As an example of an explicit call of the DecodeTree command 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 command with its default parameters, whereas in the second run we call the DecodeTree command before.

```gap
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 $H$.

```gap
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.

```gap
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
```
Chapter 46. Presentations and Tietze Transformations

This time we end up with a shorter presentation. As an example of an implicit call of the function `DecodeTree` via the command `PresentationSubgroupMtc` 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 FpGroup info level to get some additional output.

```gap
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 4446 secondary subgroup generators
  #I there are 246 generators and 617 relators of total length 2893
  #I calling DecodeTree
  #I there are 115 generators and 382 relators of total length 1837
  #I there are 69 generators and 298 relators of total length 1785
  #I there are 44 generators and 238 relators of total length 1767
  #I there are 35 generators and 201 relators of total length 2030
  #I there are 26 generators and 177 relators of total length 2084
  #I there are 23 generators and 167 relators of total length 2665
  #I there are 20 generators and 158 relators of total length 2848
  #I there are 20 generators and 148 relators of total length 3609
  #I there are 21 generators and 148 relators of total length 5170
  #I there are 24 generators and 148 relators of total length 7545
  #I there are 27 generators and 146 relators of total length 11477
  #I there are 32 generators and 146 relators of total length 18567
  #I there are 36 generators and 146 relators of total length 25440
  #I there are 39 generators and 146 relators of total length 38070
  #I there are 43 generators and 146 relators of total length 54000
  #I there are 41 generators and 143 relators of total length 64970
  #I there are 8 generators and 129 relators of total length 20031
  #I there are 7 generators and 125 relators of total length 27614
  #I there are 4 generators and 113 relators of total length 36647
  #I there are 3 generators and 108 relators of total length 44128
  #I there are 2 generators and 103 relators of total length 35394
  #I there are 2 generators and 102 relators of total length 34380
  gap> TzGoGo( P );
  #I there are 2 generators and 101 relators of total length 19076
  #I there are 2 generators and 94 relators of total length 6552
  #I there are 2 generators and 38 relators of total length 1344
  #I there are 2 generators and 9 relators of total length 94
  #I there are 2 generators and 8 relators of total length 86
  gap> TzPrintGenerators( P );
  #I 1. _x1  43 occurrences
```
Section 12. Tietze Options

46.12 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.

\begin{verbatim}
\texttt{TzOptions( P )}
\end{verbatim}

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:

\textbf{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 \texttt{TzEliminate(\( P, \) gen)} and \texttt{TzSubstitute(\( P, n, eliminate \))} because they explicitly specify the generator to be eliminated. The default value of protected is 0.

\textbf{eliminationsLimit:}

Whenever the elimination phase of the \texttt{TzGo} 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 \texttt{TzGo} command. Note, however, that it is ignored by the \texttt{TzEliminate} command. The default value of eliminationsLimit is 100.

\textbf{expandLimit:}

Whenever the routine for eliminating more than 1 generators is called for a presentation \( P \) by the \texttt{TzEliminate} command or the elimination phase of the \texttt{TzGo} 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 \texttt{expandLimit} parameter as a break criterion for the \texttt{TzGo} command. The default value of \texttt{expandLimit} is 150.

\textbf{generatorsLimit:}
Whenever the elimination phase of the \texttt{TzGo} command is entered for a presentation $P$ with $n$ generators, then it will eliminate at most $n - \texttt{generatorsLimit}$ generators (except for generators which turn out to be trivial). Hence you may use the \texttt{generatorsLimit} parameter as a break criterion for the \texttt{TzGo} command. The default value of \texttt{generatorsLimit} is 0.

\textbf{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 \texttt{lengthLimit}. The default value of \texttt{lengthLimit} is $2^{31} - 1$.

\textbf{loopLimit:}
Whenever the \texttt{TzGo} command is called for a presentation $P$, then it will loop over at most \texttt{loopLimit} of its basic steps. Hence you may use the \texttt{loopLimit} parameter as a break criterion for the \texttt{TzGo} command. The default value of \texttt{loopLimit} is \textit{infinity}.

\textbf{printLevel:}
Whenever Tietze transformation commands are called for a presentation $P$ with $\texttt{printLevel} = 0$, they will not provide any output except for error messages. If $\texttt{printLevel} = 1$, 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 $\texttt{printLevel} = 2$, you will get a much more generous amount of output. Finally, if $\texttt{printLevel} = 3$, various messages on internal details will be added. The default value of \texttt{printLevel} is 1.

\textbf{saveLimit:}
Whenever the \texttt{TzSearch} 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 \texttt{saveLimit} per cent. If this is the case, then \texttt{TzSearch} repeats its procedure instead of returning. Hence you may use the \texttt{saveLimit} parameter as a break criterion for the \texttt{TzSearch} command and, in particular, for the search phase of the \texttt{TzGo} command. The default value of \texttt{saveLimit} is 10.

\textbf{searchSimultaneous:}
Whenever the \texttt{TzSearch} or the \texttt{TzSearchEqual} command is called for a presentation $P$, then it is allowed to handle up to \texttt{searchSimultaneous} short relators simultaneously (see for the description of the \texttt{TzSearch} command for more details). The choice of this parameter may heavily influence the performance as well as the result of the \texttt{TzSearch} and the \texttt{TzSearchEqual} commands and hence also of the search phase of the \texttt{TzGo} command. The default value of \texttt{searchSimultaneous} is 20.

\begin{verbatim}
F
2> TzPrintOptions( P )

prints the current values of the Tietze options of the presentation $P$.

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
\end{verbatim}
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 and WreathProductProductAction which are construction that makes sense only for permutation groups, see 47.4.1).

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 47.5);

### 47.1 Direct Products

The direct product of groups is the cartesian product of the groups (considered as element sets) with component-wise multiplication.

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 of groups and one of these groups. (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.

For a product $P$ the operation Embedding($P$, $nr$) returns the homomorphism embedding the $nr$-th factor into $P$. The operation Projection($P$, $nr$) gives the projection of $P$ onto the $nr$-th factor (see 47.5).

```plaintext
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)
```
47.2 Semidirect Products

The semidirect product of a group $N$ with a group $G$ acting on $N$ via a homomorphism $\alpha$ from $G$ into the automorphism group of $N$ is the cartesian product $G \times N$ with the multiplication $(g, n) \cdot (h, m) = (gh, n^{\alpha(h)}m)$.

\begin{verbatim}
SemidirectProduct( G, alpha, N )
SemidirectProduct( autgp, N )
\end{verbatim}

constructs the semidirect product of $N$ with $G$ acting via $alpha$. $alpha$ 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 automorphisms of $N$, it is a shorthand for $\text{SemidirectProduct}(autgp, \text{IdentityMapping}(autgp), N)$. Note that (unless $autgrp$ has been obtained by the operation $\text{AutomorphismGroup}$) you have to test $\text{IsGroupOfAutomorphisms}(autgrp)$ to ensure that GAP knows that $autgrp$ consists of group automorphisms.

\begin{verbatim}
gap> n:=AbelianGroup(IsPcGroup,[5,5]);
<pc group of size 25 with 2 generators>
gap> au:=DerivedSubgroup(AutomorphismGroup(n));;
gap> Size(au);
120
\end{verbatim}

\begin{verbatim}
gap> p:=SemidirectProduct(au,n);
<permutation group with 4 generators>
gap> Size(p);
3000
\end{verbatim}

\begin{verbatim}
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>
\end{verbatim}

\begin{verbatim}
gap> SemidirectProduct(au,n);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
\end{verbatim}

\begin{verbatim}
gap> IsGroupOfAutomorphisms(au);
true
\end{verbatim}

\begin{verbatim}
gap> SemidirectProduct(au,n);
<pc group with 3 generators>
\end{verbatim}

\begin{verbatim}
gap> n:=AbelianGroup(IsPcGroup,[2,2]);
<pc group of size 4 with 2 generators>
gap> au:=AutomorphismGroup(n);
<group with 4 generators>
gap> apc:=IsomorphismPcGroup(au);
CompositionMapping( Pcgs([ (2,3), (1,3,2) ] ) -> [ f1, f2 ], 
[ action isomorphism ] )
gap> g:=Image(apc);
Group([ f1, f2 ])
\end{verbatim}
Section 3. Subdirect Products

For the semidirect product \( P \) of \( G \) with \( N \), Embedding\((P,1)\) embeds \( G \), Embedding\((P,2)\) embeds \( N \). The operation Projection\((P)\) returns the projection of \( P \) onto \( G \) (see 47.5).

\begin{verbatim}
    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 ... ]
\end{verbatim}

\subsection*{47.3 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 diagram:

\[
\begin{array}{ccc}
G & \xrightarrow{\varphi} & A \\
\downarrow \ & \ & \\
H & \xrightarrow{\psi} & A
\end{array}
\]

The operation SubdirectProduct\((G, H, G\text{hom}, H\text{hom})\) constructs the subdirect product of \( G \) and \( H \) with respect to the epimorphisms \( G\text{hom} \) from \( G \) onto a group \( A \) and \( H\text{hom} \) from \( H \) onto the same group \( A \).

For a subdirect product \( P \), the operation Projection\((P, nr)\) returns the projections on the \( nr \)-th factor. (In general the factors do not embed in a subdirect product.)
Chapter 47. Group Products

47.4 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.

\[ \text{WreathProduct}( G, P ) \]

\[ \text{WreathProduct}( G, H \ [, \ hom] ) \]

constructs the wreath product of the group $G$ with the permutation group $P$ (acting on its MovedPoints). The second usage constructs the wreath product of the group $G$ with the image of the group $H$ under $hom$ where $hom$ must be a homomorphism from $H$ into a permutation group. (If $hom$ is not given, and $P$ is not a permutation group the result of $\text{IsomorphismPermGroup}(P)$ – whose degree may be dependent on the method and thus is not well-defined! – is taken for $hom$).

For a wreath product $W$ of $G$ with a permutation group $P$ of degree $n$ and $1 \leq nr \leq n$ the operation $\text{Embedding}(W, nr)$ provides the embedding of $G$ in the $nr$-th component of the direct product of the base group $G^n$ of $W$. $\text{Embedding}(W, n+1)$ is the embedding of $P$ into $W$. The operation $\text{Projection}(W)$ provides the projection onto the acting group $P$ (see 47.5).

\[ g := \text{Group}((1,2,3),(1,2)); \]
\[ \text{Group}([ (1,2,3), (1,2) ]) \]

\[ \text{hom} := \text{GroupHomomorphismByImagesNC}(g,g,[[(1,2,3),(1,2)],[()],[1,2]]); \]
\[ [ (1,2,3), (1,2) ] \rightarrow [ (), (1,2) ] \]

\[ s := \text{SubdirectProduct}(g,g,\text{hom},\text{hom}); \]
\[ \text{Group}([ (1,2,3), (1,2)(4,5), (4,5,6) ]) \]

\[ \text{Size}(s); \]
\[ 18 \]

\[ p := \text{Projection}(s,2); \]
\[ 2\text{nd projection of Group}([ (1,2,3), (1,2)(4,5), (4,5,6) ]) \]

\[ \text{Image}(p,(1,3,2)(4,5,6)); \]
\[ (1,2,3) \]

\[ w := \text{WreathProduct}(g,p); \]
\[ \text{Group}([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8), (1,4,7)(2,5,8)(3,6,9) ]) \]

\[ \text{Size}(w); \]
\[ 648 \]

\[ \text{Embedding}(w,1); \]
\[ 1\text{st embedding into Group}([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8), (1,4,7)(2,5,8)(3,6,9) ]) \]

\[ \text{Image}(\text{Embedding}(w,3)); \]
\[ \text{Group}([ (7,8,9), (7,8) ]) \]

\[ \text{Image}(\text{Embedding}(w,4)); \]
\[ \text{Group}([ (1,4,7)(2,5,8)(3,6,9) ]) \]

\[ \text{Image}(\text{Projection}(w),(1,4,8,2,6,7,3,5,9)); \]
\[ (1,2,3) \]

\[ \text{WreathProductImprimitiveAction}( G, H ) \]

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 and Projection operate on this product as described for general wreath products.

```gap
w := WreathProductProductAction(g, p);
LargestMovedPoint(w);
```

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 and Projection operate on this product as described for general wreath products.

```gap
w := WreathProductProductAction(g, p);
LargestMovedPoint(w);
```

47.5 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.

1. Embedding($P$, $nr$)

returns the $nr$-th embedding in the group product $P$. The actual meaning of this embedding is described in the section for the appropriate product.

2. Projection($P[, nr]$)

returns the $(nr$-th) projection of the group product $P$. The actual meaning of the projection returned is described in the section for the appropriate product.
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 48.1),
- Classical matrix groups (see 48.2),
- the transitive permutation groups of degree at most 30 (see 48.6),
- a library of groups of small order (see 48.7),
- the finite perfect groups of size at most $10^6$ (excluding 11 sizes) (see 48.8).
- a library of primitive permutation groups of degree < 1000, complete for degrees < 256 (see 48.9),
- the irreducible solvable subgroups of $GL(n, p)$ for $n > 1$ and $p^n < 256$ (see 48.10),
- the irreducible maximal finite integral matrix groups of dimension at most 31 (see 48.11),

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.

### 48.1 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, IsMatrixGroup or IsPcGroup, 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 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 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.

1\> TrivialGroup( [filter] ) \>

constructs a trivial group in the category given by the filter filter. If filter is not given it defaults to IsPcGroup.

```gap
gap> TrivialGroup();
<pc group of size 1 with 0 generators>
gap> TrivialGroup( IsPermGroup );
Group(())
```

2\> CyclicGroup( [filt, ] n ) \>

constructs the cyclic group of size n in the category given by the filter filt. If filt is not given it defaults to IsPcGroup.

```gap
```

```gap
```
Section 1. Basic Groups

```gap
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)
```

### AbelianGroup( [filt, ]ints )

Constructs an abelian group in the category given by the filter `filt` which is of isomorphism type \( C_{\text{ints}[1]} \times C_{\text{ints}[2]} \times \cdots \times C_{\text{ints}[n]} \). `ints` must be a list of positive integers. If `filt` is not given it defaults to `IsPcGroup`. The generators of the group returned are the elements corresponding to the integers in `ints`.

```gap
gap> AbelianGroup([1,2,3]);
<pc group of size 6 with 3 generators>
```

### ElementaryAbelianGroup( [filt, ]n )

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`.

```gap
gap> ElementaryAbelianGroup(8192);
<pc group of size 8192 with 13 generators>
```

### DihedralGroup( [filt, ]n )

Constructs the dihedral group of size \( n \) in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPcGroup`.

```gap
gap> DihedralGroup(10);
<pc group of size 10 with 2 generators>
```

### ExtraspecialGroup( [filt, ]order, exp )

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`.

```gap
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>
```

### AlternatingGroup( [filt, ]deg )

Constructs the alternating group of degree `deg` in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPermGroup`. 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.

```gap
```

3 ▶ AbelianGroup( [filt, ]ints ) F

4 ▶ ElementaryAbelianGroup( [filt, ]n ) F

5 ▶ DihedralGroup( [filt, ]n ) F

6 ▶ ExtraspecialGroup( [filt, ]order, exp ) F

7 ▶ AlternatingGroup( [filt, ]deg ) F
```

\[ \text{AlternatingGroup( [filt, ]dom )} \]
Chapter 48. Group Libraries

\begin{verbatim}
gap> AlternatingGroup(5);
Alt( [ 1 .. 5 ] )
\end{verbatim}

\begin{verbatim}
SymmetricGroup( [filt, ]deg )
SymmetricGroup( [filt, ]dom )
\end{verbatim}

constructs the symmetric group of degree \textit{deg} in the category given by the filter \textit{filt}. If \textit{filt} is not given it defaults to \texttt{IsPermGroup}. In the second version, the function constructs the symmetric group on the points given in the set \textit{dom} which must be a set of positive integers.

\begin{verbatim}
gap> SymmetricGroup(10);
Sym( [ 1 .. 10 ] )
\end{verbatim}

Note that permutation groups provide special treatment of symmetric and alternating groups, see 41.3.

\begin{verbatim}
MathieuGroup( [filt, ]degree )
\end{verbatim}

constructs the Mathieu group of degree \textit{degree} in the category given by the filter \textit{filt}, where \textit{degree} must be in \{9,10,11,12,21,22,23,24\}. If \textit{filt} is not given it defaults to \texttt{IsPermGroup}.

\begin{verbatim}
gap> MathieuGroup( 11 );
Group( [ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6) ] )
\end{verbatim}

\begin{verbatim}
SuzukiGroup( [filt, ]q )
\end{verbatim}

Constructs a group isomorphic to the Suzuki group \texttt{Sz(q)} over the field with \textit{q} elements, where \textit{q} $\geq 8$ is a non-square power of 2.

If \textit{filt} is not given it defaults to \texttt{IsMatrixGroup}, and the returned group is the Suzuki group itself.

\begin{verbatim}
gap> SuzukiGroup( 32 );
Sz(32)
\end{verbatim}

\begin{verbatim}
ReeGroup( [filt, ]q )
\end{verbatim}

Constructs a group isomorphic to the Ree group \texttt{2G2(q)} where \textit{q} = $3^{1+2m}$ for \textit{m} > 0.

If \textit{filt} is not given it defaults to \texttt{IsMatrixGroup} and the generating matrices are based on [KLM01]. (No particular choice of a generating set is guaranteed.)

\begin{verbatim}
gap> ReeGroup( 27 );
Ree(27)
\end{verbatim}

48.2 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 $SU(2,q)$ and $SL(2,q)$ is used, see for example [Hup67]. The generators of the orthogonal groups are taken from [IE94] and [KL90], except that the generators of the orthogonal groups in odd dimension in even characteristic are constructed via the isomorphism to a symplectic group, see for example [Car72].

For symplectic and orthogonal matrix groups returned by the functions described below, the invariant bilinear form is stored as the value of the attribute \texttt{InvariantBilinearForm} (see 42.4.1). Analogously, the invariant sesquilinear form defining the unitary groups is stored as the value of the attribute \texttt{InvariantSesquilinearForm} (see 42.4.3). The defining quadratic form of orthogonal groups is stored as the value of the attribute \texttt{InvariantQuadraticForm} (see 42.4.5).
Section 2. Classical Groups

1. \texttt{GeneralLinearGroup( \texttt{filt, }d, q \texttt{) }}
   \texttt{GL( \texttt{filt, }d, q \texttt{) }}
   \texttt{GeneralLinearGroup( }d, \texttt{Integers }\texttt{)}
   \texttt{GL( }d, \texttt{Integers }\texttt{)}
   \texttt{GeneralLinearGroup( }d, \texttt{Integers mod }m\texttt{) }}
   \texttt{GL( }d, \texttt{Integers mod }m\texttt{) }

   The first two forms construct a group isomorphic to the general linear group \( \text{GL}(d, q) \) of all \( d \times d \) matrices over the field with \( q \) elements, in the category given by the filter \texttt{filt}.

   If \texttt{filt} is not given it defaults to \texttt{IsMatrixGroup}, and the returned group is the general linear group as a matrix group in its natural action (see also 42.3.2).

   The third and the fourth form construct the group of all integer \( d \times d \) matrices invertible within the integers (see also 42.5.4).

   The last two forms construct the general linear group of degree \( d \) over the residue class ring \( \mathbb{Z}/m\mathbb{Z} \).

   \begin{verbatim}
   gap> GL(4,3);
   GL(4,3)
   gap> GL(2,Integers);
   GL(2,Integers)
   gap> GL(3,Integers mod 12);
   GL(3,\mathbb{Z}/12\mathbb{Z})
   \end{verbatim}

2. \texttt{SpecialLinearGroup( \texttt{filt, }d, q \texttt{) }}
   \texttt{SL( \texttt{filt, }d, q \texttt{) }}
   \texttt{SpecialLinearGroup( }d, \texttt{Integers }\texttt{)}
   \texttt{SL( }d, \texttt{Integers }\texttt{)}
   \texttt{SpecialLinearGroup( }d, \texttt{Integers mod }m\texttt{) }}
   \texttt{SL( }d, \texttt{Integers mod }m\texttt{) }

   The first two forms construct a group isomorphic to the special linear group \( \text{SL}(d, q) \) of all those \( d \times d \) matrices over the field with \( q \) elements whose determinant is the identity of the field, in the category given by the filter \texttt{filt}.

   If \texttt{filt} is not given it defaults to \texttt{IsMatrixGroup}, and the returned group is the special linear group as a matrix group in its natural action (see also 42.3.4).

   The third and the fourth form construct the group of all integer \( d \times d \) matrices of determinant 1 (see also 42.5.5).

   The last two forms construct the group of all \( d \times d \) matrices over \( \mathbb{Z}/m\mathbb{Z} \) with determinant 1 (mod \( m \)).

   \begin{verbatim}
   gap> SpecialLinearGroup(2,2);
   SL(2,2)
   gap> SL(3,Integers);
   SL(3,\mathbb{Z})
   gap> SL(4,Integers mod 4);
   SL(4,\mathbb{Z}/4\mathbb{Z})
   \end{verbatim}

   Using the \texttt{OnLines} operation it is possible to obtain the corresponding projective groups in a permutation action:
Chapter 48. Group Libraries

gap> g:=GL(4,3);;Size(g);
24261120
gap> pgl:=Action(g,Orbit(g,Z(3)^0*[1,0,0,0],OnLines),OnLines);;
Size(pgl);
12130560

3. GeneralUnitaryGroup( [filt, ]d, q )
   GU( [filt, ]d, q )
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, and the returned group is the general unitary group itself.

gap> GeneralUnitaryGroup( 3, 5 );
GU(3,5)

4. SpecialUnitaryGroup( [filt, ]d, q )
   SU( [filt, ]d, q )
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, and the returned group is the special unitary group itself.

gap> SpecialUnitaryGroup( 3, 5 );
SU(3,5)

5. SymplecticGroup( [filt, ]d, q )
   Sp( [filt, ]d, q )
   SP( [filt, ]d, q )
constructs a group isomorphic to the symplectic group Sp( d, q ) of those $d \times d$ matrices over the field with $q$ elements that respect a fixed nondegenerate symplectic form, in the category given by the filter filt.

If filt is not given it defaults to IsMatrixGroup, and the returned group is the symplectic group itself.

gap> SymplecticGroup( 4, 2 );
Sp(4,2)

6. GeneralOrthogonalGroup( [filt, ][e, ]d, q )
   GO( [filt, ][e, ]d, q )
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 42.4.5) 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, 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.

7. SpecialOrthogonalGroup( [filt, ][e, ]d, q )
   SO( [filt, ][e, ]d, q )
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 48.2.6) that have determinant one,
in the category given by the filter \( \text{filt} \). (The index of \( \text{SO}(e, d, q) \) in \( \text{GO}(e, d, q) \) is 2 if \( q \) is odd, and 1 if \( q \) is even.)

If \( \text{filt} \) is not given it defaults to \text{IsMatrixGroup}, and the returned group is the special orthogonal group itself.

\[
\text{gap> GeneralOrthogonalGroup( 3, 7 );}
\]
\[
\text{GO}(0,3,7)
\]
\[
\text{gap> GeneralOrthogonalGroup( -1, 4, 3 );}
\]
\[
\text{GO}(-1,4,3)
\]
\[
\text{gap> SpecialOrthogonalGroup( 1, 4, 4 );}
\]
\[
\text{GO}(+1,4,4)
\]

\( \rightarrow \) \text{ProjectiveGeneralLinearGroup}( [\text{filt}, ]d, q )
\( \rightarrow \) \text{PGL}( [\text{filt}, ]d, q )

constructs a group isomorphic to the projective general linear group \( \text{PGL}(d, q) \) of those \( d \times d \) matrices over the field with \( q \) elements, modulo the centre, in the category given by the filter \( \text{filt} \).

If \( \text{filt} \) is not given it defaults to \text{IsPermGroup}, and the returned group is the action on lines of the underlying vector space.

\( \rightarrow \) \text{ProjectiveSpecialLinearGroup}( [\text{filt}, ]d, q )
\( \rightarrow \) \text{PSL}( [\text{filt}, ]d, q )

constructs a group isomorphic to the projective special linear group \( \text{PSL}(d, q) \) 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 \( \text{filt} \).

If \( \text{filt} \) is not given it defaults to \text{IsPermGroup}, and the returned group is the action on lines of the underlying vector space.

\( \rightarrow \) \text{ProjectiveGeneralUnitaryGroup}( [\text{filt}, ]d, q )
\( \rightarrow \) \text{PGU}( [\text{filt}, ]d, q )

constructs a group isomorphic to the projective general unitary group \( \text{PGU}(d, q) \) 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 \( \text{filt} \).

If \( \text{filt} \) is not given it defaults to \text{IsPermGroup}, and the returned group is the action on lines of the underlying vector space.

\( \rightarrow \) \text{ProjectiveSpecialUnitaryGroup}( [\text{filt}, ]d, q )
\( \rightarrow \) \text{PSU}( [\text{filt}, ]d, q )

constructs a group isomorphic to the projective special unitary group \( \text{PSU}(d, q) \) 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 \( \text{filt} \).

If \( \text{filt} \) is not given it defaults to \text{IsPermGroup}, and the returned group is the action on lines of the underlying vector space.

\( \rightarrow \) \text{ProjectiveSymplecticGroup}( [\text{filt}, ]d, q )
\( \rightarrow \) \text{PSP}( [\text{filt}, ]d, q )
\( \rightarrow \) \text{PSp}( [\text{filt}, ]d, q )

constructs a group isomorphic to the projective symplectic group \( \text{PSp}(d,q) \) 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 \( \text{filt} \).

If \( \text{filt} \) is not given it defaults to \text{IsPermGroup}, and the returned group is the action on lines of the underlying vector space.
48.3 Conjugacy Classes in Classical Groups

For general and special linear groups (see 48.2.1 and 48.2.2) 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.

```gap
g := SL(4,9);
SL(4,9)
gap> NrConjugacyClasses(g);
861
gap> cl := ConjugacyClasses(g);

gap> Length(cl);
861
```

The first of these functions compute for given \( n \in \mathbb{N} \) and prime power \( q \) the number of conjugacy classes in the classical groups \( GL(n, q) \), \( GU(n, q) \), \( SL(n, q) \), \( SU(n, q) \), \( PGL(n, q) \), \( PGU(n, q) \), \( PSL(n, q) \), respectively. (See also 37.9.2 and Section 48.2.)

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 \( \text{NrConjugacyClassesSLIsogeneous} \) 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 \( \text{NrConjugacyClassesSUIsogeneous} \) is the analogous one for the corresponding unitary groups. The formulae for the number of conjugacy classes are taken from [Mac81].

```gap
NrConjugacyClassesGL( n, q )
NrConjugacyClassesGU( n, q )
NrConjugacyClassesSL( n, q )
NrConjugacyClassesSU( n, q )
NrConjugacyClassesPGL( n, q )
NrConjugacyClassesPGU( n, q )
NrConjugacyClassesPSL( n, q )
NrConjugacyClassesPSU( n, q )
NrConjugacyClassesSLIsogeneous( n, q, f )
NrConjugacyClassesSUIsogeneous( n, q, f )
```

48.4 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 \texttt{Cons}, so for example \texttt{CyclicGroup} calls the constructor

```gap
CyclicGroupCons( cat, n )
```

The first argument \texttt{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 2.2.1 in “Programming in GAP” for the meaning of the arguments of \texttt{InstallMethod}).
Section 5. Selection Functions

InstallMethod( CyclicGroupCons,
"regular perm group",
true,
[ IsPermGroup and IsRegularProp and IsFinite, IsInt and IsPosRat ], 0,
function( filter, n )
...
end );

48.5 Selection Functions

For a number of the group libraries two selection functions are provided. Each AllLibraryGroups 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, AllIrreducibleSolvableGroups, AllTransitiveGroups, and AllPrimitiveGroups. Corresponding to each of these there is a OneLibraryGroup function (see 48.5.2) 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 the first such function must be Size, and, unlike the other library selection functions, it supports an alternative syntax where Size is omitted (see 48.7.2). Also, see 48.10.3, for details pertaining to AllIrreducibleSolvableGroups.

For an example, let us consider the selection function for the library of transitive groups (also see 48.6). The command,

\[
gap> \text{AllTransitiveGroups(NrMovedPoints, [10..15], Size, [1..100], IsAbelian, false );}
\]

returns a list of all transitive groups with degree between 10 and 15 and size less than 100 that are not abelian.

Thus the 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.)

\[
\text{AllTransitiveGroups} := \text{function}( \text{fun1, val1, fun2, val2 } )
\]

local groups, g, i;

\[
g := \text{TransitiveGroupsList}[i];
\]

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;

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.

2► OneLibraryGroup( fun1, val1, ... )

For each AllLibraryGroups function (see 48.5.1) there is a corresponding function OneLibraryGroup on exactly the same arguments, i.e. there are OneSmallGroup, OneIrreducibleSolvableGroup, OneTransitiveGroup, and OnePrimitiveGroup. Each function simply returns the first group in the library (in the stored order) that has the prescribed properties, instead of all such groups. It returns fail if no such group exists in the library.

48.6 Transitive Permutation Groups

The transitive groups library currently contains representatives for all transitive permutation groups of degree at most 30. Two permutation 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.

1► TransitiveGroup( deg, nr )

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.

2► NrTransitiveGroups( deg )

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.

The selection functions (see 48.5) for the transitive groups library are AllTransitiveGroups and OneTransitiveGroup. They obtain the following properties from the database without having to compute them anew:

NrMovedPoints, Size, Transitivity, and IsPrimitive.

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 are due to [Hulpke96,HulpkeTG]. (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 and AlternatingGroup) 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.

```gap> TransitiveGroup(10,22);
S(5)[x]2
```

```gap> l:=AllTransitiveGroups(NrMovedPoints,12,Size,1440,IsSolvable,false);
```

```gap> List(l,IsSolvable);
[ false, false ]
```

3► TransitiveIdentification( G )

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 \texttt{TransitiveIdentification} is undefined.

\[
gap> \text{TransitiveIdentification}(	ext{Group}((1,2),(1,2,3))); \\
2
\]

### 48.7 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 order 5\(^n\) and 7\(^4\) (92 groups);
- those of order \(q^n \cdot p\) where \(q^n\) divides 2\(^8\), 3\(^6\), 5\(^5\) or 7\(^4\) and \(p\) is an arbitrary prime with \(p \neq q\);
- those whose order factorises into at most 3 primes.

The first two items in this list cover an explicit range of orders; the last two 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 512 and 1536.

The library is organised in 8 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 \(1 \leq m \leq n \leq 8\). This might be useful to save disk space. There is an extensive \texttt{README} file for the Small Groups library available in the \texttt{small} directory of the \texttt{GAP} distribution containing detailed information on the layers. In summary, the layers are:

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 groups of order 5\(^5\) and 7\(^4\) and the remaining groups of order \(q^n \cdot p\) where \(q^n\) divides 3\(^6\), 5\(^5\) or 7\(^4\) and \(p\) is a prime with \(p \neq q\).
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.

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 26 MB (storing over 400 million groups in about 200 megabits). The group identification occupies about 42 MB of which 18 MB is used for the groups in layer (6). More information on the Small Groups library can be found on


This group library has been computed 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 [BEOon]. For further detailed
information on the construction of this library we refer to [New77], [O'B90], [O'B91], [BE99a], [BE99b] and [BE01].

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.

1 ▶ SmallGroup( order, i )
   ▶ SmallGroup( [order, i] )  

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.

2 ▶ AllSmallGroups( arg )

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 and the first value an order or a range of orders. 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, IsNilpotentGroup, IsSupersolvableGroup, IsSolvableGroup, RankPGroup, PClassPGroup, LGLength, 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 \( n > 2 \).

3 ▶ OneSmallGroup( arg )

returns one group with certain properties as specified by arg. The permitted arguments are those supported by AllSmallGroups.

4 ▶ NumberSmallGroups( order )

returns the number of groups of order order.

5 ▶ IdSmallGroup( G )
   ▶ IdGroup( G )

returns the library number of G; that is, the function returns a pair [order, i] where G is isomorphic to SmallGroup( order, i ).

6 ▶ IdsOfAllSmallGroups( arg )

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.

7 ▶ IdGap3SolvableGroup( G )
   ▶ Gap3CatalogueIdGroup( G )

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.

8 ▶ SmallGroupsInformation( order )

prints information on the groups of the specified order.

9 ▶ UnloadSmallGroupsData( )

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.

    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

    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>

    gap> SmallGroupsInformation( 32 );
Chapter 48. Group Libraries

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.

48.8 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 $n = 61440, 122880, 172032, 245760, 344064, 491520, 688128, 983040$, 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 1096 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.

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.

```plaintext
1 mixer SizesPerfectGroups( ) 

2 mixer PerfectGroup( [filt, ]size [, n] )

   mixer PerfectGroup( [filt, ]sizenumberpair )
```

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\ and \IsSubgroupFpGroup\.

In the latter case, the generators and relators used coincide with those given in [HP89].
Section 8. Finite Perfect Groups

\begin{verbatim}
gap> G := PerfectGroup(IsPermGroup,6048,1);
U3(3)

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.

\begin{verbatim}
gap> G:=PerfectGroup(IsPermGroup,129024,2);
L2(8) N ( 2^-6 E 2^-1 A ) C 2^-1
\end{verbatim}

\begin{verbatim}
gap> NrMovedPoints(G);
14336
\end{verbatim}
\end{verbatim}

This attribute is set for all groups obtained from the perfect groups library and has the value \([\text{size}, \text{nr}]\) if the group is obtained with these parameters from the library.

\begin{verbatim}
\end{verbatim}

returns the number of non-isomorphic perfect groups of size \(\text{size}\) for each positive integer \(\text{size}\) 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 \text{fail}.

\begin{verbatim}
\end{verbatim}

returns the number of perfect groups of size \(\text{size}\) 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.)

\begin{verbatim}
\end{verbatim}

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 \(\text{factor1}, \text{factor2}, \ldots\) 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.

- \(A(n)\) or \(A(n)\) for the alternating groups \(A_n\), \(5 \leq n \leq 9\), for example \(A5\) or \(A6\).
- \(L(n, q)\) 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
- \(U(n, q)\) 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 \(Sp4(4)\) for the symplectic group \(Sp(4, 4)\),
- \(Sz(8)\) for the Suzuki group \(Sz(8)\),
- \(M(n)\) 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 \textit{DisplayCompositionSeries} command.
of GAP. However, as the names are compared as text strings, you are restricted to the above choice. Even expressions like $L_2(2^5)$ are not accepted.

As the use of the term $PSU(n, q)$ is not unique in the literature, we mention that in this library it denotes the factor group of $SU(n, q)$ by its centre, where $SU(n, q)$ 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.

$\texttt{DisplayInformationPerfectGroups( size )}$

$\texttt{DisplayInformationPerfectGroups( size, n )}$

$\texttt{DisplayInformationPerfectGroups( [ size, n ] )}$

$\texttt{DisplayInformationPerfectGroups}$ 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. 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.

```gap
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)
```
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 DisplayInformationPerfectGroups function displays only the available degree. The message

\textit{orbit size = 8}

in the above example means that the available permutation representation is transitive and of degree 8, whereas the message

\textit{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 $F_p Q$-modules $M$ with $|Q| \cdot |M| \leq 10^6$, where $F_p$ is the field of order $p$, are assigned symbols. These will either be simply $p^r$, 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 by $p^r, p^r', \ldots$. 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 $2^1 \cdot A_5$.)

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 $F_p Q$-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 $p$th 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 $F_p Q$-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) $A_4(2^4E2^1E2^4)A$ means that the pairs $2^4E2^1$ and $2^1E2^4$ are both elementary abelian of exponent 4.

(ii) $A_4(2^4E2^1A)C2^1$ 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) $L_3(2)((2^1E) \times (N2^3E2^3)A)C2^3$ 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 $2^3A2^3$ 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 the DisplayInformationPerfectGroups function. 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.

### 48.9 Primitive Permutation Groups

GAP contains a library of primitive permutation groups which includes the following permutation groups up to permutation isomorphism (i.e., up to conjugacy in the corresponding symmetric group)

- the non-affine primitive permutation groups of degree $\leq 999$, described in [DM88], with generators calculated in [The97],
- all primitive permutation groups of degree $< 256$, in particular,
  - the primitive permutation groups up to degree 50, calculated by C. Sims,
  - 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].

Note that the affine primitive permutation groups of degrees 256–999 are not included.

For degree up to 50, the names used are as given by Buekenhout and Leemans [BL96].
The names for the groups of higher degree are as chosen by [The97] and reflect the cohort structure in [DM88]. They do not conform to the usual naming for extensions. That is $13_3,4,3$ is the third (in some arbitrary ordering) group in a cohort for socle $13_3,4$, but the socle factor is not necessarily of order 3.

The work in [The97] is known to have omissions. Because of this we do not guarantee completeness of the lists beyond degree 50, though we have corrected errors as far as we know of them. When preparing the library it also has been ensured that the groups in it are all primitive and not conjugate.

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 $\text{PrimitiveGroup}(n, nr)$ 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.)

1. $\text{PrimitiveGroup}(\text{deg}, \text{nr})$  
returns the primitive permutation group of degree $\text{deg}$ with number $\text{nr}$ 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 $\text{SimsNo}(48.9.5)$.

2. $\text{NrPrimitiveGroups}(\text{deg})$  
returns the number of primitive permutation groups of degree $\text{deg}$ in the library.

   gap> $\text{NrPrimitiveGroups}(25)$;
   28
   gap> $\text{PrimitiveGroup}(25,19)$;
   5^2:((Q_8:3)'4
   gap> $\text{PrimitiveGroup}(25,20)$;
   $\text{ASL}(2,5)$
   gap> $\text{PrimitiveGroup}(25,22)$;
   $\text{AGL}(2,5)$
   gap> $\text{PrimitiveGroup}(25,23)$;
   (A_5xA_5):2

The selection functions (see 48.5) for the primitive groups library are $\text{AllPrimitiveGroups}$ and $\text{OnePrimitiveGroup}$. They obtain the following properties from the database without having to compute them anew: $\text{NrMovedPoints}$, $\text{Size}$, $\text{Transitivity}$, $\text{ONanScottType}$, $\text{IsSimpleGroup}$, $\text{IsSolvableGroup}$, and $\text{SocleTypePrimitiveGroup}$.

(Note, that for groups of degree up to 999, O'Nan-Scott types 4a, 4b and 5 cannot occur.)

3. $\text{PrimitiveGroupsIterator}(\text{attr1}, \text{val1}, \text{attr2}, \text{val2}, \ldots)$  
returns an iterator through $\text{AllPrimitiveGroups}(\text{attr1}, \text{val1}, \text{attr2}, \text{val2}, \ldots)$ without creating all these groups at the same time.
Chapter 48. Group Libraries

4 ▶ PrimitiveIdentification( $G$ )

For a primitive permutation group for which an $S_n$-conjugate exists in the library of primitive permutation groups (see 48.9), 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.

```
gap> PrimitiveIdentification(Group([1,2],[1,2,3]));
2
```

5 ▶ SimsNo( $G$ )

If $G$ is a primitive group obtained by PrimitiveGroup (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).

```
gap> g:=PrimitiveGroup(25,2);
5^2:S_3
gap> SimsNo(g);
3
```

6 ▶ COHORTS_PRIMITIVE_GROUPS

In [DM88] the primitive groups are sorted in “cohorts” according to their socle. For each degree, the variable COHORTS_PRIMITIVE_GROUPS 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, section 41.4.2), the second entry listing the index numbers of the groups in this degree.

For example in degree 49, we have four cohorts with socles $\mathbb{F}_2^2$, $L_2(7)^2$, $A_7^2$ 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 $(A_7 \times A_7)$.

```
gap> COHORTS_PRIMITIVE_GROUPS[49];
[ [ rec( series := "Z", width := 2, parameter := 7 ),
  [ 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( series := "L", width := 2, parameter := [ 2, 7 ] ), [ 34 ] ],
```

48.10 Irreducible Solvable Matrix Groups

1 ▶ IrreducibleSolvableGroupMS( $n$, $p$, $i$ )

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 IrreducibleSolvableMatrixGroup will return fail for all but one of the duplicates.

For values of $n$, $p$, and $i$ admissible to IrreducibleSolvableGroup, IrreducibleSolvableMatrixGroup returns a representative of the same conjugacy class of subgroups of $GL(n,p)$ as IrreducibleSolvableGroup.

Note that it currently adds two more groups (missing from the original list by Mark Short) for $n = 2$, $p = 13$. 
2 ▶ NumberIrreducibleSolvableGroups( n, p )

This function returns the number of conjugacy classes of irreducible solvable subgroup of $GL(n,p)$.

3 ▶ AllIrreducibleSolvableGroups( func1, val1, func2, val2, ... )

This function returns a list of conjugacy class representatives $G$ of matrix groups over a prime field such that $func_i(G) = val_i$ or $func_i(G) \in val_i$ for all $i$. The following possibilities for $func_i$ are particularly efficient, because the values can be read off the information in the data base: DegreeOfMatrixGroup (or Dimension or DimensionOfMatrixGroup) for the linear degree, Characteristic for the field characteristic, Size, IsPrimitiveMatrixGroup (or IsLinearlyPrimitive), and MinimalBlockDimension.

4 ▶ OneIrreducibleSolvableGroup( func1, val1, func2, val2, ... )

This function returns one solvable subgroup $G$ of a matrix group over a prime field such that $func_i(G) = val_i$ or $func_i(G) \in val_i$ for all $i$. The following possibilities for $func_i$ are particularly efficient, because the values can be read off the information in the data base: DegreeOfMatrixGroup (or Dimension or DimensionOfMatrixGroup) for the linear degree, Characteristic for the field characteristic, Size, IsPrimitiveMatrixGroup (or IsLinearlyPrimitive), and MinimalBlockDimension.

5 ▶ PrimitiveIndexIrreducibleSolvableGroup

This variable provides a way to get from irreducible solvable groups to primitive groups and vice versa. For the group $G = \text{IrreducibleSolvableGroup}(n, p, k)$ and $d = p^n$, the entry $\text{PrimitiveIndexIrreducibleSolvableGroup}[d][i]$ gives the index number of the semidirect product $p^n : G$ in the library of primitive groups.

Searching for an index Position in this list gives the translation in the other direction.

6 ▶ IrreducibleSolvableGroup( n, p, i )

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 (see 48.10.1). 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 returns a representative of the same conjugacy class of subgroups of $GL(n,p)$ as IrreducibleSolvableGroup did before.

### 48.11 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
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 $G_1 \cong W(F_4)$, $G_2 \cong D_{12} \wr S_2$, $G_3 \cong G_4 \cong C_2 \times S_5$, $G_5 \cong W(B_4)$, and $G_6 \cong D_{12} \wr D_{12} : C_2$. 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 $G_4$ 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 $G = \text{Aut}(F, L) = \{g \in GL_n(\mathbb{Z}) \mid Lg = L \text{ and } gFg^{\text{tr}} = F\}$ 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 $vFv^{\text{tr}}$ 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, \textbf{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, \textbf{ImfInvariants} and \textbf{DisplayImfInvariants}, 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 a more 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.

1. `ImfNumberQQClasses(dim)`
2. `ImfNumberQClasses(dim)`
3. `ImfNumberZClasses(dim, q)`

- **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.
- **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 $GL_{dim}(\mathbb{Q})$. 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.

2. `DisplayImfInvariants(dim, q)`
3. `DisplayImfInvariants(dim, q, z)`

- **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 $dim\cdot q\cdot z$, if $dim$ is at most 11 or a prime at most 23, or its $\mathbb{Q}$-class number in the form $dim\cdot q$, 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` or `IsomorphismPermGroupImfGroup` 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.
Chapter 48. Group Libraries

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 \texttt{DisplayImfInvariants} uses a kind of shorthand to display the elementary divisors. E. g., the expression $1 \cdot 3^3 \cdot 9^3 \cdot 27$ in the preceding example stands for the elementary divisors $1, 3, 3, 9, 9, 27$. (See also the next example which shows that the function \texttt{ImfInvariants} provides the elementary divisors in form of an ordinary GAP list.)

In the description of the isomorphism types the following notations are used:

$A \times B$ denotes a direct product of a group $A$ by a group $B$,

$A \text{ subd } B$ denotes a subdirect product of $A$ by $B$,

$A \odot Y B$ 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 $B$,

$A \cdot B$ 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 $C_n$, $D_n$, and $Q_n$, respectively,

- the alternating groups $A_n$ and symmetric groups $S_n$ of degree $n$, denoted by $A_n$ and $S_n$, 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 $O_q^n(2)$ and $PO_q^n(2)$, 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 $G_2(3)$,

- the twisted Chevalley group $^3D_4(2)$, denoted by $3D_4(2)$,

- the Suzuki group $S_z(8)$, 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 $Co_1$, $Co_2$, $Co_3$, $HS$, $J_2$, $M_{11}$, $M_{22}$, $M_{23}$, $M_{24}$, and $Mc$, denoted by $Co1$, $Co2$, $Co3$, $HS$, $J2$, $M11$, $M22$, $M23$, $M24$, 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.

```plaintext
ImfInvariants( dim, q )
ImfInvariants( dim, q, z )
```

ImfInvariants returns a record which provides some $Z$-class invariants of the groups in the $z^{th}$ $Z$-class in the $q^{th}$ $Q$-class of i.m.f. integral matrix groups of dimension $dim$. A value $z\neq 1$ must not be specified if the $Z$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \not\in \{13, 17, 19, 23\}$. The default value of $z$ is 1. This value of $z$ will be accepted even if the $Z$-classes are not known. Then it specifies the only representative group which is available for the $q^{th}$ $Q$-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, see 24.8.1),

- **minimalNorm**
  
  the norm of the associated short vectors,

- **sizesOrbitsShortVectors**
  
  the sizes of the orbits of short vectors under $F$,

- **maximalQClass**
  
  the $Q$-class number of an i.m.f. group in $GL_n(Q)$ that contains $G$ as a subgroup (only in case that not $G$ itself is an i.m.f. subgroup of $GL_n(Q)$).
Chapter 48. Group Libraries

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 \(Z\)-class invariants, but also \(Q\)-class invariants.

Note further that, though the isomorphism type is a \(Q\)-class invariant, you will sometimes get different descriptions for different \(Z\)-classes of the same \(Q\)-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.

```gap
gap> ImfInvariants( 8, 15, 1 );
rec( size := 2592, isSolvable := true, isomorphismType := "C2 x (S3 wr S3)",
   elementaryDivisors := [ 1, 3, 3, 3, 9, 9, 9, 27 ], minimalNorm := 8,
   sizesOrbitsShortVectors := [ 54 ], maximalQClass := 5 )
gap> ImfInvariants( 24, 1 ).size;
1040939685273332453861621760000

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 PGL(2,23)
C2 x PGL(2,23)

4\>
```

\textbf{ImfMatrixGroup} is the essential extraction function of this library (note that its name has been changed from \textbf{ImfMatGroup} in GAP 3 to \textbf{ImfMatrixGroup} in GAP 4). It returns a representative group, \(G\) say, of the \(z\)th \(Z\)-class in the \(q\)th \(Q\)-class of i.m.f. integral matrix groups of dimension \(dim\). A value \(z \neq 1\) must not be specified if the \(Z\)-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 \(Z\)-classes are not known. Then it specifies the only representative group which is available for the \(q\)th \(Q\)-class. The greatest legal value of \(dim\) is 31.

```gap
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 ],
```

\textbf{ImfMatrixGroup}
The attributes Size 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 \texttt{ImfInvariants} described above (see 48.11.3), 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

- \texttt{form}
  the associated Gram matrix \( F \), and
- \texttt{repsOrbitsShortVectors}
  representatives of the orbits of short vectors under \( F \).

The last one of these components will be required by the function \texttt{IsomorphismPermGroup} below.

Example:

```gap
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 \textsc{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 \texttt{IsomorphismPermGroup} or \texttt{IsomorphismPermGroupImfGroup} to get an isomorphism to such a permutation group (note that these \textsc{GAP} 4 functions have replaced the \textsc{GAP} 3 functions \texttt{PermGroup} and \texttt{PermGroupImfGroup}).

\[ \texttt{IsomorphismPermGroup( G )} \]

returns an isomorphism, \( \varphi \) 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 \( \varphi \) 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 \( \varphi^{-1} \) from \( P \) to \( G \) is constructed by determining a \( \mathbb{Q} \)-basis \( B \subset S \) of \( \mathbb{Q}^{\dim} \) in \( S \) and, in addition, the associated base change matrix \( M \) which transforms \( B \) into the standard base of \( \mathbb{Z}^{\dim} \). This allows a simple computation of \( \varphi^{-1} \) for 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 and PreImage (see 31.3.6 and 31.4.6) 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 $G$.

```gap
# Perform the computations in an isomorphic permutation group.
phi := IsomorphismPermGroup( G );;
P := Image( phi );
Group([ (1,7,6)(2,9)(4,5,10), (2,3,4,5)(6,9,8,7) ])
D := DerivedSubgroup( P );
Group([ (1,2,10,9)(3,8)(4,5)(6,7), (1,3,10,8)(2,5)(4,7)(6,9),
   (1,2,8,7,5)(3,4,6,10,9) ])
Size( D );
960
IsPerfectGroup( D );
true
# We have found the solvable residuum of P,
# now move the results back to the matrix group G.
R := PreImage( phi, D );
<matrix group of size 960 with 3 generators>
for m in GeneratorsOfGroup( R ) do PrintArray( m ); od;
```

6. IsomorphismPermGroupImfGroup($G$, $n$)

IsomorphismPermGroupImfGroup returns an isomorphism, $\varphi$ 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 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).

```gap
ImfInvariants( 12, 9 ).sizesOrbitsShortVectors;
[ 120, 300 ]
G := ImfMatrixGroup( 12, 9 );
ImfMatrixGroup(12,9)
phi1 := IsomorphismPermGroupImfGroup( G, 1 );;
P1 := Image( phi1 );
```
Section 11. Irreducible Maximal Finite Integral Matrix Groups

<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
This chapter describes functions for creating semigroups and determining information about them.

1. `IsSemigroup( D )` returns `true` if the object `D` is a semigroup. A semigroup is a magma (see 33) with associative multiplication.

2. `Semigroup( gen1, gen2 ... )`  
   `Semigroup( gens )`

   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` (see 33.2.1) and `IsAssociative` (see 33.4.7) if you want to check whether a magma is in fact a semigroup.

   ```gap
   gap> a:= Transformation([2, 3, 4, 1]);
   Transformation( [ 2, 3, 4, 1 ] )
   gap> b:= Transformation([2, 2, 3, 4]);
   Transformation( [ 2, 2, 3, 4 ] )
   gap> s:= Semigroup(a, b);
   <semigroup with 2 generators>
   ```

3. `Subsemigroup( S, gens )`  
   `SubsemigroupNC( S, gens )`

   are just synonyms of `Submagma` and `SubmagmaNC`, respectively (see 33.2.7).

   ```gap
   gap> a:= GeneratorsOfSemigroup(s)[1];
   Transformation( [ 2, 3, 4, 1 ] )
   gap> t:= Subsemigroup(s, [a]);
   <semigroup with 1 generator>
   ```

4. `SemigroupByGenerators( gens )`  

   is the underlying operation of `Semigroup` (see 49).

5. `AsSemigroup( C )`

   If `C` is a collection whose elements form a semigroup (see 49) then `AsSemigroup` returns this semigroup. Otherwise `fail` is returned.

6. `AsSubsemigroup( D, C )`

   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.
GeneratorsOfSemigroup( S )

Semigroup generators of a semigroup $D$ are the same as magma generators (see 33.4.1).

gap> GeneratorsOfSemigroup(a);
gap> GeneratorsOfSemigroup(t);
[ Transformation( [ 2, 3, 4, 1 ] ) ]

FreeSemigroup( [wfilt], rank )

Called in the first form, FreeSemigroup returns a free semigroup on rank generators. Called in the second form, FreeSemigroup returns a free semigroup on as many generators as arguments, printed as name1, name2 etc. Called in the fourth 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].

If the extra argument wfilt is given, it must be either IsSyllableWordsFamily or IsLetterWordsFamily or IsWLetterWordsFamily or IsBLetterWordsFamily. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

SemigroupByMultiplicationTable( A )

returns the semigroup whose multiplication is defined by the square matrix $A$ (see 33.3.1) if such a semigroup exists. Otherwise fail is returned.

The following functions determine information about semigroups:

IsRegularSemigroup( S )

returns true if $S$ is regular—i.e. if every D class of $S$ is regular.

IsRegularSemigroupElement( S, x )

returns true if $x$ has a general inverse in $S$—i.e. there is an element $y \in S$ such that $xyx = x$ and $yxy = y$.

IsSimpleSemigroup( S )

is true if and only if the semigroup has no proper ideals.

IsZeroSimpleSemigroup( S )

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.
Chapter 49. Semigroups

49.1 Making transformation semigroups

Cayley’s Theorem gives special status to semigroups of transformations, and accordingly there are special functions to deal with them, and to create them from other finite semigroups.

- **IsTransformationSemigroup** (obj)
- **IsTransformationMonoid** (obj)

A transformation semigroup (resp. monoid) is a subsemigroup (resp. submonoid) of the full transformation monoid. Note that for a transformation semigroup to be a transformation monoid we necessarily require the identity transformation to be an element.

- **DegreeOfTransformationSemigroup** (S)

The number of points the semigroup acts on.

- **IsomorphismTransformationSemigroup** (S)
- **HomomorphismTransformationSemigroup** (S, r)

IsomorphismTransformationSemigroup is a generic attribute which is a transformation semigroup isomorphic to S (if such can be computed). In the case of an fp- semigroup, a todd-coxeter will be attempted. For a semigroup of endomorphisms of a finite domain of n elements, it will be to a semigroup of transformations of \{1, \ldots, n\}. Otherwise, it will be the right regular representation on S or \(S^1\) if S has no MultiplicativeNeutralElement.

HomomorphismTransformationSemigroup finds a representation of S as transformations of the set of equivalence classes of the right congruence r.

49.2 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 (33.2.1).

- **SemigroupIdealByGenerators** (S, gens)

S is a semigroup, gens is a list of elements of S. Returns the two-sided ideal of S generated by gens.

- **ReesCongruenceOfSemigroupIdeal** (I)

A two sided ideal I of a semigroup S defines a congruence on S given by \(\Delta \cup I \times I\).

- **IsLeftSemigroupIdeal** (I)
- **IsRightSemigroupIdeal** (I)
- **IsSemigroupIdeal** (I)

Categories of semigroup ideals.

- **IsRightSemigroupIdealEnumerator** (obj)
- **IsLeftSemigroupIdealEnumerator** (obj)
- **IsSemigroupIdealEnumerator** (obj)

Categories for generic semigroup ideal enumerators. Note: this does not imply that the UnderlyingDomain of the enumerator is a semigroup ideal, only that the object being enumerated is being enumerated as a semigroup ideal. In particular, semigroups are generally enumerated as a right or left ideal of themselves.
Section 5. Green’s Relations

49.3 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 (33.2.1).

1 ▶ IsSemigroupCongruence( c )

a magma congruence c on a semigroup.

2 ▶ IsReesCongruence( c )

returns true precisely when the congruence c has at most one nonsingleton congruence class.

49.4 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.

1 ▶ IsQuotientSemigroup( S )

is the category of semigroups constructed from another semigroup and a congruence on it

Elements of a quotient semigroup are equivalence classes of elements of QuotientSemigroupPreimage(S) under the congruence QuotientSemigroupCongruence(S).

It is probably most useful for calculating the elements of the equivalence classes by using Elements or by looking at the images of elements of the QuotientSemigroupPreimage(S) under QuotientSemigroupHomomorphism(S):QuotientSemigroupPreimage(S) \rightarrow 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.

2 ▶ HomomorphismQuotientSemigroup( cong )

for a congruence cong and a semigroup S. Returns the homomorphism from S to the quotient of S by cong.

3 ▶ QuotientSemigroupPreimage( S )
▶ QuotientSemigroupCongruence( S )
▶ QuotientSemigroupHomomorphism( S )

for a quotient semigroup S.

49.5 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 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 \land L, and D = R \circ L.

Recall that relations R, L and J induce a partial order among the elements of the semigroup: 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^1yS^1.

We extend this preorder to a partial order on equivalence classes in the natural way.
The Green’s relations (which are equivalence relations) are attributes of the semigroup `semigroup`.

- `GreensRRelation(semigroup)`: A
- `GreensLRelation(semigroup)`: A
- `GreensJRelation(semigroup)`: A
- `GreensDRelation(semigroup)`: A
- `GreensHRelation(semigroup)`: A

return `true` if the equivalence relation `equiv-relation` is a Green’s relation of any type, or of `R`, `L`, `J`, `H`, `D` type, respectively, or `false` otherwise.

- `IsGreensClass(equiv-class)`: P
- `IsGreensRClass(equiv-class)`: P
- `IsGreensLClass(equiv-class)`: P
- `IsGreensJClass(equiv-class)`: P
- `IsGreensHClass(equiv-class)`: P
- `IsGreensDClass(equiv-class)`: P

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.

- `IsGreensLessThanOrEqual(C1, C2)`: O
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.

- `RClassOfHClass(H)`: A
- `LClassOfHClass(H)`: A
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.

- `EggBoxOfDClass(Dclass)`: A
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.

- `DisplayEggBoxOfDClass(Dclass)`: F
displays a “picture” of the D class `Dclass`, as an array of 1s and 0s. A 1 represents a group `H` class.

- `GreensRClassOfElement(S, a)`: O
- `GreensLClassOfElement(S, a)`: O
- `GreensDClassOfElement(S, a)`: O
- `GreensJClassOfElement(S, a)`: O
- `GreensHClassOfElement(S, a)`: O

Creates the `X` class of the element `a` in the semigroup `S` where `X` is one of `L`, `R`, `D`, `J` or `H`. 
Section 6. Rees Matrix Semigroups

9 ▶ GreensRClasses( semigroup )
   ▶ GreensLClasses( semigroup )
   ▶ GreensJClasses( semigroup )
   ▶ GreensDClasses( semigroup )
   ▶ GreensHClasses( semigroup )

return the R, L, J, H, or D Green’s classes, respectively for semigroup semigroup.

10 ▶ GroupHClassOfGreensDClass( Dclass )

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.

11 ▶ IsGroupHClass( Hclass )

returns true if the Greens H class Hclass is a group, which in turn is true if and only if Hclass^2 intersects Hclass.

12 ▶ IsRegularDClass( Dclass )

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 Greens 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).

49.6 Rees Matrix Semigroups

In this section we describe GAP functions for Rees matrix semigroups and Rees 0-matrix semigroups. The importance of this construction is 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

Recall that a Rees Matrix semigroup is constructed from a semigroup (the underlying semigroup), and a matrix. A Rees Matrix semigroup element is a triple (s, i, lambda) where s is an element of the underlying semigroup S and i, lambda are indices. This can be thought of as a matrix with zero everywhere except for an occurrence of s at row i and column lambda. The multiplication is defined by (s, i, lambda)*(t, j, mu) = (s*P^lambda, t, i, mu) where P is the defining matrix of the semigroup. In the case that the underlying semigroup has a zero we can make the ReesZeroMatrixSemigroup, wherein all elements (i, 0, lambda) are identified to zero. The zero in S is found automatically. If one cannot be found, an error is signalled.

1 ▶ ReesMatrixSemigroup( S, matrix )

for a semigroup S and matrix whose entries are in S. Returns the Rees Matrix semigroup with multiplication defined by matrix.

2 ▶ ReesZeroMatrixSemigroup( S, matrix )

for a semigroup S with zero, and matrix over S returns the Rees 0-Matrix semigroup such that all elements (i, 0, lambda) are identified to zero.

The zero in S is found automatically. If one cannot be found, an error is signalled.

3 ▶ IsReesMatrixSemigroup( T )

returns true if the object T is a (whole) Rees matrix semigroup.

4 ▶ IsReesZeroMatrixSemigroup( T )

returns true if the object T is a (whole) Rees 0-matrix semigroup.
for a Rees matrix semigroup \( R \), \( a \) in \( \text{UnderlyingSemigroup}(R) \), \( i \) and \( \lambda \) in the row (resp. column) ranges of \( R \), returns the element of \( R \) corresponding to the matrix with zero everywhere and \( a \) in row \( i \) and column \( \lambda \).

is the category of elements of a Rees (0-) matrix semigroup. Returns true if \( e \) is an element of a Rees Matrix semigroup.

each return the defining matrix of the Rees (0-) matrix semigroup.

For an element \( x \) of a Rees Matrix semigroup, of the form \((s, i, \lambda)\), the row index is \( i \), the column index is \( \lambda \) and the underlying element is \( s \). If we think of an element as a matrix then this corresponds to the row where the non-zero entry is, the column where the non-zero entry is and the entry at that position, respectively.

returns true if \( x \) is the zero of the Rees 0-matrix semigroup.

Given a regular \( D \) class of a finite semigroup, it can be viewed as a Rees matrix semigroup by identifying products which do not lie in the \( D \) class with zero, and this is what it is returned.

Formally, let \( I_1 \) be the ideal of all \( J \) classes less than or equal to \( D \), \( I_2 \) the ideal of all \( J \) classes strictly less than \( D \), and \( \rho \) the Rees congruence associated with \( I_2 \). Then \( I_1/\rho \) is zero-simple. Then \( \text{AssociatedReesMatrixSemigroupOfDClass}( D ) \) returns this zero-simple semigroup as a Rees matrix semigroup.

an isomorphism to a Rees matrix semigroup over a group (resp. zero group).
50

Monoids

This chapter describes functions for monoids. Currently there are only few of them. More general functions for magmas and semigroups can be found in Chapters 33 and 49.

1 ▶ IsMonoid( D )

A monoid is a magma-with-one (see 33) with associative multiplication.

2 ▶ Monoid( gen1, gen2 ... )
   ▶ Monoid( gens )
   ▶ Monoid( gens, id )

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 third form, Monoid returns the monoid generated by the elements in the homogeneous list gens, with identity id.

It is **not** checked whether the underlying multiplication is associative, use MagmaWithOne (see 33.2.2) and IsAssociative (see 33.4.7) if you want to check whether a magma-with-one is in fact a monoid.

3 ▶ Submonoid( M, gens )
   ▶ SubmonoidNC( M, gens )

are just synonyms of SubmagmaWithOne and SubmagmaWithOneNC, respectively (see 33.2.8).

4 ▶ MonoidByGenerators( gens )
   ▶ MonoidByGenerators( gens, one )

is the underlying operation of Monoid (see 50).

5 ▶ AsMonoid( C )

If C is a collection whose elements form a monoid (see 50) then AsMonoid returns this monoid. Otherwise fail is returned.

6 ▶ AsSubmonoid( D, C )

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.

7 ▶ GeneratorsOfMonoid( M )

Monoid generators of a monoid M are the same as magma-with-one generators (see 33.4.2).

8 ▶ TrivialSubmonoid( M )

is just a synonym for TrivialSubmagmaWithOne (see 33.4.14).
Chapter 50. Monoids

\begin{verbatim}
FreeMonoid( [wfilt, ]rank )  F
FreeMonoid( [wfilt, ]rank, name )  F
FreeMonoid( [wfilt, ]name1, name2, ... )  F
FreeMonoid( [wfilt, ]names )  F
FreeMonoid( [wfilt, ]infinity, name, init )  F
\end{verbatim}

Called in the first form, \texttt{FreeMonoid} returns a free monoid on \texttt{rank} generators. Called in the second form, \texttt{FreeMonoid} returns a free monoid on \texttt{rank} generators, printed as \texttt{name1, name2} etc., that is, each name is the concatenation of the string \texttt{name} and an integer from 1 to \texttt{range}. Called in the third form, \texttt{FreeMonoid} returns a free monoid on as many generators as arguments, printed as \texttt{name1, name2} etc. Called in the fourth form, \texttt{FreeMonoid} returns a free monoid on as many generators as the length of the list \texttt{names}, the \( i \)-th generator being printed as \texttt{names[i]}. Called in the fifth form, \texttt{FreeMonoid} returns a free monoid on infinitely many generators, where the first generators are printed by the names in the list \texttt{init}, and the other generators by \texttt{name} and an appended number.

If the extra argument \texttt{wfilt} is given, it must be either \texttt{IsSyllableWordsFamily} or \texttt{IsLetterWordsFamily} or \texttt{IsWLetterWordsFamily} or \texttt{IsBLetterWordsFamily}. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

\begin{verbatim}
MonoidByMultiplicationTable( A )  F
\end{verbatim}

returns the monoid whose multiplication is defined by the square matrix \( A \) (see 33.3.1) if such a monoid exists. Otherwise \texttt{fail} is returned.
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.

\[
\text{gap> } \text{f:=FreeSemigroup("a","b");;}
\text{gap> } \text{x:=GeneratorsOfSemigroup(f);;}
\text{gap> } \text{s:=f/[ [x[1]*x[2],x[2]*x[1]] ];}
\text{<fp semigroup on the generators [ a, b ]>}
\text{gap> } \text{GeneratorsOfSemigroup(s);}
[ a, b ]
\text{gap> } \text{RelationsOfFpSemigroup(s);}
[ [ a*b, b*a ] ]
\]

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.

\[
\text{gap> } \text{f:=FreeMonoid("a","b");;}
\text{gap> } \text{x:=GeneratorsOfMonoid(f);}
[ a, b ]
\text{gap> } \text{e:=Identity(f);}
<identity ...>
\text{gap> } \text{m:=f/[ [x[1]*x[2],e] ];}
\text{<fp monoid on the generators [ a, b ]>}
\text{gap> } \text{RelationsOfFpMonoid(m);}
[ [ a*b, <identity ...> ] ]
\]

Notice that for GAP a finitely presented monoid is not a finitely presented semigroup.

\[
\text{gap> } \text{IsFpSemigroup(m);}
\text{false}
\]

However, one can build a finitely presented semigroup isomorphic to that finitely presented monoid (see 51.1.3).

Also note that it 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.
Chapter 51. Finitely Presented Semigroups and Monoids

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 51.5.)

```
gap> a*b=a^5;  
false
gap> a^5*b^2*a=a^6*b^2;  
true
```

Note than elements of a finitely presented semigroup (or monoid) are not printed in a unique way:

```
gap> a^5*b^2*a;  
a^5*b^2*a

1 ▶ IsSubsemigroupFpSemigroup( t )
A 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).

2 ▶ IsSubmonoidFpMonoid( t )
A 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.
Section 1. Creating Finitely Presented Semigroups

3 ▶ IsFpSemigroup( s )

is a synonym for IsSubsemigroupFpSemigroup(s) and IsWholeFamily(s) (this is because a subsemigroup of a finitely presented semigroup is not necessarily finitely presented).

4 ▶ IsFpMonoid( m )

is a synonym for IsSubmonoidFpMonoid(m) and IsWholeFamily(m) (this is because a submonoid of a finitely presented monoid is not necessarily finitely presented).

5 ▶ IsElementOfFpSemigroup( elm )

returns true if elm is an element of a finitely presented semigroup.

6 ▶ IsElementOfFpMonoid( elm )

returns true if elm is an element of a finitely presented monoid.

7 ▶ FpGrpMonSmgOfFpGrpMonSmgElement( elm )

returns the finitely presented group, monoid or semigroup to which elm belongs

```
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
```

51.1 Creating Finitely Presented Semigroups

1 ▶ \( F/\text{rels} \)

creates a finitely presented semigroup given by the presentation \( \langle \text{gens} \mid \text{rels} \rangle \) where \( \text{gens} \) are the generators of the free semigroup \( F \), and the relations \( \text{rels} \) are entered as pairs of words in the generators of the free semigroup.

```
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 ]>
```

One may also call the following functions to construct finitely presented semigroups:

2 ▶ FactorFreeSemigroupByRelations( f, rels )

for a free semigroup \( f \) and \( \text{rels} \) 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 \( \text{rels} \).

```
gap> FactorFreeSemigroupByRelations(f,[[s[1]*s[2]*s[1],s[1]],[s[2]^4,s[1]]]);
<fp semigroup on the generators [ s1, s2, s3 ]>
```

Finally, if one has a finitely presented group or a finitely presented monoid, to find an isomorphic finitely presented semigroup use
3► IsomorphismFpSemigroup( s )

for a semigroup s returns an isomorphism from s to a finitely presented semigroup.

\[
gap> f := FreeGroup(2);;
gap> g := f/[f.1^4,f.2^5];
\]
\<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 ]>

51.2 Comparison of Elements of Finitely Presented Semigroups

1► a = b

Two elements 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 51.5).

51.3 Preimages in the Free Semigroup

1► FreeSemigroupOfFpSemigroup( s )

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)).

2► FreeGeneratorsOfFpSemigroup( s )

returns the underlying free generators corresponding to the generators of the finitely presented semigroup s.

3► RelationsOfFpSemigroup( s )

returns the relations of the finitely presented semigroup s as pairs of words in the free generators provided by FreeGeneratorsOfFpSemigroup(s).

\[
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 ] ]

Elements of a finitely presented semigroup are not words, but are represented using a word from the free semigroup as representative.
UnderlyingElement( elm )

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.

```gap
gap> w := GeneratorsOfSemigroup(s)[1] * GeneratorsOfSemigroup(s)[2];
a*b
gap> IsWord (w);
false
gap> ue := UnderlyingElement( w );
a*b
gap> IsWord( ue );
true
```

ElementOfFpSemigroup( fam, w )

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.

```gap
gap> fam := FamilyObj( GeneratorsOfSemigroup(s)[1] );;
gap> ge := ElementOfFpSemigroup( fam, a*b );
a*b
gap> ge in f;
false
gap> ge in s;
true
```

51.4 Finitely presented monoids

creates a finitely presented monoid given by the monoid presentation \( \langle \text{gens} \mid \text{rels} \rangle \) where gens are the generators of the free monoid \( F \), and the relations rel are entered as pairs of words in both the identity and the generators of the free monoid.

```gap
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 ]>
```

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.
Chapter 51. Finitely Presented Semigroups and Monoids

51.5 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.

\begin{verbatim}
ReducedConfluentRewritingSystem(S) A
ReducedConfluentRewritingSystem(S, ordering) A
\end{verbatim}

in the first form returns a reduced confluent rewriting system of the finitely presented semigroup or monoid \( S \) with respect to the length plus lexicographic ordering on words (also called the shortlex ordering; for the definition see for example Sims [Sim94]).

In the second form it returns a reduced confluent rewriting system of the finitely presented semigroup or monoid \( S \) with respect to the reduction ordering \( ordering \) (see 29).

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).

\begin{verbatim}
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 ] ]
\end{verbatim}

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 36. 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 36).

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).

\begin{verbatim}
KnuthBendixRewritingSystem(s, wordord)
KnuthBendixRewritingSystem(m, wordord)
\end{verbatim}

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 semigroup, it returns the Knuth-Bendix rewriting system of the finitely presented semigroup \( s \) using the reduction ordering \( wordord \).
Section 6. Todd-Coxeter Procedure

3 ▶ SemigroupOfRewritingSystem( rws )
returns the semigroup over which rws is a rewriting system

4 ▶ MonoidOfRewritingSystem( rws )
returns the monoid over which rws is a rewriting system

5 ▶ FreeSemigroupOfRewritingSystem( rws )
returns the free semigroup over which rws is a rewriting system

6 ▶ FreeMonoidOfRewritingSystem( rws )
returns the free monoid over which rws is a rewriting system

   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).

   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

51.6 Todd-Coxeter Procedure

This procedure gives a standard way of finding a transformation representation of a finitely presented semi-
group. Usually one does not explicitly call this procedure but uses IsomorphismTransformationSemigroup
or HomomorphismTransformationSemigroup (see 49.1.3).

1 ▶ CosetTableOfFpSemigroup( r )

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.
This chapter describes functions for transformations.

A transformation in GAP is an endomorphism of a set of integers of the form \( \{1, \ldots, n\} \). Transformations are taken to act on the right, which defines the composition \( i(\alpha \beta) = (i\alpha)\beta \) for \( i \) in \( \{1, \ldots, n\} \).

For a transformation \( \alpha \) on the set \( \{1, \ldots, n\} \), we define its degree to be \( n \), its image list to be the list, \([1\alpha, \ldots, n\alpha]\), its image to be the image list considered as a set, and its rank to be the size of the image. We also define the kernel of \( \alpha \) to be the equivalence relation containing the pair \( (i, j) \) if and only if \( i\alpha = j\alpha \).

Note that unlike permutations, we do not consider unspecified points to be fixed by a transformation. Therefore multiplication is only defined on two transformations of the same degree.

1. \( \text{IsTransformation}(\ obj\ ) \)
2. \( \text{IsTransformationCollection}(\ obj\ ) \)

We declare it as \( \text{IsMultiplicativeElementWithOne} \) since the identity automorphism of \( \{1, \ldots, n\} \) is a multiplicative two sided identity for any transformation on the same set.

3. \( \text{TransformationFamily}(\ n\ ) \)
4. \( \text{TransformationType}(\ n\ ) \)
5. \( \text{TransformationData}(\ n\ ) \)

For each \( n > 0 \) there is a single family and type of transformations on \( n \) points. To speed things up, we store these in a database of types. The three functions above a then access functions. If the \( n \)th entry isn’t yet created, they trigger creation as well.

For \( n > 0 \), element \( n \) of the type database is \([\text{TransformationFamily}(n), \text{TransformationType}(n)]\)

6. \( \text{Transformation}(\ images\ ) \)
7. \( \text{TransformationNC}(\ images\ ) \)

both return a transformation with the image list \( images \). The normal version checks that the all the elements of the given list lie within the range \( \{1, \ldots, n\} \) where \( n \) is the length of \( images \), but for speed purposes, a non-checking version is also supplied.

8. \( \text{IdentityTransformation}(\ n\ ) \)

return the identity transformation of degree \( n \)

9. \( \text{RandomTransformation}(\ n\ ) \)

returns a random transformation of degree \( n \)

10. \( \text{DegreeOfTransformation}(\ trans\ ) \)

returns the degree of \( trans \).
gap> t := Transformation([2, 3, 4, 2, 4]);
Transformation([ 2, 3, 4, 2, 4 ])
gap> DegreeOfTransformation(t);
5

7 ▶ ImageListOfTransformation( trans )
returns the image list of trans.

gap> ImageListOfTransformation(t);
[ 2, 3, 4, 2, 4 ]

8 ▶ ImageSetOfTransformation( trans )
returns the image of trans as a set.

gap> ImageSetOfTransformation(t);
[ 2, 3, 4 ]

9 ▶ RankOfTransformation( trans )
returns the rank of trans.

gap> RankOfTransformation(t);
3

10 ▶ KernelOfTransformation( trans )
Returns the kernel of trans as an equivalence relation (See 32.1).

gap> KernelOfTransformation(t);
<equivalence relation on <object> >
gap> EquivalenceRelationPartition(last);
[ [ 1, 4 ], [ 3, 5 ] ]

11 ▶ PreimagesOfTransformation( trans, i )
returns the subset of \{1, \ldots, n\} which maps to i under trans.

gap> PreimagesOfTransformation(t, 2);
[ 1, 4 ]

12 ▶ RestrictedTransformation( trans, alpha )
The transformation trans is restricted to only those points of alpha.

13 ▶ AsTransformation( O )
▶ AsTransformation( O, n )
▶ AsTransformationNC( O, n )
returns the object O as a transformation. Supported objects are permutations and binary relations on points. In the second form, the operation returns a transformation of degree n, signalling an error if such a representation is not possible. AsTransformationNC does not perform this check.
gap> AsTransformation((1, 3)(2, 4));
Transformation( [ 3, 4, 1, 2 ] )
gap> AsTransformation((1, 3)(2, 4), 10);
Transformation( [ 3, 4, 1, 2, 5, 6, 7, 8, 9, 10 ] )
gap> AsTransformation((1, 3)(2, 4), 3);
Error, Permutation moves points over the degree specified 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;

Given transformations \( tr1 \) and \( tr2 \) with equal kernel and image, we compute the permutation induced by \( (tr1)^{-1} \ast tr2 \) on the set of images of \( tr1 \). If the kernels and images are not equal, an error is signaled.

returns \( trans \) when considered as a binary relation.

returns the binary relation \( R \) when considered as a transformation. Only makes sense for injective binary relations over \([1..n]\). Returns an error if the relation is not over \([1..n]\), and \texttt{fail} if it is not injective.
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 $\ast$, (see 33), they are the basic algebraic structures. In many cases, the addition is commutative (see 53.3.1), the domain is called an additive magma then; every module (see 55), vector space (see 59), ring (see 54), or field (see 56) 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 53.1).

53.1 (Near-)Additive Magma Categories

1. IsNearAdditiveMagma( obj )
   C
   A near-additive magma in GAP is a domain $A$ with an associative but not necessarily commutative addition $+ : A \times A \to A$.

2. IsNearAdditiveMagmaWithZero( obj )
   C
   A near-additive magma-with-zero in GAP is a near-additive magma $A$ with an operation $0^\ast$ (or Zero) that yields the zero 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 53.3.5). This element $z$ can be computed with the operation Zero (see 30.10.3) as Zero( $A$ ) , and $z$ is also equal to Zero( $elm$ ) and to $0^\ast elm$ for each element $elm$ in $A$.
   Note that a near-additive magma containing a zero may not lie in the category IsNearAdditiveMagmaWithZero (see 30.6).

3. IsNearAdditiveGroup( obj )
   IsNearAdditiveMagmaWithInverses( obj )
   C
   A near-additive group in GAP is a near-additive magma-with-zero $A$ with an operation $-1^\ast : A \to A$ that maps each element $a$ of $A$ to its additive inverse $-1^\ast a$ (or AdditiveInverse( $a$ ), see 30.10.9).
   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 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.

4. IsAdditiveMagma( obj )
   C
   An additive magma in GAP is a domain $A$ with an associative and commutative addition $+ : A \times A \to A$, see 53.1.1 and 53.3.1.

5. IsAdditiveMagmaWithZero( obj )
   C
   An additive magma-with-zero in GAP is an additive magma $A$ with an operation $0^\ast$ (or Zero) that yields the zero of $A$.  

Chapter 53. Additive Magmas (preliminary)

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 53.3.5). This element \( z \) can be computed with the operation \( \text{Zero} \) (see 30.10.3) as \( \text{Zero}(A) \), and \( z \) is also equal to \( \text{Zero}(elm) \) and to \( 0*elm \) for each element \( elm \) in \( A \).

Note that an additive magma containing a zero may not lie in the category \( \text{IsAdditiveMagmaWithZero} \) (see 30.6).

6 ▶ \text{IsAdditiveGroup}( \ obj \ )
6 ▶ \text{IsAdditiveMagmaWithInverses}( \ obj \ )

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 \( \text{AdditiveInverse}(a) \), see 30.10.9).

The addition \(+\) of \( A \) is assumed to be commutative and associative, so an additive group is not more than an additive magma-with-inverses. \( \text{IsAdditiveMagmaWithInverses} \) is just a synonym for \( \text{IsAdditiveGroup} \), and can be used alternatively in all function names involving \( \text{AdditiveGroup} \).

Note that not every trivial additive magma is an additive magma-with-zero, but every trivial additive magma-with-zero is an additive group.

53.2 (Near-)Additive Magma Generation

1 ▶ \text{NearAdditiveMagma}( \ gens \ )
1 ▶ \text{NearAdditiveMagma}( \ Fam, \ gens \ )

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).

2 ▶ \text{NearAdditiveMagmaWithZero}( \ gens \ )
2 ▶ \text{NearAdditiveMagmaWithZero}( \ Fam, \ gens \ )

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 \( \text{Zero} \). The family \( Fam \) of \( A \) can be entered as first argument; this is obligatory if \( gens \) is empty (and hence \( A \) is trivial).

3 ▶ \text{NearAdditiveGroup}( \ gens \ )
3 ▶ \text{NearAdditiveGroup}( \ Fam, \ gens \ )

returns the (near-)additive group \( A \) that is generated by the elements in the list \( gens \), that is, the closure of \( gens \) under addition \(+\), \( \text{Zero} \), and \( \text{AdditiveInverse} \). The family \( Fam \) of \( A \) can be entered as first argument; this is obligatory if \( gens \) is empty (and hence \( A \) is trivial).

The underlying operations for which methods can be installed are the following.

4 ▶ \text{NearAdditiveMagmaByGenerators}( \ gens \ )
4 ▶ \text{NearAdditiveMagmaByGenerators}( \ Fam, \ gens \ )

5 ▶ \text{NearAdditiveMagmaWithZeroByGenerators}( \ gens \ )
5 ▶ \text{NearAdditiveMagmaWithZeroByGenerators}( \ Fam, \ gens \ )

6 ▶ \text{NearAdditiveGroupByGenerators}( \ gens \ )
6 ▶ \text{NearAdditiveGroupByGenerators}( \ Fam, \ gens \ )

Substructures of an additive magma can be formed as follows.
Section 3. Attributes and Properties for (Near-)Additive Magmas

SubnearAdditiveMagma( \( D, \text{gens} \) )

SubnearAdditiveMagmaNC( \( D, \text{gens} \) )

SubadditiveMagma returns the near-additive magma generated by the elements in the list \( \text{gens} \), with parent the domain \( D \). SubadditiveMagmaNC does the same, except that it is not checked whether the elements of \( \text{gens} \) lie in \( D \).

SubnearAdditiveMagmaWithZero( \( D, \text{gens} \) )

SubnearAdditiveMagmaWithZeroNC( \( D, \text{gens} \) )

SubadditiveMagmaWithZero returns the near-additive magma-with-zero generated by the elements in the list \( \text{gens} \), with parent the domain \( D \). SubadditiveMagmaWithZeroNC does the same, except that it is not checked whether the elements of \( \text{gens} \) lie in \( D \).

SubnearAdditiveGroup( \( D, \text{gens} \) )

SubnearAdditiveGroupNC( \( D, \text{gens} \) )

SubadditiveGroup returns the near-additive group generated by the elements in the list \( \text{gens} \), with parent the domain \( D \). SubadditiveGroupNC does the same, except that it is not checked whether the elements of \( \text{gens} \) lie in \( D \).

53.3 Attributes and Properties for (Near-)Additive Magmas

IsAdditivelyCommutative( \( A \) )

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, (see 33.4.9).

GeneratorsOfNearAdditiveMagma( \( A \) )

GeneratorsOfAdditiveMagma( \( A \) )

is a list \( \text{gens} \) of elements of the near-additive magma \( A \) that generates \( A \) as a near-additive magma, that is, the closure of \( \text{gens} \) under addition is \( A \).

GeneratorsOfNearAdditiveMagmaWithZero( \( A \) )

GeneratorsOfAdditiveMagmaWithZero( \( A \) )

is a list \( \text{gens} \) 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 \( \text{gens} \) under addition and \( \text{Zero} \) (see 30.10.3) is \( A \).

GeneratorsOfNearAdditiveGroup( \( A \) )

GeneratorsOfAdditiveGroup( \( A \) )

is a list \( \text{gens} \) of elements of the near-additive group \( A \) that generates \( A \) as a near-additive group, that is, the closure of \( \text{gens} \) under addition, taking the zero element, and taking additive inverses (see 30.10.9) is \( A \).

AdditiveNeutralElement( \( A \) )

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 \( \text{Zero}( A ) \) or as \( 0 \cdot \text{elm} \) for an element \( \text{elm} \) in \( A \), see 30.10.3.

TrivialSubnearAdditiveMagmaWithZero( \( A \) )

is the additive magma-with-zero that has the zero of the near-additive magma-with-zero \( A \) as only element.
53.4 Operations for (Near-)Additive Magmas

\[ \text{ClosureNearAdditiveGroup}( A, a ) \]  
\[ \text{ClosureNearAdditiveGroup}( A, B ) \]

returns the closure of the near-additive magma \( A \) with the element \( a \) or the near-additive magma \( B \), w.r.t. addition, taking the zero element, and taking additive inverses.
This chapter deals with domains that are additive groups closed under multiplication \(*\). Such a domain, if \(*\) and \(+\) are distributive, is called a ring in GAP. Each division ring, field (see 56), or algebra (see 60) is a ring, important examples are the integers (see 14) and matrix rings.

In the case of a ring-with-one, additional multiplicative structure is present, see 54.3.1.

Several functions for ring elements, such as IsPrime (54.5.7) and Factors (54.5.8), 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 54.1.3.

### 54.1 Generating Rings

1. **IsRing( \(R\))**
   
   A ring in GAP is an additive group (see 53.1.6) that is also a magma (see 33.1.1), such that addition \(\oplus\) and multiplication \(*\) are distributive.

   The multiplication need not be associative (see 33.4.7). For example, a Lie algebra (see 61) is regarded as a ring in GAP.

2. **Ring( \(r, s, \ldots\))**
   
   In the first form Ring returns the smallest ring that contains all the elements \(r, s, \ldots\) etc. 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 (see 54.1.3) in that it returns the smallest ring in which the elements lie, while DefaultRing may return a larger ring if that makes sense.

   ```
   gap> Ring( 2, E(4) );
   <ring with 2 generators>
   ```

3. **DefaultRing( \(r, s, \ldots\))**
   
   In the first form DefaultRing returns a ring that contains all the elements \(r, s, \ldots\) 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 the ring functions like Quotient, IsPrime, Factors, or Gcd if no explicit ring is given.
Ring (see 54.1.2) 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.

```plaintext
gap> DefaultRing( 2, E(4) );
GaussianIntegers
```

4 ▶ RingByGenerators( C )

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.

```plaintext
gap> RingByGenerators([ 2, E(4) ]);<ring with 2 generators>
```

5 ▶ DefaultRingByGenerators( coll )

```plaintext
gap> DefaultRingByGenerators([ 2, E(4) ]);GaussianIntegers
```

6 ▶ GeneratorsOfRing( R )

GeneratorsOfRing returns a list of elements such that the ring R is the closure of these elements under addition, multiplication, and taking additive inverses.

```plaintext
gap> R:=Ring( 2, 1/2 );<ring with 2 generators>
gap> GeneratorsOfRing( R );[ 2, 1/2 ]
```

7 ▶ AsRing( C )

If the elements in the collection C form a ring then AsRing returns this ring, otherwise fail is returned.

```plaintext
gap> ClosureRing( Integers, E(4) );<ring-with-one, with 2 generators>
```

8 ▶ Subring( R, gens ) F

SubringNC( R, gens ) F

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.

```plaintext
gap> R:=Integers;
Integers
gap> S:=Subring( R, [ 4, 6 ] );<ring with 2 generators>
gap> Parent( S );Integers
```

9 ▶ ClosureRing( R, r ) O

ClosureRing( R, S ) O

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.

```plaintext
gap> ClosureRing( Integers, E(4) );<ring-with-one, with 2 generators>
```

10 ▶ Quotient( R, r, s ) O

Quotient( r, s ) O

In the first form Quotient returns the quotient of the two ring elements r and s in the ring R. In the second form Quotient returns the quotient of the two ring elements r and s in their default ring. 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 /.)

```
gap> Quotient( 2, 3 );
fail

gap> Quotient( 6, 3 );
2
```

### 54.2 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 `IsIdealInParent` can be used to store whether a ring is an ideal in its parent.

(Whenever the term `Ideal` occurs without 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`, `GeneratorsOfRightIdeal`, and `GeneratorsOfTwoSidedIdeal`. The acting rings can be accessed as `LeftActingRingOfIdeal` and `RightActingRingOfIdeal`, 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` 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 `IsIdeal` is of course independent of the presence of the attribute values.

Ideals are constructed with `LeftIdeal`, `RightIdeal`, `TwoSidedIdeal`. Principal ideals of the form $x \cdot R$, $R \cdot x$, $R \cdot x \cdot R$ 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.

```
1 ↑ TwoSidedIdeal( R, gens[, "basis"] )
1 ↑ Ideal( R, gens[, "basis"] )
1 ↑ LeftIdeal( R, gens[, "basis"] )
1 ↑ RightIdeal( R, gens[, "basis"] )
```

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` or `RightActingRingOfIdeal` (or both) of $I$.

If $R$ is a left $F$-module then also $I$ is a left $F$-module, in particular the `LeftActingDomain` (see 55.1.11) 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 $R$. 
Chapter 54. Rings

Ideal is simply a synonym of TwoSidedIdeal.

gap> R := Integers;;
gap> I := Ideal( R, [ 2 ] );
<two-sided ideal in Integers, (1 generators)>

The effects of TwoSidedIdealNC, LeftIdealNC, and RightIdealNC are the same as TwoSidedIdeal, LeftIdeal, and RightIdeal, respectively (see 54.2.1), but they do not check whether all entries of gens lie in R.

The properties IsTwoSidedIdealInParent etc., are attributes of the ideal, and once known they are stored in the ideal.

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

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 or RightActingRingOfIdeal, gens can be accessed by GeneratorsOfTwoSidedIdeal.

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, gens can be accessed by GeneratorsOfLeftIdeal.

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, gens can be accessed by GeneratorsOfRightIdeal.

GeneratorsOfTwoSidedIdeal is a list of generators for the bi-ideal I, with respect to the action of LeftActingRingOfIdeal( I ) from the left and the action of RightActingRingOfIdeal( I ) from the right.
Note that \texttt{LeftActingRingOfIdeal}(I) and \texttt{RightActingRingOfIdeal}(I) coincide if I is a two-sided ideal.

\begin{verbatim}
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> I:= Ideal( A, [ One( A ) ] );;
gap> GeneratorsOfIdeal( I );
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] ]
\end{verbatim}

8 ▶ \texttt{GeneratorsOfLeftIdeal}( I )

is a list of generators for the left ideal I, with respect to the action of \texttt{LeftActingRingOfIdeal}( I ) from the left.

9 ▶ \texttt{GeneratorsOfRightIdeal}( I )

is a list of generators for the right ideal I, with respect to the action of \texttt{RightActingRingOfIdeal}( I ) from the right.

10 ▶ \texttt{LeftActingRingOfIdeal}( I )

\texttt{A}

11 ▶ \texttt{RightActingRingOfIdeal}( I )

\texttt{A}

\texttt{AsLeftIdeal}( R, S )

\texttt{O}

\texttt{AsRightIdeal}( R, S )

\texttt{O}

\texttt{AsTwoSidedIdeal}( R, S )

\texttt{O}

Let S be a subring of R.

If S is a left ideal in R then \texttt{AsLeftIdeal} returns this left ideal, otherwise \texttt{fail} is returned. If S is a right ideal in R then \texttt{AsRightIdeal} returns this right ideal, otherwise \texttt{fail} is returned. If S is a two-sided ideal in R then \texttt{AsTwoSidedIdeal} returns this two-sided ideal, otherwise \texttt{fail} is returned.

\begin{verbatim}
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)>
\end{verbatim}

54.3 Rings With One

\textbf{IsRingWithOne}( R )\hfill P

A ring-with-one in GAP is a ring (see 54.1.1) that is also a magma-with-one (see 33.1.2).

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.

2 ▶ \texttt{RingWithOne}( r, s, ... )

\texttt{F}

2 ▶ \texttt{RingWithOne}( C )

\texttt{F}

In the first form \texttt{RingWithOne} returns the smallest ring with one that contains all the elements r, s... etc. In the second form \texttt{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.
Chapter 54. Rings

gap> RingWithOne( [ 4, 6 ] );
<ring-with-one, with 2 generators>

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.

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.

gap> R:= RingWithOne( [ 4, 6 ] );
<ring-with-one, with 2 generators>

SubringWithOne returns the ring with one 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.

IsIntegralRing(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.

IsUniqueFactorizationRing A ring R is called a unique factorization ring if it is an integral ring (see 54.4.1), and every element has a unique factorization into irreducible elements, i.e., a unique representation as product of irreducibles (see 54.5.6). Unique in this context means unique up to permutations of the factors and up to multiplication of the factors by units (see 54.5.2).

Mathematically, a field should therefore also be a unique factorization ring, since every element is a unit. In GAP, however, at least at present fields do not lie in the filter IsUniqueFactorizationRing (see 54.4.2), since Operations such as Factors, Gcd, StandardAssociate 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 category since the factorization of an element needs not exist in a subring. As an example, consider the subring $\mathbb{Z}_4$ + 1 of the ring $\mathbb{Z}_4$ + 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.)

\[
\text{gap} \> \text{IsUniqueFactorizationRing( PolynomialRing( Rationals, 1 ) );}
\]

true

3. \texttt{IsLDistributive( C )}

is \texttt{true} if the relation \(a \ast (b + c) = (a \ast b) + (a \ast c)\) holds for all elements \(a, b, c\) in the collection \(C\), and \texttt{false} otherwise.

4. \texttt{IsRDistributive( C )}

is \texttt{true} if the relation \((a + b) \ast c = (a \ast c) + (b \ast c)\) holds for all elements \(a, b, c\) in the collection \(C\), and \texttt{false} otherwise.

5. \texttt{IsDistributive( C )}

is \texttt{true} if the collection \(C\) is both left and right distributive, and \texttt{false} otherwise.

\[
\text{gap} \> \text{IsDistributive( Integers );}
\]

true

6. \texttt{IsAnticommutative( R )}

is \texttt{true} if the relation \(a \ast b = -b \ast a\) holds for all elements \(a, b\) in the ring \(R\), and \texttt{false} otherwise.

7. \texttt{IsZeroSquaredRing( R )}

is \texttt{true} if \(a \ast a\) is the zero element of the ring \(R\) for all \(a\) in \(R\), and \texttt{false} otherwise.

8. \texttt{IsJacobianRing( R )}

is \texttt{true} if the Jacobi identity holds in \(R\), and \texttt{false} otherwise. The Jacobi identity means that \(x \ast (y \ast z) + z \ast (x \ast y) + y \ast (z \ast x)\) is the zero element of \(R\), for all elements \(x, y, z\) in \(R\).

\[
\text{gap} \> \text{L:= FullMatrixLieAlgebra( GF( 5 ), 7 );}
\]

\texttt{<Lie algebra over GF(5), with 13 generators>}

\[
\text{gap} \> \text{IsJacobianRing( L );}
\]

true

54.5 Units and Factorizations

1. \texttt{IsUnit( R, r )}

2. \texttt{IsUnit( r )}

In the first form \texttt{IsUnit} returns \texttt{true} if \(r\) is a unit in the ring \(R\). In the second form \texttt{IsUnit} returns \texttt{true} if the ring element \(r\) is a unit in its default ring (see 54.1.3).

An element \(r\) is called a \texttt{unit} in a ring \(R\), if \(r\) has an inverse in \(R\).

\texttt{IsUnit} may call \texttt{Quotient}.

3. \texttt{Units( R )}

\texttt{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 \texttt{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.
Chapter 54. Rings

```gap
gap> Units( GaussianIntegers );
[ -1, 1, -E(4), E(4) ]
gap> Units( GF( 16 ) );
<group with 1 generators>
```

3➤ IsAssociated( R, r, s )
➤ IsAssociated( r, s )

In the first form IsAssociated returns true if the two ring elements r and s are associated in the ring R and false otherwise. In the second form IsAssociated returns true if the two ring elements r and s are associated in their default ring (see 54.1.3) and false otherwise.

Two elements r and s of a ring R are called associated if there is a unit u of R such that ru = s.

4➤ Associates( R, r )
➤ Associates( r )

In the first form Associates returns the set of associates of r in the ring R. In the second form Associates returns the set of associates of the ring element r in its default ring (see 54.1.3).

Two elements r and s of a ring R are called associated if there is a unit u of R such that ru = s.

```gap
gap> Associates( Integers, 2 );
[ -2, 2 ]
gap> Associates( GaussianIntegers, 2 );
[ -2, 2, -2*E(4), 2*E(4) ]
```

5➤ StandardAssociate( R, r )
➤ StandardAssociate( r )

In the first form StandardAssociate returns the standard associate of the ring element r in the ring R. In the second form StandardAssociate returns the standard associate of the ring element r in its default ring (see 54.1.3).

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.

```gap
gap> x := Indeterminate( Rationals, "x" );;
gap> StandardAssociate( -x^2-x+1 );
-1+x+x^2
```

6➤ IsIrreducibleRingElement( R, r )
➤ IsIrreducibleRingElement( r )

In the first form IsIrreducibleRingElement returns true if the ring element r is irreducible in the ring R and false otherwise. In the second form IsIrreducibleRingElement returns true if the ring element r is irreducible in its default ring (see 54.1.3) and false otherwise.

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 54.5.1). Each prime element (see 54.5.7) is irreducible.

```gap
gap> IsIrreducibleRingElement( Integers, 2 );
true
```

7➤ IsPrime( R, r )
➤ IsPrime( r )

In the first form IsPrime returns true if the ring element r is a prime in the ring R and false otherwise. In the second form IsPrime returns true if the ring element r is a prime in its default ring (see 54.1.3) and false otherwise.
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 54.5.6) is a prime.

\[
\text{Factors}( R, r )
\]

In the first form \texttt{Factors} returns the factorization of the ring element \( r \) in the ring \( R \). In the second form \texttt{Factors} returns the factorization of the ring element \( r \) in its default ring (see 54.1.3). The factorization is returned as a list of primes (see 54.5.7). Each element in the list is a standard associate (see 54.5.5) except the first one, which is multiplied by a unit as necessary to have \( \text{Product}( \text{Factors}( R, r ) ) = r \). This list is usually also sorted, thus smallest prime factors come first. If \( r \) is a unit or zero, \texttt{Factors}( \( R, r \) ) = [ \( r \) ].

\[
gap> \text{x := Indeterminate( Rationals, "x" );;}
\]
\[
gap> \text{Factors( x^3+3*x^2+3*x+1 );}
\]
\[
[ 1+x, 1+x, 1+x ]
\]

### 54.6 Euclidean Rings

A ring \( R \) is called a Euclidean ring if it is an integral ring and there exists a function \( \delta \), 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 \texttt{GAP} the Euclidean degree \( \delta \) 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.

\[
gap> \text{IsEuclideanRing( GaussianIntegers );}
\]
\[
\text{true}
\]

\[
\text{EuclideanDegree}( R, r )
\]

In the first form \texttt{EuclideanDegree} returns the Euclidean degree of the ring element in the ring \( R \). In the second form \texttt{EuclideanDegree} returns the Euclidean degree of the ring element \( r \) in its default ring. \( R \) must of course be a Euclidean ring (see 54.6.1).

\[
gap> \text{EuclideanDegree( GaussianIntegers, 3 );}
\]
\[
9
\]

\[
\text{EuclideanQuotient}( R, r, m )
\]

In the first form \texttt{EuclideanQuotient} returns the Euclidean quotient of the ring elements \( r \) and \( m \) in the ring \( R \). In the second form \texttt{EuclideanQuotient} returns the Euclidean quotient of the ring elements \( r \) and \( m \) in their default ring. The ring \( R \) must be a Euclidean ring (see 54.6.1) otherwise an error is signalled.

\[
gap> \text{EuclideanQuotient( 8, 3 );}
\]
\[
2
\]

\[
\text{EuclideanRemainder}( R, r, m )
\]

In the first form \texttt{EuclideanRemainder} returns the remainder of the ring element \( r \) modulo the ring element \( m \) in the ring \( R \). In the second form \texttt{EuclideanRemainder} returns the remainder of the ring element \( r \) modulo the ring element \( m \) in their default ring. The ring \( R \) must be a Euclidean ring (see 54.6.1) otherwise an error is signalled.
Chapter 54. Rings

gap> EuclideanRemainder( 8, 3 );
2

5▷ QuotientRemainder( R, r, s )
▷ QuotientRemainder( r, s )

In the first form QuotientRemainder returns the Euclidean quotient and the Euclidean remainder of the ring elements \( r \) and \( m \) in the ring \( R \). In the second form QuotientRemainder returns the Euclidean quotient and the Euclidean remainder of the ring elements \( r \) and \( m \) in their default ring as pair of ring elements. The ring \( R \) must be a Euclidean ring (see 54.6.1) otherwise an error is signalled.

gap> QuotientRemainder( GaussianIntegers, 8, 3 );
[ 3, -1 ]

54.7 Gcd and Lcm

1▷ Gcd( R, r1, r2, ... )
▷ Gcd( R, list )
▷ Gcd( r1, r2, ... )
▷ Gcd( list )

In the first two forms Gcd returns the greatest common divisor of the ring elements \( r1, r2, \ldots \) resp. of the ring elements in the list \( list \) in the ring \( R \). In the second two forms Gcd returns the greatest common divisor of the ring elements \( r1, r2, \ldots \) resp. of the ring elements in the list \( list \) in their default ring (see 54.1.3). \( R \) must be a Euclidean ring (see 54.6.1) so that QuotientRemainder (see 54.6.5) can be applied to its elements. Gcd returns the standard associate (see 54.5.5) of the greatest common divisors.

A greatest common divisor of the elements \( r1, r2, \ldots \) of the ring \( R \) is an element of largest Euclidean degree (see 54.6.2) that is a divisor of \( r1, r2, \ldots \).

We define \( Gcd( r, 0_R ) = Gcd( 0_R, r ) = StandardAssociate( r ) \) and \( Gcd( 0_R, 0_R ) = 0_R \).

gap> Gcd( Integers, [ 10, 15 ] );
5

2▷ GcdOp( R, r, s )
▷ GcdOp( r, s )

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.

3▷ GcdRepresentation( R, r1, r2, ... )
▷ GcdRepresentation( R, list )
▷ GcdRepresentation( r1, r2, ... )
▷ GcdRepresentation( list )

In the first two forms GcdRepresentation returns the representation of the greatest common divisor of the ring elements \( r1, r2, \ldots \) resp. of the ring elements in the list \( list \) in the ring \( R \). In the second two forms GcdRepresentation returns the representation of the greatest common divisor of the ring elements \( r1, r2, \ldots \) resp. of the ring elements in the list \( list \) in their default ring (see 54.1.3). \( R \) must be a Euclidean ring (see 54.6.1) so that Gcd (see 54.7.1) can be applied to its elements.

The representation of the gcd \( g \) of the elements \( r1, r2, \ldots \) of a ring \( R \) is a list of ring elements \( s1, s2, \ldots \) of \( R \), such that \( g = s1r1 + s2r2 + \cdots \). That this representation exists can be shown using the Euclidean algorithm, which in fact can compute those coefficients.
GcdRepresentationOp is the operation to compute the representation of the greatest common divisor of two ring elements \( r, s \) in the ring \( R \) or in their default ring, respectively.

In the first two forms \( \text{Lcm} \) returns the least common multiple of the ring elements \( r_1, r_2, \ldots \) resp. of the ring elements in the list \( \text{list} \) in the ring \( R \). In the second two forms \( \text{Lcm} \) returns the least common multiple of the ring elements \( r_1, r_2, \ldots \) resp. of the ring elements in the list \( \text{list} \) in their default ring (see 54.1.3). \( R \) must be a Euclidean ring (see 54.6.1) so that \( \text{Gcd} \) (see 54.7.1) can be applied to its elements. \( \text{Lcm} \) returns the standard associate (see 54.5.5) of the least common multiples.

A least common multiple of the elements \( r_1, r_2, \ldots \) of the ring \( R \) is an element of smallest Euclidean degree (see 54.6.2) that is a multiple of \( r_1, r_2, \ldots \).

We define \( \text{Lcm}( r, 0_R ) = \text{Lcm}( 0_R, r ) = \text{StandardAssociate}( r ) \) and \( \text{Lcm}( 0_R, 0_R ) = 0_R \).

\( \text{Lcm} \) uses the equality \( \text{lcm}(m, n) = m \ast n / \text{gcd}(m, n) \) (see 54.7.1).

\( \text{LcmOp} \) 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.

In the first form \( \text{QuotientMod} \) returns the quotient of the ring elements \( r \) and \( s \) modulo the ring element \( m \) in the ring \( R \). In the second form \( \text{QuotientMod} \) returns the quotient of the ring elements \( r \) and \( s \) modulo the ring element \( m \) in their default ring (see 54.1.3). \( R \) must be a Euclidean ring (see 54.6.1) so that \( \text{EuclideanRemainder} \) (see 54.6.4) can be applied. If the modular quotient does not exist, \( \text{fail} \) 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 0 (if \( r \) is divisible by \( m \)) or the Euclidean degree of \( q \) is strictly smaller than the Euclidean degree of \( m \).

\( \text{PowerMod} \) returns the \( e \)-th power of the ring element \( r \) modulo the ring element \( m \) in the ring \( R \). In the second form \( \text{PowerMod} \) returns the \( e \)-th power of the ring element \( r \) modulo the ring element \( m \) in their default ring (see 54.1.3). \( e \) must be an integer. \( R \) must be a Euclidean ring (see 54.6.1) so that \( \text{EuclideanRemainder} \) (see 54.6.4) can be applied to its elements.

If \( e \) is positive the result is \( r^e \) modulo \( m \). If \( e \) is negative then \( \text{PowerMod} \) 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 \( \text{PowerMod} \) returns \( \text{PowerMod}( R, i, -e, m ) \).
\textbf{PowerMod} reduces the intermediate values modulo \( m \), improving performance drastically when \( e \) is large and \( m \) small.

\begin{verbatim}
gap> PowerMod( 12, 100000, 7 );
2
\end{verbatim}

\textbf{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.

\begin{verbatim}
gap> InterpolatedPolynomial( Integers, [ 1, 2, 3 ], [ 5, 7, 0 ] );
-6+31/2*x-9/2*x^2
\end{verbatim}
55 Modules (preliminary)

55.1 Generating modules

1► IsLeftOperatorAdditiveGroup( D )
   
   A domain \( D \) lies in IsLeftOperatorAdditiveGroup if it is an additive group that is closed under scalar multiplication from the left, and such that \( \lambda \cdot (x + y) = \lambda \cdot x + \lambda \cdot y \) for all scalars \( \lambda \) and elements \( x, y \in D \).

2► IsLeftModule( M )
   
   A domain \( M \) lies in IsLeftModule if it lies in IsLeftOperatorAdditiveGroup, and the set of scalars forms a ring, and \((\lambda + \mu) \cdot x = \lambda \cdot x + \mu \cdot x\) for scalars \( \lambda, \mu \) and \( x \in M \), and scalar multiplication satisfies \( \lambda \cdot (\mu \cdot x) = (\lambda \cdot \mu) \cdot x \) for scalars \( \lambda, \mu \) and \( x \in M \).

   gap> V := FullRowSpace( Rationals, 3 );
   ( Rationals^3 )
   gap> IsLeftModule( V );
   true

3► GeneratorsOfLeftOperatorAdditiveGroup( D )
   
   returns a list of elements of \( D \) that generates \( D \) as a left operator additive group.

4► GeneratorsOfLeftModule( M )
   
   returns a list of elements of \( M \) that generate \( M \) as a left module.

   gap> V := FullRowSpace( Rationals, 3 );;
   gap> GeneratorsOfLeftModule( V );
   [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]

5► AsLeftModule( R, D )
   
   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.

   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)>

6► IsRightOperatorAdditiveGroup( D )
   
   A domain \( D \) lies in IsRightOperatorAdditiveGroup if it is an additive group that is closed under scalar multiplication from the right, and such that \((x + y) \cdot \lambda = x \cdot \lambda + y \cdot \lambda\) for all scalars \( \lambda \) and elements \( x, y \in D \).
55. Modules (preliminary)

7 ▶ IsRightModule( M )

A domain \( M \) lies in \( \text{IsRightModule} \) if it lies in \( \text{IsRightOperatorAdditiveGroup} \), and the set of scalars forms a ring, and \( x + (\lambda + \mu) = x + \lambda + x + \mu \) for scalars \( \lambda, \mu \) and \( x \in M \), and scalar multiplication satisfies \( (x + \mu) \lambda = x \lambda + (\mu \lambda) \) for scalars \( \lambda, \mu \) and \( x \in M \).

8 ▶ GeneratorsOfRightOperatorAdditiveGroup( D )

returns a list of elements of \( D \) that generates \( D \) as a right operator additive group.

9 ▶ GeneratorsOfRightModule( M )

returns a list of elements of \( M \) that generate \( M \) as a left module.

10 ▶ LeftModuleByGenerators( R, gens )

returns the left module over \( R \) generated by \( gens \).

\[
gap\text{coll} := \left[ \begin{array}{c} Z(2), 0*Z(2) \\ 0*Z(2), Z(2) \\ Z(2), Z(2) \end{array} \right];;
\]
\[
\text{V} := \text{LeftModuleByGenerators}( \text{GF}(16), \text{coll} );;
\]
\[
\text{W} := \text{Submodule}( \text{V}, \left[ \text{coll}[1], \text{coll}[2] \right] );
\]
\[
\text{Parent}( \text{W} ) = \text{V};
\]
true

55.2 Submodules

1 ▶ Submodule( M, gens )

is the left module generated by the collection \( gens \), with parent module \( M \). The second form generates the submodule of \( M \) for that the list \( gens \) is known to be a list of basis vectors; in this case, it is not checked whether \( gens \) really are linearly independent and whether all in \( gens \) lie in \( M \).

\[
gap\text{coll} := \left[ \begin{array}{c} Z(2), 0*Z(2) \\ 0*Z(2), Z(2) \\ Z(2), Z(2) \end{array} \right];;
\]
\[
\text{V} := \text{LeftModuleByGenerators}( \text{GF}(16), \text{coll} );;
\]
\[
\text{W} := \text{Submodule}( \text{V}, [\text{coll}[1], \text{coll}[2]] );
\]
\[
\text{Parent}( \text{W} ) = \text{V};
\]
true

2 ▶ SubmoduleNC( M, gens )

SubmoduleNC does the same as Submodule, except that it does not check whether all in \( gens \) lie in \( M \).

3 ▶ ClosureLeftModule( M, m )

is the left module generated by the left module generators of \( M \) and the element \( m \).

\[
\text{V} := \text{LeftModuleByGenerators}( \text{Rationals}, \left[ \begin{array}{c} 1, 0, 0 \\ 0, 1, 0 \end{array} \right] );
\]
\[
\text{ClosureLeftModule}( \text{V}, [1, 1, 1] );
\]
\[
\text{TrivialSubmodule}( \text{M} )
\]
returns the zero submodule of \( M \).

\[
\text{V} := \text{LeftModuleByGenerators}( \text{Rationals}, \left[ \begin{array}{c} 1, 0, 0 \\ 0, 1, 0 \end{array} \right] );
\]
\[
\text{TrivialSubmodule}( \text{V} );
\]
\[
\text{vector space over Rationals, with 0 generators}
Section 3. Free Modules

55.3 Free Modules

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 30.10.1) of a free left module is defined as the characteristic of its left acting domain (see 55.1.11).

FreeLeftModule( $R$, $gens$ )
FreeLeftModule( $R$, $gens$, zero )
FreeLeftModule( $R$, $gens$, "basis" )
FreeLeftModule( $R$, $gens$, zero, "basis" )

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 30.3 for the general meaning of “generators” in GAP.) In particular, FreeLeftModule is not an equivalent of commands such as FreeGroup (see 35.2.1) 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 59.8.4) in the finite dimensional case and polynomial rings (see 64.14.1) 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 63).

AsFreeLeftModule( $F$, $D$ )

if the domain $D$ is a free left module over $F$, then AsFreeLeftModule( $F$, $D$ ) returns the domain $D$ viewed as free left module over $F$.

Dimension( $M$ )

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 also for free left modules over different left acting domains.)

IsFiniteDimensional( $M$ )

is true if $M$ is a free left module that is finite dimensional over its left acting domain, and false otherwise.
gap> V := FreeLeftModule( Rationals, [[1, 0], [0, 1], [1, 1]] );;
gap> IsFiniteDimensional( V );
true

6 ► UseBasis( V, gens )

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 from 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.

gap> V := FreeLeftModule( Rationals, [[1, 0], [0, 1], [1, 1]] );;
gap> UseBasis( V, [[1, 0], [1, 1]] );

7 ► IsRowModule( V )

A row module is a free left module whose elements are row vectors.

8 ► IsMatrixModule( V )

A matrix module is a free left module whose elements are matrices.

9 ► IsFullRowModule( M )

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 with entries in $R$ and such that the dimension is equal to the length of the row vectors.

Several functions delegate their tasks to full row modules, for example Iterator and Enumerator.

10 ► FullRowModule( R, n )

is the row module $R^n$, for a ring $R$ and a nonnegative integer $n$.

gap> V := FullRowModule( Integers, 5 ); ( Integers^5 )

11 ► IsFullMatrixModule( M )

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 matrices with entries in $R$ and such that the dimension is equal to the number of entries in each matrix.

12 ► FullMatrixModule( R, m, n )

is the row module $R^{[m,n]}$, for a ring $R$ and nonnegative integers $m$ and $n$.

gap> FullMatrixModule( GaussianIntegers, 3, 6 ); ( GaussianIntegers^[3,6] )

13 ► IsHandledByNiceBasis( M )

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.
A **division ring** is a ring (see Chapter 54) 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 57) and abelian number fields (see Chapter 58), in particular the field of rationals (see Chapter 16).

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 59). 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 56.1.2), thus each field is in fact a field extension in a natural way, which is used by functions such as `Norm` and `Trace` (see 56.3).

### 56.1 Generating Fields

1. ▶ **IsDivisionRing( D )**

   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 `LeftActingDomain( D )`.

2. ▶ **IsField( D )**

   A **field** is a commutative division ring (see 56.1.1 and 33.4.9).

   ```gap
   gap> IsField( GaloisField(16) );  # the field with 16 elements
   true
   gap> IsField( Rationals );        # the field of rationals
   true
   gap> q:= QuaternionAlgebra( Rationals );;  # a noncommutative division ring
   gap> IsField( q );  IsDivisionRing( q );
   false
   true
   gap> mat:= [ [ 1 ] ];  a:= Algebra( Rationals, [ mat ] );
   gap> IsDivisionRing( a );  # an algebra not constructed as a division ring
   false
   ```

3. ▶ **Field( z, ... )**

   **Field** returns the smallest field $K$ that contains all the elements $z, \ldots$, 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 $K$. In the third form, Field constructs the field generated by the field $F$ and the elements in the list list, as a vector space over $F$.

4► DefaultField( $z$, ... )
4► DefaultField( list )

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 (see 56.1.3). 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.

```gap
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
```

5► DefaultFieldByGenerators( [ $z$, ... ] )

returns the default field containing the elements $z$, ... This field may be bigger than the smallest field containing these elements.

6► GeneratorsOfDivisionRing( $D$ )

generators with respect to addition, multiplication, and taking inverses (the identity cannot be omitted ...)

7► GeneratorsOfField( $F$ )

generators with respect to addition, multiplication, and taking inverses. This attribute is the same as GeneratorsOfDivisionRing (see 56.1.6).

8► DivisionRingByGenerators( [ $z$, ... ] )
8► DivisionRingByGenerators( $F$, [ $z$, ... ] )

The first version returns a division ring as vector space over FieldOverItselfByGenerators( gens ).

9► AsDivisionRing( $C$ )
9► AsDivisionRing( $F$, $C$ )
9► AsField( $C$ )
9► AsField( $F$, $C$ )

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.
56.2 Subfields of Fields

1. \texttt{Subfield( F, gens )} F
2. \texttt{SubfieldNC( F, gens )} F
3. \texttt{FieldOverItselfByGenerators( \{ z, \ldots \} )} O
4. \texttt{PrimitiveElement( D )} A
5. \texttt{PrimeField( D )} A
6. \texttt{DegreeOverPrimeField( F )} A
7. \texttt{DefiningPolynomial( F )} A
8. \texttt{RootOfDefiningPolynomial( F )} A
9. \texttt{FieldExtension( F, poly )} O
10. \texttt{Subfields( F )} A

Constructs the subfield of \( F \) generated by \( gens \).

This operation is needed for the call of \texttt{Field} or \texttt{FieldByGenerators} without explicitly given subfield, in order to construct a left acting domain for such a field.

is an element of \( D \) that generates \( D \) as a division ring together with the left acting domain.

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.

A division ring is a prime field if it is equal to its prime field (see 56.2.4).

is the degree of the field \( F \) over its prime field (see 56.2.4).

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 \texttt{RootOfDefiningPolynomial} (see 56.2.8).

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 \texttt{DefiningPolynomial} (see 56.2.7).

is the field obtained on adjoining a root of the irreducible polynomial \( poly \) to the field \( F \).

is the set of all subfields of the field \( F \).

56.3 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 \( \varphi_\alpha \) on \( L \), and for a fixed \( K \)-basis of \( L \), we can associate to \( \alpha \) the matrix \( M_\alpha \) (over \( K \)) of this mapping.

The norm of \( \alpha \) is defined as the determinant of \( M_\alpha \), the trace of \( \alpha \) is defined as the trace of \( M_\alpha \), the minimal polynomial \( \mu_\alpha \) and the trace polynomial \( \chi_\alpha \) of \( \alpha \) are defined as the minimal polynomial (see 56.3.2) and the characteristic polynomial (see 24.12.1 and 56.3.3) of \( M_\alpha \). (Note that \( \mu_\alpha \) depends only on \( K \) whereas \( \chi_\alpha \) depends on both \( L \) and \( K \).)

Thus norm and trace of \( \alpha \) are elements of \( K \), and \( \mu_\alpha \) and \( \chi_\alpha \) are polynomials over \( K \), \( \chi_\alpha \) being a power of \( \mu_\alpha \), and the degree of \( \chi_\alpha \) equals the degree of the field extension \( L > K \).

The conjugates of \( \alpha \) in \( L \) are those roots of \( \chi_\alpha \) (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 \( \alpha \) are all roots of \( \chi_\alpha \) (with multiplicity), the set of conjugates equals the roots of \( \mu_\alpha \), the norm of \( \alpha \) equals the product and the trace of \( \alpha \) equals the sum of the conjugates of \( \alpha \), and the Galois group in the sense of the above definition equals the usual Galois group.

Note that \( \text{MinimalPolynomial}( F, z ) \) is a polynomial over \( F \), whereas \( \text{Norm}( F, z ) \) is the norm of the element \( z \) in \( F \) w.r.t. the field extension \( F > \text{LeftActingDomain}(F) \).

1. **GaloisGroup** \( ( F ) \)

The **Galois group** of a field \( F \) is the group of all field automorphisms of \( F \) that fix the subfield \( K = \text{LeftActingDomain}(F) \) pointwise.

Note that the field extension \( F > K \) need **not** be a Galois extension.

```gap
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
```

2. **MinimalPolynomial** \( ( F, z[, ind] ) \)

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 \( \text{MinimalPolynomial} \) for rings, see 64.8.1.)

```gap
gap> MinimalPolynomial( Rationals, E(8) );
1+x_1^4
gap> MinimalPolynomial( CF(4), E(8) );
-E(4)+x_1^2
gap> MinimalPolynomial( CF(8), E(8) );
-E(8)+x_1
```

3. **TracePolynomial** \( ( L, K, z[, inum] ) \)

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 \( \text{inum} \) (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 \( \text{CharacteristicPolynomial} \) (see 24.12.1) that call **TracePolynomial** in the case of field extensions.

```gap
gap> TracePolynomial( CF(8), Rationals, E(8) );
1+x_1^4
gap> TracePolynomial( CF(16), Rationals, E(8) );
1+2*x_1^4+x_1^8
```

4. **Norm** \( ( z ) \)

**Norm** \( ( L, z ) \)

**Norm** \( ( L, K, z ) \)

**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 \( \text{LeftActingDomain}( L ) \) is taken as default for the subfield \( K \), and if no field is given then \( \text{DefaultField}( z ) \) is taken as default for \( L \).
Section 3. Galois Action

5 Trace( z ) A
▶ Trace( mat ) A
▶ Trace( L, z ) O
▶ Trace( L, K, z ) O

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.

6 Conjugates( z ) A
▶ Conjugates( L, z ) O
▶ Conjugates( L, K, z ) O

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 \( L \), respectively of the default field of \( z \). The result list need not be sorted.

```
gap> Norm( E(8) ); Norm( CF(8), E(8) );
1
1
gap> Norm( CF(8), CF(4), E(8) );
-2z

gap> Norm( AsField( CF(4), CF(8) ), E(8) );
-2z

E(8)
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 ]
```

The default methods for field elements are as follows. MinimalPolynomial solves a system of linear equations. TracePolynomial computes the appropriate power of the minimal polynomial, Norm and Trace values are obtained as coefficients of the characteristic polynomial, and Conjugates 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 returns the sorted list of images (with multiplicity) of the element under the Galois group, Norm computes the product of the conjugates, Trace computes the sum of the conjugates, TracePolynomial and MinimalPolynomial compute the product of linear factors \( x - c \) with \( c \) ranging over the conjugates and the set of conjugates, respectively.
Let $F$ be a field that is a Galois extension of its subfield \texttt{LeftActingDomain}(F). Then \texttt{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 56.3.1) 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 [Art68].

\begin{verbatim}
    gap> NormalBase( CF(5) );
    gap> NormalBase( CF(4) );
    [ 1/2-1/2*E(4), 1/2+1/2*E(4) ]
\end{verbatim}
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 56) also applies to finite fields.

In the following, the term finite field element is used to denote GAP objects in the category IsFFE (see 57.1.1), 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, and in fact GAP does currently not support such fields.

Special representations exist for row vectors and matrices over small finite fields (see sections 23.2 and 24.13).

57.1 Finite Field Elements

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 28) lies in IsFFECollection, and a collection of such collections (e.g., a matrix) lies in IsFFECollColl.

For creating elements of a finite field the function Z can be used. The call $Z( p^d )$ returns the designated generator of the multiplicative group of the finite field with $p^d$ elements. $p$ must be a prime and $p^d$ must be less than or equal to $2^{16} = 65536$.

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.

The additive neutral element is $0*Z(p)$. It is different from the integer 0 in subtle ways. First IsInt( 0*Z(p) ) (see 14.1.1) is false and IsFFE( 0*Z(p) ) (see 57.1.1) 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 14.1.1) is false and IsFFE( Z(p)^0 ) (see 57.1.1) 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 $p^n - 1$ and $Z(p^m)$ is an element of order $p^m - 1$. 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 57.5.1).
of characteristic $p$. Those polynomials were defined by J. H. Conway, and many of them were computed by R. A. Parker.

Elements of prime fields of order larger than $2^{16}$ can be handled using the machinery of Residue Class Rings (see section 14.4).

```gap
gap> a:= Z( 32 );
Z(2^5)
gap> a+a;
0*Z(2)
gap> a*a;
Z(2^5)^2
```

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 $b$. Finite field elements are ordered in the following way. If the two elements lie in fields of different characteristics the one that lies in the field with the smaller characteristic is smaller. If the two elements lie in different fields of the same characteristic the one that lies in the smaller field is smaller. If the two elements lie in the same field and one is the zero and the other is not, the zero element is smaller. If the two elements lie in the same field and both are nonzero, and are represented as $Z(p^d)_{i1}$ and $Z(p^d)_{i2}$ respectively, then the one with the smaller $i$ is smaller.

For the comparison of finite field elements with other GAP objects, see 4.11.

```gap
gap> Z( 16 )^10 = Z( 4 )^2; # this illustrates the embedding of GF(4) in GF(16)
true
gap> 0 < 0*Z(101);
true
gap> Z(256) > Z(101);
false
```

### 57.2 Operations for Finite Field Elements

Since finite field elements are scalars, the operations `Characteristic`, `One`, `Zero`, `Inverse`, `AdditiveInverse`, `Order` can be applied to them (see 30.10). Contrary to the situation with other scalars, `Order` is defined also for the zero element in a finite field, with value 0.

```gap
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
```

`DegreeFFE( z )` O
`
DegreeFFE( vec )` O
`
DegreeFFE( mat )` O

`DegreeFFE` returns the degree of the smallest finite field $F$ containing the element $z$, respectively all elements of the vector $vec$ over a finite field (see 23), or matrix $mat$ over a finite field (see 24).
Section 3. Creating Finite Fields

gap> DegreeFFE( Z(16)^10 );
2
gap> DegreeFFE( Z(16)^11 );
4
gap> DegreeFFE( [ Z(2^13), Z(2^10) ] );
130

2 ▶ LogFFE( z, r )

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, or if \( z \) is not a power of \( r \).

The discrete logarithm of an element \( z \) with respect to a root \( r \) is the smallest nonnegative integer \( i \) such that \( r^i = z \).

gap> LogFFE( Z(409)^116, Z(409) );
116
gap> LogFFE( Z(409)^116, Z(409)^2 );
2

3 ▶ IntFFE( z )

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 \( i * \text{One}(z) = z \).

The correspondence between elements from a finite prime field of characteristic \( p \) 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 15.2.3).

IntFFE is installed as a method for the operation Int (see 14.1.3) with argument a finite field element.

gap> IntFFE( Z(13) );
2
gap> IntFFE( Z(409) );
21

4 ▶ IntVecFFE( vecffe )

is the list of integers corresponding to the vector \( \text{vecffe} \) of finite field elements in a prime field (see 57.2.3).

57.3 Creating Finite Fields

DefaultField (see 56.1.4) and DefaultRing (see 54.1.3) for finite field elements are defined to return the smallest field containing the given elements.

gap> DefaultField( [ Z(4), Z(4)^2 ] );
GF(2^2)
gap> DefaultField( [ Z(4), Z(8) ] );
GF(2^6)

1 ▶ GaloisField( p^d )
2 ▶ GaloisField( p, d )
3 ▶ GaloisField( S, d )
4 ▶ GaloisField( p, pol )
5 ▶ GaloisField( S, pol )

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.
Chapter 57. Finite Fields

The first argument specifies the subfield $S$ over which the new field $F$ is to be taken. It can be a prime 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 with the Conway polynomial (see 57.5.1) of degree $d$ over the subfield $S$. If it is an irreducible polynomial $pol$ over $S$, the new field is constructed as polynomial extension of the subfield $S$ with this polynomial; in this case, $pol$ is accessible as the value of `DefiningPolynomial` (see 56.2.7) for the new field, and a root of $pol$ in the new field is accessible as the value of `RootOfDefiningPolynomial` (see 56.2.8).

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 56.3.1, 56.3.6, 56.3.4, 56.3.5, 56.3.2, 56.3.3).

The field is regarded as a vector space (see 59) over the given subfield, so this determines the dimension and the canonical basis of the field.

```gap
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) ]
```

A primitive root of a finite field is a generator of its multiplicative group. A primitive root is always a primitive element (see 56.2.3), the converse is in general not true.

```gap
gap> f := GF( 3^5 );
GF(3^5)
gap> PrimitiveRoot( f );
Z(3^5)
```

57.4 FrobeniusAutomorphism

returns the Frobenius automorphism of the finite field $F$ as a field homomorphism (see 31.11).

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 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 56.3.1).
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 $ij \pmod{\text{Size}(F) - 1}$. The zeroth power of $f$ is $\text{IdentityMapping}(F)$.

57.5 Conway Polynomials

ConwayPolynomial($p$, $n$) 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, ..., $p-1$\} 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 64.4.12). 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 $\Phi_{d,p}(x^{\frac{p^n-1}{d}}) \equiv 0 \pmod{\Phi_{n,p}(x)}$ holds.

With respect to the ordering defined above, $\Phi_{n,p}$ shall be minimal.

```
gap> ConwayPolynomial(2, 5); ConwayPolynomial(3, 7);
Z(2)^0+x_1^2+x_1^5
Z(3)^0-x_1^2+x_1^7
```
Abelian Number Fields (preliminary)

1. CyclotomicField(n)
2. CyclotomicField(gens)
3. CyclotomicField(subfield, n)
4. CyclotomicField(subfield, gens)

The first version creates the n-th cyclotomic field. The second version creates the cyclotomic field generated by gens. In both cases the field can be generated as an extension of a designated subfield.

2. AbelianNumberField(n, stab)

fixed field of the group generated by stab (prime residues modulo n) in the cyclotomic field with conductor n.

3. GaussianRationals

is the field Q(i) of Gaussian rationals.

4. IsGaussianRationals(obj)

58.1 Integral Bases for Abelian Number Fields

1. ZumbroichBase(n, m)

is the set of exponents e for which E(n)^e belongs to the (generalized) Zumbroich base of the cyclotomic field Q_n, viewed as vector space over Q_m.

The base, the base conversion and the reduction to the minimal cyclotomic field are described in [Zum89].

Note: For n \equiv 2 \pmod 4 we have ZumbroichBase(n, 1) = 2 * ZumbroichBase(n/2, 1). However, List(ZumbroichBase(n, 1), x -> E(n)^x) = List(ZumbroichBase(n/2, 1), x -> E(n/2)^x).

2. LenstraBase(n, stabilizer, super, m)

is a list of lists of integers, each list indexing the exponents of an orbit of a subgroup of stabilizer on n-th roots of unity.

super is a list representing a supergroup of stabilizer which shall act consistently with the action of stabilizer, i.e., each orbit of supergroup is a union of orbits of stabilizer.

m is a positive integer. The basis described by the returned list is an integral basis over the cyclotomic field Q_m.

Note: The elements are in general not sets, since the first element is always an element of ZumbroichBase(n, m); this property is used by NF and Coefficients.

Note: stabilizer must not contain the stabilizer of a proper cyclotomic subfield of the n-th cyclotomic field.

For details about the bases see [Bre97].
58.2 Operations for Abelian Number Fields

For operations for cyclotomics, e.g., \texttt{Conductor} (18.1.5), see Chapter 18.

1. \texttt{IsNumberField( F )}
   
   returns \texttt{true} if the field \( F \) is a number field, or \texttt{false} otherwise.

   A \textbf{number field} is a finite extension of a prime field in characteristic zero.

2. \texttt{IsAbelianNumberField( F )}
   
   returns the abelian number field of the field \( F \).

   An \textbf{abelian number field} is a number field that is a Galois extension of the prime field with abelian Galois group (see 56.3.1).

3. \texttt{GaloisStabilizer( F )}
   
   For an abelian number field \( F \), \texttt{GaloisStabilizer} returns the set of all integers \( k \) in the range from 1 to the conductor of \( F \) such that the field automorphism induced by raising roots of unity in \( F \) to the \( k \)-th power acts trivially on \( F \).

4. \texttt{ComplexConjugate( z )}
   
   For a cyclotomic number \( z \), \texttt{ComplexConjugate} returns \texttt{GaloisCyc( z, -1 )}. For a quaternion \( z = c_1e + c_2i + c_3j + c_4k \), \texttt{ComplexConjugate} returns \( c_1e - c_2i - c_3j - c_4k \).

58.3 Galois Groups of Abelian Number Fields

The \textbf{Galois automorphisms} of the cyclotomic field \( \mathbb{Q}(e_n) \) (see Chapter 18) are given by linear extension of the maps \( *k : e_n \mapsto e_k^n \) where \( 1 \leq k < n \) and \( \text{Gcd}( n, k ) = 1 \) hold (see 18.5.1). Note that this action is \textbf{not} equal to exponentiation of cyclotomics, i.e., for general cyclotomics \( z \), \( z^*k \) is different from \( z^k \):

\[
gap> ( E(5) + E(5)^4 )^2; \quad \text{GaloisCyc}( E(5) + E(5)^4, 2 );
\]
\[
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
\]
\[
E(5)^2+E(5)^3
\]

For \( \text{Gcd}(n,k) \neq 1 \), the map \( e_n \mapsto e_k^n \) is \textbf{not} a field automorphism but only a linear map.

\[
gap> \text{GaloisCyc}( E(5)+E(5)^4, 5 ); \quad \text{GaloisCyc}( ( E(5)+E(5)^4 )^2, 5 );
\]
\[
2
\]
\[
-6
\]

1. \texttt{ANFAutomorphism( F, k )}
   
   Let \( F \) be an abelian number field \( F \) and \( k \) an integer. If \( k \) is coprime to the conductor (see 18.1.5) of \( F \) then \texttt{ANFAutomorphism} returns the automorphism of \( F \) defined as the linear extension of the map that raises each root of unity in \( F \) to its \( k \)-th power, otherwise an error is signalled.

\[
gap> f:= \text{CF}(25);
\]
\[
\text{CF}(25)
\]
\[
gap> \alpha:= \text{ANFAutomorphism}( f, 2 );
\]
\[
\text{ANFAutomorphism}( \text{CF}(25), 2 )
\]
\[
gap> \alpha^2;
\]
\[
\text{ANFAutomorphism}( \text{CF}(25), 4 )
\]
\[
gap> \text{Order}( \alpha );
\]
\[
20
\]
\[
gap> E(5)^\alpha;
\]
\[
E(5)^2
\]

The \textbf{Galois group} \( \text{Gal}(\mathbb{Q}(e_n) \mathbb{Q}) \) of the field extension \( \mathbb{Q}(e_n) / \mathbb{Q} \) is isomorphic to the group \( (\mathbb{Z}/n\mathbb{Z})^* \) of prime residues modulo \( n \), via the isomorphism from \( (\mathbb{Z}/n\mathbb{Z})^* \) to \( \text{Gal}(\mathbb{Q}(e_n), \mathbb{Q}) \) defined by \( k + n\mathbb{Z} \mapsto (z \mapsto z^k) \).
Chapter 58. Abelian Number Fields (preliminary)

The Galois group of the field extension \( \mathbb{Q}(e_n)/L \) with any abelian number field \( L \subseteq \mathbb{Q}(e_n) \) is simply the factor group of \( \text{Gal}(\mathbb{Q}_n, \mathbb{Q}) \) 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 \texttt{GeneratorsPrimeResidues} (see 15.1.4).

```gap
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) ])
gap> GeneratorsOfGroup( g );
[ ANFAutomorphism( CF(15), 11 ), ANFAutomorphism( CF(15), 7 ) ]
```

### 58.4 Gaussians

1. \texttt{GaussianIntegers} is the ring of Gaussian integers. This is the subring \( \mathbb{Z}[i] \) of the complex numbers, where \( i \) is a square root of \(-1\).

2. \texttt{IsGaussianIntegers(obj)} is the defining category for the domain \texttt{GaussianIntegers}. 
A vector space in GAP is a free left module (see 55.3.1) over a division ring (see Chapter 56). Whenever we talk about an $F$-vector space $V$ then $V$ is an additive group (see 53.1.6) 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 $\text{LeftActingDomain}$ (see 55.1.11).

The characteristic (see 30.10.1) of a vector space is equal to the characteristic of its left acting domain. Vector spaces in GAP are always left vector spaces, $\text{IsLeftVectorSpace}$ and $\text{IsVectorSpace}$ are synonyms.

### 59.1 Constructing Vector Spaces

For a field $F$ and a collection $\text{gens}$ of vectors, $\text{VectorSpace}$ returns the $F$-vector space spanned by the elements in $\text{gens}$.

The optional argument $\text{zero}$ can be used to specify the zero element of the space; $\text{zero}$ must be given if $\text{gens}$ is empty. The optional string "$\text{basis}$" indicates that $\text{gens}$ is known to be linearly independent over $F$, in particular the dimension of the vector space is immediately set; note that $\text{Basis}$ (see 59.4.2) need not return the basis formed by $\text{gens}$ if the argument "$\text{basis}$" is given.

```gap
gap> V:= VectorSpace( Rationals, \[ \[ 1, 2, 3 \], \[ 1, 1, 1 \] \] );
<vector space over Rationals, with 2 generators>
```

For an $F$-vector space $V$ and a list or collection $\text{gens}$ that is a subset of $V$, $\text{Subspace}$ returns the $F$-vector space spanned by $\text{gens}$; if $\text{gens}$ is empty then the trivial subspace (see 59.2.2) of $V$ is returned. The parent (see 30.7) of the returned vector space is set to $V$.

$\text{SubspaceNC}$ does the same as $\text{Subspace}$, except that it omits the check whether $\text{gens}$ is a subset of $V$.

The optional string "$\text{basis}$" indicates that $\text{gens}$ is known to be linearly independent over $F$. In this case the dimension of the subspace is immediately set, and both $\text{Subspace}$ and $\text{SubspaceNC}$ do not check whether $\text{gens}$ really is linearly independent and whether $\text{gens}$ is a subset of $V$.

```gap
gap> V:= VectorSpace( Rationals, \[ \[ 1, 2, 3 \], \[ 1, 1, 1 \] \] );;
gap> W:= Subspace( V, \[ \[ 0, 1, 2 \] \] );
<vector space over Rationals, with 1 generators>
```

Let $F$ be a division ring and $D$ a domain. If the elements in $D$ form an $F$-vector space then $\text{AsVectorSpace}$ returns this $F$-vector space, otherwise $\text{fail}$ is returned.

$\text{AsVectorSpace}$ can be used for example to view a given vector space as a vector space over a smaller or larger division ring.
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

4 \textbf{AsSubspace}( V, U )

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 \texttt{AsSubspace} returns this vector space, with parent set to $V$ (see 59.1.3). Otherwise \texttt{fail} is returned.

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

59.2 Operations and Attributes for Vector Spaces

\textbf{GeneratorsOfLeftVectorSpace}( V )

For an $F$-vector space $V$, \texttt{GeneratorsOfLeftVectorSpace} returns a list of vectors in $V$ that generate $V$ as an $F$-vector space.

gap> GeneratorsOfVectorSpace( FullRowSpace( Rationals, 3 ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]

\textbf{TrivialSubspace}( V )

For a vector space $V$, \texttt{TrivialSubspace} returns the subspace of $V$ that consists of the zero vector in $V$.

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) ] ]
59.3 Domains of Subspaces of Vector Spaces

Let $V$ be a finite vector space. In the first form, `Subspaces` returns the domain of all subspaces of $V$. In the second form, $k$ must be a nonnegative integer, and `Subspaces` returns the domain of all $k$-dimensional subspaces of $V$.

Special `Size` and `Iterator` methods are provided for these domains.

59.4 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 55.3.4). In particular, as each basis is a list (see Chapter 21), it has a length (see 21.17.5), and the $i$-th vector of $B$ can be accessed as $B[i]$.

```
gap> V := Rationals^3;
( Rationals^3 )
gap> B := Basis( V );
CanonicalBasis( ( Rationals^3 )
gap> Length( B );
3
```

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` (see 59.5.1), `Coefficients` (see 59.5.3), `LinearCombination` (see 59.5.4), and `UnderlyingLeftModule` (see 59.5.2). These and other operations for arbitrary bases are described in 59.5.

For special kinds of bases, further operations are defined (see 59.6).

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 59.4.4).
Chapter 59. Vector Spaces

Bases handled by nice bases delegate the work to bases of isomorphic left modules over the same left acting domain (see 59.10).

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 59.8), $\text{Basis}(V)$ is a semi-echelonized basis (see 59.8.7) that uses Gaussian elimination; such a basis is of the third kind. $\text{Basis}(V, \text{vectors})$ 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 59.10. Examples are non-Gaussian row and matrix spaces, and subspaces of finite fields and abelian number fields that are themselves not fields.

1. \texttt{IsBasis( obj )}

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 59.5.3). Bases are constructed by $\text{Basis}$ (see 59.4.2). Each basis is an immutable list, the $i$-th entry being the $i$-th basis vector.

(See 59.7 for mutable bases.)

\begin{verbatim}
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
\end{verbatim}

2. \texttt{Basis( V )}

\texttt{Basis( V, vectors )}

\texttt{BasisNC( V, vectors )}

Called with a free left $F$-module $V$ as the only argument, $\text{Basis}$ returns an $F$-basis of $V$ whose vectors are not further specified.

If additionally a list $\text{vectors}$ of vectors in $V$ is given that forms an $F$-basis of $V$ then $\text{Basis}$ returns this basis; if $\text{vectors}$ is not linearly independent over $F$ or does not generate $V$ as a free left $F$-module then $\text{fail}$ is returned.

$\text{BasisNC}$ does the same as $\text{Basis}$ for two arguments, except that it does not check whether $\text{vectors}$ form a basis.

If no basis vectors are prescribed then $\text{Basis}$ need not compute basis vectors; in this case, the vectors are computed in the first call to $\text{BasisVectors}$.

\begin{verbatim}
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 ] ]
)  
\end{verbatim}

3. \texttt{CanonicalBasis( V )}

If the vector space $V$ supports a canonical basis then $\text{CanonicalBasis}$ returns this basis, otherwise $\text{fail}$ 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 55.1.11) 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 59.8).

If one designs a new kind of vector spaces (see 59.11) and defines a canonical basis for these spaces then the `CanonicalBasis` method one installs (see 2.2.1 in “Programming in GAP”) must not call `Basis`. On the other hand, one probably should install a `Basis` method that simply calls `CanonicalBasis`, the value of the method (see 2.2 and 2.3 in “Programming in GAP”) being `CANONICAL_BASIS_FLAGS`.

```gap
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 ] ]
```

4▶ `RelativeBasis( B, vectors )` O
5▶ `RelativeBasisNC( B, vectors )` O

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.

## 59.5 Operations for Vector Space Bases

1▶ `BasisVectors( B )` A

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.

```gap
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 ] ]
```

2▶ `UnderlyingLeftModule( B )` A

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 59.4.2); furthermore, in some cases it is convenient to test membership of a vector in the module before computing coefficients w.r.t. a basis.

```gap
gap> B:= Basis( GF(2)^6 );
UnderlyingLeftModule( B );
( GF(2)^6 )
```

3▶ `Coefficients( B, v )` O

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.
Chapter 59. Vector Spaces

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

If \( B \) is a basis of length \( n \), say, and \( \text{coeff} \) is a row vector of the same length as \( B \), \text{LinearCombination} returns the vector \( \sum_{i=1}^{n} \text{coeff}[i] \times B[i] \).

If \( \text{vectors} \) and \( \text{coeff} \) are homogeneous lists of the same length \( n \), say, \text{LinearCombination} returns the vector \( \sum_{i=1}^{n} \text{coeff}[i] \times \text{vectors}[i] \). Perhaps the most important usage is the case where \( \text{vectors} \) forms a basis.

For a basis \( B \) of the free left \( F \)-module \( V \) of dimension \( n \), say, \text{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 59.8.4) of dimension \( n \) over \( F \), in the succession given by the default enumerator of this row space.

For a basis \( B \) of the free left \( F \)-module \( V \) of dimension \( n \), say, \text{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 59.8.4) of dimension \( n \) over \( F \), in the succession given by the default enumerator of this row space.
Section 6. Operations for Special Kinds of Bases

59.6 Operations for Special Kinds of Bases

1 ▶ IsCanonicalBasis( B )

If the underlying free left module V of the basis B supports a canonical basis (see 59.4.3) then IsCanonicalBasis returns true if B is equal to the canonical basis of V, and false otherwise.

2 ▶ IsIntegralBasis( B )

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.

3 ▶ IsNormalBasis( B )

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 56.3.1) of the field extension F/S, and false otherwise.

4 ▶ StructureConstantsTable( B )

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 6.2 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 [v1, ..., vn] is determined by the so-called structure matrices M_k = [m_{ijk}]_{1 \leq k \leq n}. Let a = [a_1, ..., a_n] and b = [b_1, ..., b_n]. Then

\[
\left( \sum_i a_i v_i \right) \left( \sum_j b_j v_j \right) = \sum_{i,j} a_i b_j (v_i v_j) = \sum_k \left( \sum_j a_i m_{i,j,k} \right) b_j v_k = \sum_k (a M_k^r b^t) v_k.
\]

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.

```gap
gap> A := QuaternionAlgebra( Rationals );
gap> StructureConstantsTable( Basis( A ) );
```

579
59.7 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 59.5.2).

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 (see 59.7.3), IsContainedInSpan (see 59.7.5), CloseMutableBasis (see 59.7.6), ImmutableBasis (see 59.7.4), Coefficients (see 59.5.3), and BasisVectors (see 59.5.1). ShallowCopy (see 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. 59.4), 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 59.10) 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.

1. IsMutableBasis( MB )

Every mutable basis lies in the category IsMutableBasis.

2. MutableBasis( R, vectors[, zero] )

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!

\[ \text{gap> MB:= MutableBasis( Rationals, \{ \{ 1, 2, 3 \}, \{ 0, 1, 0 \} \} );} \]
\[ \text{<mutable basis over Rationals, 2 vectors>} \]

3. NrBasisVectors( MB )

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.
Section 8. Row and Matrix Spaces

```gap
gap> MB := MutableBasis( Rationals, [[1, 1], [2, 2]] );;
gap> NrBasisVectors( MB );
1

4\textbf{ ImmutableBasis( MB[, V] )}

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 (see 59.5.2) 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.

```gap
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>
```

5\textbf{ IsContainedInSpan( MB, v )}

For a mutable basis $MB$ over the coefficient ring $R$, say, and a vector $v$, IsContainedInSpan returns \texttt{true} is $v$ lies in the $R$-span of the current basis vectors of $MB$, and \texttt{false} otherwise.

```gap
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] );
<mutable basis over Rationals, 3 vectors>
gap> IsContainedInSpan( MB, [1, 0, 0] );
true
```

6\textbf{ CloseMutableBasis( MB, v )}

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 $v$.

\textbf{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 59.8.7).

```gap
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] );
<mutable basis over Rationals, 3 vectors>
gap> MB;
<mutable basis over Rationals, 3 vectors>
```

59.8 Row and Matrix Spaces

1\textbf{ IsRowSpace( V )}

A \textbf{row space} in GAP is a vector space that consists of row vectors (see Chapter 23).

2\textbf{ IsMatrixSpace( V )}

A \textbf{matrix space} in GAP is a vector space that consists of matrices (see Chapter 24).

3\textbf{ IsGaussianSpace( V )}

The filter \texttt{IsGaussianSpace} (see 13.2) for the row space (see 59.8.1) or matrix space (see 59.8.2) $V$ 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 \textbf{Gaussian} vector space. Bases for Gaussian spaces can be computed using Gaussian elimination for a given list of vector space generators.

```
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

4► FullRowSpace( F, n )

For a field $F$ and a nonnegative integer $n$, FullRowSpace returns the $F$-vector space that consists of all row vectors (see 23) of length $n$ with entries in $F$.

An alternative to construct this vector space is via $F^n$.

    gap> FullRowSpace( GF( 9 ), 3 );
    ( GF(3^2)^3 )
    gap> GF(9)^3;  # the same as above
    ( GF(3^2)^3 )

5► FullMatrixSpace( F, m, n )

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 24.1.1) with entries in $F$.

If $m = n$ then the result is in fact an algebra (see 60.4.4).

An alternative to construct this vector space is via $F^{m \times n}$.

    gap> FullMatrixSpace( GF(2), 4, 5 );
    ( GF(2)^[ 4, 5 ] )
    gap> GF(2)^[ 4, 5 ];  # the same as above
    ( GF(2)^[ 4, 5 ] )

6► DimensionOfVectors( M )

For a left module $M$ that consists of row vectors (see 55.3.7), DimensionOfVectors returns the common length of all row vectors in $M$. For a left module $M$ that consists of matrices (see 55.3.8), DimensionOfVectors returns the common matrix dimensions (see 24.3.1) of all matrices in $M$.

    gap> DimensionOfVectors( GF(2)^5 );
    5
    gap> DimensionOfVectors( GF(2)^[2,3] );
    [ 2, 3 ]

7► IsSemiEchelonized( B )

Let $B$ be a basis of a Gaussian row or matrix space $V$, say (see 59.8.3) 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. 24.9.1).

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. 24.9.4).
Section 8. Row and Matrix Spaces

gap> V := GF(2)^2;;
gap> B1 := Basis( V, [ [ 0, 1 ], [ 1, 0 ] ] * Z(2) );;
gap> IsSemiEchelonized( B1 );
true

Let $V$ be a Gaussian row or matrix vector space over the field $F$ (see 59.8.3, 59.8.1, 59.8.2).

Called with $V$ as the only argument, SemiEchelonBasis returns a basis of $V$ that has the property IsSemiEchelonized (see 59.8.7).

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 SemiEchelonBasis for two arguments, except that it is not checked whether $vectors$ form a semi-echelonized basis.

gap> V := GF(2)^2;;
gap> B := SemiEchelonBasis( V );
gap> Print( BasisVectors( B ), \n );
gap> B := SemiEchelonBasis( V, [ [ 0, 1 ], [ 1, 1 ] ] * Z(2) );
gap> Coefficients( B, [ 0, 1 ] * Z(2) );
[ Z(2)^0, 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

IsCanonicalBasisFullRowModule returns true if $B$ is the canonical basis (see 59.6.1) of a full row module (see 55.3.9), and false otherwise.

The canonical basis of a Gaussian row space is defined as the unique semi-echelonized (see 59.8.7) 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.

IsCanonicalBasisFullMatrixModule returns true if $B$ is the canonical basis (see 59.6.1) of a full matrix module (see 55.3.11), and false otherwise.

The canonical basis of a Gaussian matrix space is defined as the unique semi-echelonized (see 59.8.7) basis for which the list of concatenations of the basis vectors forms the canonical basis of the corresponding Gaussian row space.
Chapter 59. Vector Spaces

11 ▶ NormedRowVectors( V )

For a finite Gaussian row space $V$ (see 59.8.1, 59.8.3), \texttt{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 \texttt{OnLines} (see 39.2.12), which yields the natural action on one-dimensional subspaces of $V$ (see also 59.3.1).

\begin{verbatim}
gap> vecs:= NormedRowVectors( GF(3)^2 );
gap> Action( GL(2,3), vecs, OnLines );
group([ (3,4), (1,2,4) ])
\end{verbatim}

12 ▶ SiftedVector( B, v )

Let $B$ be a semi-echelonized basis (see 59.8.7) of a Gaussian row or matrix space $V$ (see 59.8.3), and $v$ a row vector or matrix, respectively, of the same dimension as the elements in $V$. \texttt{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 59.7) of a Gaussian row or matrix space.

\begin{verbatim}
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 ]
\end{verbatim}

59.9 Vector Space Homomorphisms

Vector space homomorphisms (or linear mappings) are defined in Section 31.10. 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.

1 ▶ LeftModuleGeneralMappingByImages( V, W, gens, imgs )

Let $V$ and $W$ be two left modules over the same left acting domain $R$, say, and $\text{gens}$ and $\text{imgs}$ lists (of the same length) of elements in $V$ and $W$, respectively. \texttt{LeftModuleGeneralMappingByImages} returns the general mapping with source $V$ and range $W$ that is defined by mapping the elements in $\text{gens}$ to the corresponding elements in $\text{imgs}$, and taking the $R$-linear closure.

$\text{gens}$ 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 31.2.3).

\begin{verbatim}
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]] );
gap> IsMapping( f );
false
\end{verbatim}

2 ▶ LeftModuleHomomorphismByImages( V, W, gens, imgs )
\begin{verbatim}
\end{verbatim}

\begin{verbatim}
\end{verbatim}

Let $V$ and $W$ be two left modules over the same left acting domain $R$, say, and $\text{gens}$ and $\text{imgs}$ lists (of the same length) of elements in $V$ and $W$, respectively. \texttt{LeftModuleHomomorphismByImages} returns the left
Section 9. Vector Space Homomorphisms

R-module homomorphism with source \( V \) and range \( W \) that is defined by mapping the elements in \( \text{gens} \) to the corresponding elements in \( \text{ imgs} \).

If \( \text{ gens} \) does not generate \( V \) or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then \( \text{ fail} \) is returned. For creating a possibly multi-valued mapping from \( V \) to \( W \) that respects addition, multiplication, and scalar multiplication, \text{LeftModuleGeneralMappingByImages} \( \) can be used.

\text{LeftModuleHomomorphismByImagesNC} \( \) does the same as \text{LeftModuleHomomorphismByImages}, except that it omits all checks.

\begin{verbatim}
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 ] ] \rightarrow \([ [ 1, 0, 1 ], [ 1, 2, 3 ] ]\)
gap> Image( f, \([1,1]\) );
\([ [1,0,1], [1,2,3] ]\)
\end{verbatim}

\text{LeftModuleHomomorphismByMatrix} \( BS, \text{ matrix}, BR \) \( \)

Let \( BS \) and \( BR \) be bases of the left \( R \)-modules \( V \) and \( W \), respectively. \text{LeftModuleHomomorphismByMatrix} returns the \( R \)-linear mapping from \( V \) to \( W \) that is defined by the matrix \( \text{ matrix } \) 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 \( \text{ matrix } \).

\begin{verbatim}
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)>
\end{verbatim}

\text{NaturalHomomorphismBySubspace} \( V, \text{ W} \) \( \)

For an \( R \)-vector space \( V \) and a subspace \( W \) of \( V \), \text{NaturalHomomorphismBySubspace} returns the \( R \)-linear mapping that is the natural projection of \( V \) onto the factor space \( V / W \).

\begin{verbatim}
gap> V:= Rationals^3;;
gap> W:= VectorSpace( Rationals, \([ [ 1, 1, 1 ] ]\) );;
gap> f:= NaturalHomomorphismBySubspace( V, W );
<linear mapping by matrix, ( Rationals^3 ) -> ( Rationals^2 )>
\end{verbatim}

\text{Hom} \( F, V, W \) \( \)

For a field \( F \) and two vector spaces \( V \) and \( W \) that can be regarded as \( F \)-modules (see 55.1.5), \text{Hom} returns the \( F \)-vector space of all \( F \)-linear mappings from \( V \) to \( W \).

\begin{verbatim}
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 of dimension 2 over Rationals> )
gap> Dimension( H );
4
\end{verbatim}

\text{End} \( F, V \) \( \)

For a field \( F \) and a vector space \( V \) that can be regarded as an \( F \)-module (see 55.1.5), \text{End} returns the \( F \)-algebra of all \( F \)-linear mappings from \( V \) to \( V \).
A full hom module is a module of all $R$-linear mappings between two left $R$-modules. The function \texttt{Hom} (see 59.9.5) can be used to construct a full hom module.

\begin{verbatim}
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> H:= Hom( Rationals, V, W );;
gap> IsFullHomModule( H );
true
\end{verbatim}

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 59.4.3) because it depends on the stored bases of source and range.

\begin{verbatim}
gap> IsPseudoCanonicalBasisFullHomModule( Basis( H ) );
true
\end{verbatim}

If an $F$-vector space $V$ is in the filter \texttt{IsLinearMappingsModule} then this expresses that $V$ consists of linear mappings, and that $V$ is handled via the mechanism of nice bases (see 59.10) in the following way. Let $S$ and $R$ be the source and the range, respectively, of each mapping in $V$. Then the \texttt{NiceFreeLeftModuleInfo} value of $V$ is a record with the components \texttt{basissource} (a basis $B_S$ of $S$) and \texttt{basisrange} (a basis $B_R$ of $R$), and the \texttt{NiceVector} value of $v \in V$ is defined as the matrix of the $F$-linear mapping $v$ w.r.t. the bases $B_S$ and $B_R$.

\section{59.10 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 \textbf{nice free left module} of $V$, and $C$ the \textbf{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 \texttt{IsBasisByNiceBasis} (see 59.10.5), and stores $C$ as value of the attribute \texttt{NiceBasis} (see 59.10.4). $V$ indicates the intended behaviour by the filter \texttt{IsHandledByNiceBasis} (see 59.10.6), and stores $W$ as value of the attribute \texttt{NiceFreeLeftModule} (see 59.10.1).

The bijection between $V$ and $W$ is implemented by the functions \texttt{NiceVector} (see 59.10.2) and \texttt{UglyVector} (see 59.10.2); additional data needed to compute images and preimages can be stored as value of \texttt{NiceFreeLeftModuleInfo} (see 59.10.3).
Section 11. How to Implement New Kinds of Vector Spaces

1 ▶ NiceFreeLeftModule( V )

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.

2 ▶ NiceVector( V, v )
▶ UglyVector( V, r )

NiceVector and UglyVector provide the linear bijection between the free left module V and \( W := \text{NiceFreeLeftModule}( V ) \).

If \( v \) lies in the elements family of the family of \( V \) then \( \text{NiceVector}( v ) \) 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 \( \text{UglyVector}( r ) \) 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 \( \text{UglyVector}( V, \text{NiceVector}( V, v ) ) = v \). If \( r \) lies in \( W \) (which usually can be checked) then \( \text{NiceVector}( V, \text{UglyVector}( V, r ) ) = r \).

(This allows one to implement for example a membership test for \( V \) using the membership test in \( W \).)

3 ▶ NiceFreeLeftModuleInfo( V )

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 \( \text{NiceVector} \) and \( \text{UglyVector} \) (see 59.10.2).

4 ▶ NiceBasis( B )

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 \( \text{NiceBasis} \) then \( \text{NiceBasis}( B ) \) is obtained as \( \text{Basis}( \text{NiceFreeLeftModule}( V ) ) \). If basis vectors are stored then \( \text{NiceBasis}( B ) \) is the result of the call of \( \text{Basis} \) with arguments \( \text{NiceFreeLeftModule}( V ) \) and the \( \text{NiceVector} \) 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 \( \text{GeneratorsOfLeftModule} \) of the underlying left modules of \( B \) and the result of \( \text{NiceBasis} \) correspond via \( \text{NiceVector} \) and \( \text{UglyVector} \).

5 ▶ IsBasisByNiceBasis( B )

This filter indicates that the basis \( B \) delegates tasks such as the computation of coefficients (see 59.5.3) to a basis of an isomorphic “nicer” free left module.

6 ▶ IsHandledByNiceBasis( M )

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.

59.11 How to Implement New Kinds of Vector Spaces

1 ▶ DeclareHandlingByNiceBasis( name, info )
▶ InstallHandlingByNiceBasis( name, record )

These functions are used to implement a new kind of free left modules that shall be handled via the mechanism of nice bases (see 59.10).

\( name \) must be a string, a filter \( f \) with this name is created, and a logical implication from \( f \) to \( \text{IsHandledByNiceBasis} \) (see 59.10.6) 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 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 methods call CheckForHandlingByNiceBasis (see 59.11.3), which sets the appropriate filter for the desired left module if applicable.

2 ▶ NiceBasisFiltersInfo

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 59.8.3).

3 ▶ CheckForHandlingByNiceBasis( $R$, $gens$, $M$, $zero$ )

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.
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 59) is also applicable to algebras.

1 ▶ InfoAlgebra

is the info class for the functions dealing with algebras (see 7.4).

60.1 Constructing Algebras by Generators

1 ▶ Algebra( F, gens ) F
1 ▶ Algebra( F, gens, zero ) F
1 ▶ Algebra( F, gens, "basis" ) F
1 ▶ Algebra( F, gens, zero, "basis" ) F

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).

    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

2 ▶ AlgebraWithOne( F, gens ) F
2 ▶ AlgebraWithOne( F, gens, zero ) F
2 ▶ AlgebraWithOne( F, gens, "basis" ) F
2 ▶ AlgebraWithOne( F, gens, zero, "basis" ) F

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).
Chapter 60. Algebras

```plaintext
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 ] ]
```

60.2 Constructing Algebras as Free Algebras

1. FreeAlgebra( R, rank )
2. FreeAlgebra( R, rank, name )
3. FreeAlgebra( R, name1, name2, ... )

is a free (nonassociative) algebra of rank rank over the ring R. Here name, and name1, name2,... are optional strings that can be used to provide names for the generators.

```plaintext
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))
```

2. FreeAlgebraWithOne( R, rank )
3. FreeAlgebraWithOne( R, rank, name )
4. FreeAlgebraWithOne( R, name1, name2, ... )

is a free (nonassociative) algebra-with-one of rank rank over the ring R. Here name, and name1, name2,... are optional strings that can be used to provide names for the generators.

```plaintext
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 ]
```

3. FreeAssociativeAlgebra( R, rank )
4. FreeAssociativeAlgebra( R, rank, name )
5. FreeAssociativeAlgebra( R, name1, name2, ... )

is a free associative algebra of rank rank over the ring R. Here name, and name1, name2,... are optional strings that can be used to provide names for the generators.

```plaintext
gap> A:= FreeAssociativeAlgebra( GF( 5 ), 4, "a" );
<algebra over GF(5), with 4 generators>
```
Section 3. Constructing Algebras by Structure Constants

For an introduction into structure constants and how they are handled by GAP, we refer to Section 6.2 of the user’s tutorial.

\[ \text{EmptySCTable}( \text{dim, zero} ) \]
\[ \text{EmptySCTable}( \text{dim, zero, "symmetric"} ) \]
\[ \text{EmptySCTable}( \text{dim, zero, "antisymmetric"} ) \]

\text{EmptySCTable} returns a structure constants table for an algebra of dimension \text{dim}, describing trivial multiplication. \text{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.

For filling up the structure constants table, see 60.3.2.

\[ \text{SetEntrySCTable}( T, i, j, \text{list} ) \]

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 \( \text{list} \).

If \( T \) is known to be antisymmetric or symmetric then also the value \( T[j][i] \) is set.

\( \text{list} \) 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 \( \text{list} \) 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.

\[ \text{GapInputSCTable}( T, \text{varname} ) \]

is a string that describes the structure constants table \( T \) in terms of \text{EmptySCTable} and \text{SetEntrySCTable}. The assignments are made to the variable \text{varname}.

\[ \text{TestJacobi}( T ) \]

tests whether the structure constants table \( T \) satisfies the Jacobi identity \( v_i \cdot (v_j \cdot v_k) + v_j \cdot (v_k \cdot v_i) + v_k \cdot (v_i \cdot v_j) = 0 \) 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.

\[
\text{gap} > T := \text{EmptySCTable}( 2, 0, \text{"antisymmetric"});;
\text{gap} > \text{SetEntrySCTable}( T, 1, 2, [ 1, 2 ]);;
\text{gap} > \text{TestJacobi}( T );
\text{true}
\]

returns a free left module \(A\) over the ring \(R\), with multiplication defined by the structure constants table \(sctable\). Here \(name\) and \(name1\), \(name2\), ... are optional strings that can be used to provide names for the elements of the canonical basis of \(A\). \(names\) is a list of strings that can be entered instead of the specific names \(name1\), \(name2\), ... . 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 \(R\).

\[
\text{gap} > T := \text{EmptySCTable}( 2, 0 );; \\
\text{gap} > \text{SetEntrySCTable}( T, 1, 1, [ 1/2, 1, 2/3, 2 ] );; \\
\text{gap} > A := \text{AlgebraByStructureConstants}( \text{Rationals}, T ); \\
\text{<algebra of dimension 2 over Rationals>}
\]

\[
\text{gap} > b := \text{BasisVectors}( \text{Basis}( A ) );;
\text{gap} > b[1]^2;
(1/2)*v.1+(2/3)*v.2
\]

\[
\text{gap} > b[1]*b[2];
0*v.1
\]

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 \(T\).

\[
\text{gap} > T := \text{EmptySCTable}( 2, 0 );; \\
\text{gap} > \text{SetEntrySCTable}( T, 1, 1, [ 1 ] );; \\
\text{gap} > \text{SetEntrySCTable}( T, 1, 2, [ 1 ] );; \\
\text{gap} > \text{SetEntrySCTable}( T, 2, 1, [ 1 ] );; \\
\text{gap} > \text{IdentityFromSCTable}( T ); \\
[ 1, 0 ]
\]

Let \(T\) be a structure constants table of an algebra \(A\) of dimension \(n\). \(QuotientFromSCTable( T, num, den )\) 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 \(num = x\cdot den\). 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 \(a_k = \sum_{i,j} c_{ij} x_j c_{ijk}\) for all \(k\). Here \(c_{ijk}\) denotes the structure constants with respect to \(B\). This means that (as a vector) \(a = xM\) with \(M_{jk} = \sum_{i=1}^n c_{ijk}\).
Section 4. Some Special Algebras

60.4 Some Special Algebras

QuaternionAlgebra( F )
QuaternionAlgebra( F, a, b )

is a quaternion algebra over the field F with parameters a and b in F, 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 = ac, ij = -ji = k, ik = -ki = a, jk = be, jk = -kj = bi, kk = -abe. The default value for both a and b is -1 ∈ F.

The GeneratorsOfAlgebra (see 60.8.1) and CanonicalBasis (see 59.4.3) value of an algebra constructed with QuaternionAlgebra is the list [e, i, j, k].

The embedding of the field GaussianRationals into a quaternion algebra A over Rationals is not uniquely determined. One can specify one as a vector space homomorphism that maps 1 to the first algebra generator of A, and E(4) to one of the others.

ComplexificationQuat( vector )
ComplexificationQuat( matrix )

Let A = eF ⊕ iF ⊕ jF ⊕ kF be a quaternion algebra over the field F of cyclotomics, with basis (e, i, j, k).

If v = v1 + v2j is a row vector over A with v1 = ew1 + iw2 and v2 = ev3 + iw4 then ComplexificationQuat( v ) is the concatenation of w1 + E(4)w2 and w3 + E(4)w4.

If M = M1 + M2j is a matrix over A with M1 = eN1 + iN2 and M2 = eN3 + iN4 then ComplexificationQuat( M ) is the block matrix A over eF ⊕ iF such that A(1,1) = N1 + E(4)N2, A(2,2) = N1 - E(4)N2, A(1,2) = N3 + E(4)N4 and A(2,1) = -N3 + E(4)N4.

Then ComplexificationQuat(v)*ComplexificationQuat(M) = ComplexificationQuat(v*M), since

vM = v1M1 + v2jM1 + v1M2j + v2jM2j = (v1M1 - v2jM2) + (v1M2 + v2jM1)j.

OctaveAlgebra( F )
The algebra of octonions over F.

FullMatrixAlgebra( R, n )
MatrixAlgebra( R, n )
MatAlgebra( R, n )
is the full matrix algebra R^{n×n}, for a ring R and a nonnegative integer n.
Chapter 60. Algebras

\[ \text{gap> A:=FullMatrixAlgebra( Rationals, 20 );} \]
\[ \text{( Rationals^[ 20, 20 ])} \]
\[ \text{gap> Dimension( A );} \]
\[ 400 \]

\textbf{NullAlgebra( R )}

The zero-dimensional algebra over \( R \).

\[ \text{gap> A:= NullAlgebra( Rationals );} \]
\[ \langle \text{algebra over Rationals} \rangle \]
\[ \text{gap> Dimension( A );} \]
\[ 0 \]

\section*{60.5 Subalgebras}

\textbf{Subalgebra( A, gens )}

\textbf{Subalgebra( A, gens, "basis" )}

is the \( F \)-algebra generated by \( gens \), with parent algebra \( A \), where \( F \) is the left acting domain of \( A \).

\textbf{Note} that being a subalgebra of \( A \) means to be an algebra, to be contained in \( A \), \textbf{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 \textbf{not} checked whether \( gens \) really are linearly independent and whether all elements in \( gens \) lie in \( A \).

\[ \text{gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];} \]
\[ \text{gap> A:= Algebra( Rationals, [ m ] );} \]
\[ \langle \text{algebra over Rationals, with 1 generators} \rangle \]
\[ \text{gap> B:= Subalgebra( A, [ m^2 ] );} \]
\[ \langle \text{algebra over Rationals, with 1 generators} \rangle \]

\textbf{SubalgebraNC( A, gens )}

\textbf{SubalgebraNC( A, gens, "basis" )}

\textbf{SubalgebraNC} constructs the subalgebra generated by \( gens \), only it does not check whether all elements in \( gens \) lie in \( A \).

\[ \text{gap> m:= RandomMat( 3, 3 );} \]
\[ \text{gap> A:= Algebra( Rationals, [ m ] );} \]
\[ \langle \text{algebra over Rationals, with 1 generators} \rangle \]
\[ \text{gap> SubalgebraNC( A, [ IdentityMat( 3, 3 ) ], "basis" );} \]
\[ \langle \text{algebra of dimension 1 over Rationals} \rangle \]

\textbf{SubalgebraWithOne( A, gens )}

\textbf{SubalgebraWithOne( A, gens, "basis" )}

is the algebra-with-one generated by \( gens \), with parent algebra \( A \).

The optional third argument "basis" may be added if it is known that the elements from \( gens \) are linearly independent. Then it is \textbf{not} checked whether \( gens \) really are linearly independent and whether all elements in \( gens \) lie in \( A \).
Section 6. 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 54. Here we give examples demonstrating the use of ideals in algebras. For an introduction into the construction of quotient algebras we refer to Chapter 6.2 of the user’s tutorial.

```gap
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
SubalgebraWithOneNC does not check whether all elements in gens lie in A.

gap> m := RandomMat( 3, 3 );;
A := Algebra( Rationals, [ m ] );;
gap> SubalgebraWithOneNC( A, [ m ] );
<algebra-with-one over Rationals, with 1 generators>

5 TrivialSubalgebra( A )
The zero dimensional subalgebra of the algebra A.

gap> A := QuaternionAlgebra( Rationals );;
gap> B := TrivialSubalgebra( A );
<algebra over Rationals>
gap> Dimension( B );
0

60.6 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 54. Here we give examples demonstrating the use of ideals in algebras. For an introduction into the construction of quotient algebras we refer to Chapter 6.2 of the user’s tutorial.

```
Chapter 60. Algebras

60.7 Categories and Properties of Algebras

1. \texttt{IsFLMLOR( obj )}
   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.

   \begin{verbatim}
   gap> A:= FullMatrixAlgebra( Rationals, 2 );;
   gap> IsFLMLOR ( A );
   true
   \end{verbatim}

2. \texttt{IsFLMLORWithOne( obj )}
   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).

   \begin{verbatim}
   gap> A:= FullMatrixAlgebra( Rationals, 2 );;
   gap> IsFLMLORWithOne ( A );
   true
   \end{verbatim}

3. \texttt{IsAlgebra( obj )}
   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.

   \begin{verbatim}
   gap> A:= MatAlgebra( Rationals, 3 );;
   gap> IsAlgebra( A );
   true
   \end{verbatim}

4. \texttt{IsAlgebraWithOne( obj )}
   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.

   \begin{verbatim}
   gap> A:= MatAlgebra( Rationals, 3 );;
   gap> IsAlgebraWithOne( A );
   true
   \end{verbatim}

5. \texttt{IsLieAlgebra( A )}
   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).
Section 8. Attributes and Operations for Algebras

```gap
gap> A := FullMatrixLieAlgebra( Rationals, 3 );;
gap> IsLieAlgebra( A );
true
```

6 ▶ IsSimpleAlgebra( A )

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 0.

```gap
gap> A := FullMatrixLieAlgebra( Rationals, 3 );;
gap> IsSimpleAlgebra( A );
false
gap> A := MatAlgebra( Rationals, 3 );;
gap> IsSimpleAlgebra( A );
true
```

7 ▶ IsFiniteDimensional( matalg )

returns true (always) for a matrix algebra matalg, since matrix algebras are always finite dimensional.

```gap
gap> A := MatAlgebra( Rationals, 3 );;
gap> IsFiniteDimensional( A );
true
```

8 ▶ IsQuaternion( obj ) ▶ IsQuaternionCollection( obj ) ▶ IsQuaternionCollColl( obj )

IsQuaternion is the category of elements in an algebra constructed by QuaternionAlgebra. A collection of quaternions lies in the category IsQuaternionCollection. Finally, a collection of quaternion collections (e.g., a matrix) lies in the category IsQuaternionCollColl.

```gap
gap> A :=QuaternionAlgebra( Rationals );;
gap> b := BasisVectors( Basis( A ) );
[ e, i, j, k ]
gap> IsQuaternion( b[1] );
true
true
```

60.8 Attributes and Operations for Algebras

1 ▶ GeneratorsOfAlgebra( A )

returns a list of elements that generate A as an algebra.

```gap
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 ] ] ]
```

2 ▶ GeneratorsOfAlgebraWithOne( A )

returns a list of elements of A that generate A as an algebra with one.
ProductSpace( U, V )

is the vector space \langle u \star 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.

PowerSubalgebraSeries( A )

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.

AdjointBasis( B )

Let x be an element of an algebra A. Then the adjoint map of x is the left multiplication by x. It is a linear map of A. For the basis B of an algebra A, this function returns a particular basis C of the matrix space generated by adA, (the matrix spaces spanned by the matrices of the left multiplication); namely a basis consisting of elements of the form adx_i, where x_i is a basis element of B.

IndicesOfAdjointBasis( B )

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.

AsAlgebra( F, A )

Returns the algebra over F generated by A.
Section 8. Attributes and Operations for Algebras

```gap
gap> V := VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> A := AsAlgebra( Rationals, V );;
<algebra of dimension 1 over Rationals>
```

**AsAlgebraWithOne( F, A )**

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.

```gap
gap> V := VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> A := AsAlgebra( Rationals, V );
gap> A1 := AsAlgebraWithOne( Rationals, A );
<algebra-with-one over Rationals, with 1 generators>
```

**AsSubalgebra( A, B )**

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.

```gap
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>
```

**AsSubalgebraWithOne( A, B )**

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.

```gap
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>
```

**MutableBasisOfClosureUnderAction( F, Agens, from, init, opr, zero, maxdim )**

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.

**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**.)

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
Chapter 60. Algebras

algebra generators of $A$, from denotes the appropriate side(s), init is a list of ideal generators of $I$, and opr is again $\ast$.

(Note that bases of ideals in not necessarily associative algebras can be computed using \texttt{MutableBasisOfIdealInNonassociativeAlgebra}.)

Finally, bases of right $A$-modules also can be computed using \texttt{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.

```gap
gap> A:= QuaternionAlgebra( Rationals );;
gap> g:= GeneratorsOfAlgebra( A );;
gap> B:= MutableBasisOfClosureUnderAction( Rationals, g, "left", [ g{1} ], \ast, Zero(A), 4 );
<mutable basis over Rationals, 4 vectors>
gap> BasisVectors( B );
[ e, i, j, k ]
```

12 \textbf{MutableBasisOfNonassociativeAlgebra}( $F$, Agens, zero, maxdim )

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 must be the value of maxdim.

The difference to \texttt{MutableBasisOfClosureUnderAction} 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.)

```gap
gap> L:= FullMatrixLieAlgebra( Rationals, 4 );;
gap> m1:= Random( L );;
gap> m2:= Random( L );;
gap> B:= MutableBasisOfNonassociativeAlgebra( Rationals, [ m1, m2 ], Zero( L ),
> 16 );
<mutable basis over Rationals, 16 vectors>
```

13 \textbf{MutableBasisOfIdealInNonassociativeAlgebra}( $F$, Vgens, Igens, zero, from, maxdim )

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 \texttt{MutableBasisOfClosureUnderAction}), and maxdim is a known upper bound on the dimension of the ideal; if no finite bound for the dimension is known then infinity must be the value of maxdim.

The difference to \texttt{MutableBasisOfClosureUnderAction} 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.)

```gap
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 ] ] ]
```

14 \textbf{DirectSumOfAlgebras}( $A1$, $A2$ )

\textbf{DirectSumOfAlgebras}( list )

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.

```gap
gap> A:= QuaternionAlgebra( Rationals );;
gap> DirectSumOfAlgebras( [A, A, A] );
<algebra of dimension 12 over Rationals>
```

15 ▶ **FullMatrixAlgebraCentralizer( F, lst )**

Compute the centralizer of the list of matrices in the list `lst` in the full matrix algebra over the ring `F`.

```gap
gap> A:= QuaternionAlgebra( Rationals );;
gap> mats:= List(BasisVectors(Basis( A ) ), x -> AdjointMatrix(Basis(A), x ));;
gap> FullMatrixAlgebraCentralizer( Rationals, mats );
<algebra-with-one of dimension 4 over Rationals>
```

16 ▶ **RadicalOfAlgebra( A )**

is the maximal nilpotent ideal of `A`, where `A` is an associative algebra.

```gap
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOneByGenerators( Rationals, [ m ] );
gap> RadicalOfAlgebra( A );
<algebra of dimension 2 over Rationals>
```

17 ▶ **CentralIdempotentsOfAlgebra( A )**

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`.)

```gap
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= DirectSumOfAlgebras( [A, A] );
gap> CentralIdempotentsOfAlgebra( B );
[ v.9, v.5, v.1 ]
```

18 ▶ **DirectSumDecomposition( L )**

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 Lie algebras (semisimple or not).

```gap
gap> G:= SymmetricGroup( 4 );;
gap> A:= GroupRing( Rationals, G );
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)>],
```
Chapter 60. Algebras

<br/two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
(1 generators)>,
<br/two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
(1 generators)> ]
gap> List( dd, Dimension );
[ 1, 1, 4, 9, 9 ]

19 ▶ LeviMalcevDecomposition( L )

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.

\[
\text{gap> } m := [ [ 1, 2, 0 ], [ 0, 1, 3 ], [ 0, 0, 1 ] ];;
\text{gap> } A := Algebra( \text{Rationals}, [ m ] );;
\text{gap> } \text{LeviMalcevDecomposition}( A );
[ <algebra of dimension 1 over Rationals>,
<algebra of dimension 2 over Rationals> ]
\]

20 ▶ Grading( A )

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 \( A_g \cdot A_h \subseteq A_{g+h} \) 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**: 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 61.8.4, 61.11.2).

\[
\text{gap> } G := \text{SmallGroup}(3^6, 100 );
<\text{pc group of size 729 with 6 generators}>
\text{gap> } L := \text{JenningsLieAlgebra}( G );;
<\text{Lie algebra of dimension 6 over GF(3)>}
\text{gap> } g := \text{Grading}( L );
\text{rec( min_degree := 1, max_degree := 9, source := Integers,}
\text{ hom_components := function( d ) ... end })
\text{gap> } g.\text{hom_components}( 3 );
<\text{vector space over GF(3), with 1 generators}>
\text{gap> } g.\text{hom_components}( 14 );
<\text{vector space over GF(3), with 0 generators}>
\]

60.9 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.
Section 9. Homomorphisms of Algebras

1 ▶ AlgebraGeneralMappingByImages( A, B, gens, imgs )

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 (59.9.2).

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 ] ] ]

2 ▶ AlgebraHomomorphismByImages( A, B, gens, imgs )

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, and one can construct multi-valued mappings with AlgebraGeneralMappingByImages.

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 ] );;

3 ▶ AlgebraHomomorphismByImagesNC( A, B, gens, imgs )

AlgebraHomomorphismByImagesNC is the operation that is called by the function AlgebraHomomorphismByImages. 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 can be used.

For the definitions of the algebras A and B in the next example we refer to the previous example.

[ [ 1, 0 ] ]

4 ▶ AlgebraWithOneGeneralMappingByImages( A, B, gens, imgs )

This function is analogous to 60.9.1; the only difference being that the identity of A is automatically mapped to the identity of B.
Chapter 60. Algebras

5\& AlgebraWithOneHomomorphismByImages( A, B, gens, imgs )

AlgebraWithOneHomomorphismByImages returns the algebra-with-one homomorphism with source $A$ and range $B$ that is defined by mapping the list $\text{gens}$ of generators of $A$ to the list $\text{imgs}$ 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 $\text{gens}$ needs to generate $A$ only as an algebra-with-one.

If $\text{gens}$ 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 \text{AlgebraWithOneHomomorphismByImagesNC}, and one can construct multi-valued mappings with \text{AlgebraWithOneGeneralMappingByImages}.

6\& AlgebraWithOneHomomorphismByImagesNC( A, B, gens, imgs )

AlgebraWithOneHomomorphismByImagesNC is the operation that is called by the function \text{AlgebraWithOneHomomorphismByImages}. Its methods may assume that $\text{gens}$ generates $A$ and that the mapping of $\text{gens}$ to $\text{imgs}$ 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, \text{AlgebraWithOneGeneralMappingByImages} can be used.

7\& NaturalHomomorphismByIdeal( A, I )

is the homomorphism of algebras provided by the natural projection map of $A$ onto the quotient algebra $A/I$. This map can be used to take pre-images in the original algebra from elements in the quotient.
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>

8 ▶ OperationAlgebraHomomorphism( A, B [, opr] )

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.

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.

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 ] ]

9 ▶ IsomorphismFpAlgebra( A )

isomorphism from the algebra \( A \) onto a finitely presented algebra.

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 ...> ] ]

10 ▶ IsomorphismMatrixAlgebra( A )

isomorphism from the algebra \( A \) onto a matrix algebra. Currently this is only implemented for associative algebras with one.

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 ] ]
For a basis $B$ of an algebra $A$, say, $\text{IsomorphismSCAlgebra}$ returns an algebra isomorphism from $A$ to an algebra $S$ given by structure constants (see 60.3), such that the canonical basis of $S$ is the image of $B$.

For an algebra $A$, $\text{IsomorphismSCAlgebra}$ chooses a basis of $A$ and returns the $\text{IsomorphismSCAlgebra}$ value for that basis.

```gap
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)> )
```

$\text{RepresentativeLinearOperation}( A, v, w, opr )$ 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 $\text{fail}$ is returned.

```gap
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
```

### 60.10 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 $\text{IsLeftAlgebraModuleElement}$ or $\text{IsRightModuleElement}$ 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 $\text{ExtRepOfObj}$ and $\text{ObjByExtRep}$. For an element $v$ of the algebra module, $\text{ExtRepOfObj}( v )$ returns the underlying element of the original vector space. Furthermore, if $\text{vec}$ is an element of the original vector space, and $\text{fam}$ the elements family of the corresponding algebra module, then $\text{ObjByExtRep}( \text{fam}, \text{vec} )$ 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 $^\cdot$. If $x$ is an element of an algebra $A$, and $v$ an element of a left $A$-module, then $x^\cdot v$ calculates the result of the action of $x$ on $v$. Similarly, if $v$ is an element of a right $A$-module, then $v^\cdot x$ calculates the action of $x$ on $v$.

$\text{LeftAlgebraModuleByGenerators}( A, op, gens )$ 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.

$\text{RightAlgebraModuleByGenerators}( A, op, gens )$ 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.
Section 10. Representations of Algebras

3 BiAlgebraModuleByGenerators( A, B, opl, opr, gens )

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.

\begin{verbatim}
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)>
\end{verbatim}

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.

4 LeftAlgebraModule( A, op, V )

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.

5 RightAlgebraModule( A, op, V )

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.

6 BiAlgebraModule( A, B, opl, opr, V )

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.

\begin{verbatim}
gap> A:= Rationals"^[3,3];
gap> V:= Rationals^3;
( Rationals^3 )
gap> W:= Rationals^3;;
gap> M:= BiAlgebraModule( A, A, \*, \*, V );
<bi-module over ( Rationals^[3,3] ) (left) and ( Rationals^[3,3] ) (right)>
gap> Dimension( M );
3
\end{verbatim}

7 GeneratorsOfAlgebraModule( M )

A list of elements of M that generate M as an algebra module.
Chapter 60. Algebras

```gap
gap> A := Rationals^[3,3];;
gap> V := LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> GeneratorsOfAlgebraModule( V );
[ [ 1, 0, 0 ] ]
```

8► IsAlgebraModuleElement( obj ) C
8► IsAlgebraModuleElementCollection( obj ) C
8► IsAlgebraModuleElementFamily( fam ) C

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).

9► IsLeftAlgebraModuleElement( obj ) C
9► IsLeftAlgebraModuleElementCollection( obj ) C

Category of left algebra module elements. If an object has IsLeftAlgebraModuleElementCollection, then it is a left-algebra module.

10► IsRightAlgebraModuleElement( obj ) C
10► IsRightAlgebraModuleElementCollection( obj ) C

Category of right algebra module elements. If an object has IsRightAlgebraModuleElementCollection, then it is a right-algebra module.

```gap
gap> A := Rationals^[3,3];;
gap> M := BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );
gap> <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> 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 ]
```

11► LeftActingAlgebra( V ) A

Here V is a left-algebra module; this function returns the algebra that acts from the left on V.

12► RightActingAlgebra( V ) A

Here V is a right-algebra module; this function returns the algebra that acts from the right on V.
Section 10. Representations of Algebras

13 ▶ ActingAlgebra( V )

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).

```
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 ] )
```

14 ▶ IsBasisOfAlgebraModuleElementSpace( B )

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).

```
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
```

15 ▶ MatrixOfAction( B, x )

```
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 ] ]
```

16 ▶ SubAlgebraModule( M, gens [, "basis"] )

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.
Chapter 60. Algebras

```
gap> m1 := NullMat( 2, 2 );; m1[1][1] := 1;;
gap> m2 := NullMat( 2, 2 );; m2[2][2] := 1;;
gap> M := Algebra( Rationals, [ m1, m2 ] );;
<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
```

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 $hom$.

```
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.

```
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.

```
gap> f := AlgebraHomomorphismByImages( A, B, Basis(A), [ m1, m2, m3 ] );
gap> RightModuleByHomomorphismToMatAlg( A, f );
<2-dimensional right-module over <algebra of dimension 3 over Rationals>>
```

returns the $A$-module defined by the left action of $A$ on itself.
Section 10. Representations of Algebras

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

20 ▶ FaithfulModule( A )

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 $p > 0$.)

gap> T := EmptySCTable( 2, 0 );;
gap> A := AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 2 over Rationals>

A is a 2-dimensional algebra where all products are zero.

gap> V := FaithfulModule( A );
<3-dimensional 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 ]

21 ▶ ModuleByRestriction( V, sub )

O

ModuleByRestriction( V, subl, subr )

O

Here V is an algebra module and sub is a subalgebra of the acting algebra of V. This function returns the module that is the restriction of V to sub. So it has the same underlying vector space as V, but the acting algebra is sub. If two subalgebras are given then V is assumed to be a bi-module, and subl a subalgebra of the algebra acting on the left, and subr a subalgebra of the algebra acting on the right.

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 );
<3-dimensional left-module over <algebra over Rationals, with 1 generators>>

22 ▶ NaturalHomomorphismBySubAlgebraModule( V, W )

O

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( v, v ) and pre-images with PreImagesRepresentative( f, u ).
The quotient module can also be formed by entering $V/W$.  


Chapter 60. Algebras

```gap
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
```
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 $\star$. 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 $\star$ 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 $\star$ for LieObjects to be the commutator (see 61.1);

61.1 Lie objects

Let $x$ be a ring element, then $\text{LieObject}(x)$ wraps $x$ up into an object that contains the same data (namely $x$). The multiplication $\star$ for Lie objects is formed by taking the commutator. More exactly, if $l_1$ and $l_2$ are the Lie objects corresponding to the ring elements $r_1$ and $r_2$, then $l_1 \star l_2$ is equal to the Lie object corresponding to $r_1 \cdot r_2 - r_2 \cdot r_1$. 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.

1. $\text{LieObject}( \text{obj} )$

Let $\text{obj}$ be a ring element. Then $\text{LieObject}( \text{obj} )$ is the corresponding Lie object. If $\text{obj}$ lies in the family $F$, then $\text{LieObject}( \text{obj} )$ lies in the family $\text{LieFamily}( F )$ (see 61.1.3).

```
gap> m:= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix};;
gap> lo:= \text{LieObject}( m );
\text{LieObject}( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} )
gap> m*m;
\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
gap> lo*lo;
\text{LieObject}( \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} )
```

2. $\text{IsLieObject}( \text{obj} )$

An object lies in $\text{IsLieObject}$ if and only if it lies in a family constructed by $\text{LieFamily}$.

3. $\text{IsLieObjectCollection}( \text{obj} )$

An object lies in $\text{IsLieObjectCollection}$ if and only if it is a collection of LieObjects.
Chapter 61. Lie Algebras

is a family $F$ in bijection with the family $\text{Fam}$, but with the Lie bracket as infix multiplication. That is, for $x, y$ in $\text{Fam}$, the product of the images in $F$ will be the image of $x \ast y - y \ast x$.

The standard type of objects in a Lie family $F$ is $F\_!.\text{packedType}$.

The bijection from $\text{Fam}$ to $F$ is given by $\text{Embedding}(\text{Fam}, F)$; this bijection respects addition and additive inverses.

If $\text{Fam}$ is a Lie family then $\text{UnderlyingFamily}(\text{Fam})$ is a family $F$ such that $\text{Fam} = \text{LieFamily}(F)$.

61.2 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 60.

For an associative algebra $L$, $\text{LieAlgebra}(L)$ is the Lie algebra isomorphic to $L$ as a vector space but with the Lie bracket as product.

$\text{LieAlgebra}(F, gens)$ 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 $\text{LieAlgebra}$ does not contain the vectors in $gens$. The elements in $gens$ are wrapped up as Lie objects (see 61.1). 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 $\ast$.

If there are three arguments, a division ring $F$ and a list $gens$ and an element zero, then $\text{LieAlgebra}(F, gens, zero)$ 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.

```gap
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>
3 ▶
FreeLieAlgebra( R, rank )
F
FreeLieAlgebra( R, rank, name )
F
FreeLieAlgebra( R, name1, name2, ... )
F
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.

```gap
gap> L:= FreeLieAlgebra( Rationals, "x", "y", "z" );;
<Lie algebra over Rationals, with 3 generators>
```
type must be one of A, B, C, D, E, F, G, H, K, S, W. For the types A to G, n must be a positive integer. The last four 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 other types, S and W, n must be a list of positive integers of any length. 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.

\begin{verbatim}
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)>
\end{verbatim}

## 61.3 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.).

1. **LieCentre(L)**
   - 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`.

\begin{verbatim}
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)>
\end{verbatim}

2. **LieCentralizer(L, S)**
   - 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 $L$.

\begin{verbatim}
gap> b:= BasisVectors( Basis( L ) );;
gap> LieCentralizer( L, Subalgebra( L, [ b[1], b[2] ] ) ); <Lie algebra of dimension 1 over Rationals>
\end{verbatim}

3. **LieNormalizer(L, U)**
   - is the normalizer of the subspace $U$ in the Lie algebra $L$, that is, the set $N_L(U) = \{ x \in L; [x, U] \subset U \}$.

\begin{verbatim}
gap> b:= BasisVectors( Basis( L ) );;
gap> LieNormalizer( L, Subalgebra( L, [ b[1], b[2] ] ) ); <Lie algebra of dimension 8 over Rationals>
\end{verbatim}

4. **LieDerivedSubalgebra(L)**
   - is the (Lie) derived subalgebra of the Lie algebra $L$.
Section 4. Series of Ideals

5 ▶ LieNil Radical( L )

This function calculates the (Lie) nil radical of the Lie algebra L.

In the following two examples we temporarily increase the line length limit from its default value 80 to 81 in order to make the long output expressions fit each into one line.

    gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];
    gap> L:= LieAlgebra( Rationals, mats );
    gap> LieNil Radical( L );
    <two-sided ideal in <Lie algebra of dimension 3 over Rationals>, (dimension 2)>
    gap> SizeScreen([ 80, ]);;

6 ▶ Lie Solvable Radical( L )

Returns the (Lie) solvable radical of the Lie algebra L.

    gap> L:= FullMatrixLieAlgebra( GF( 3 ), 3 );
    gap> LieSolvable Radical( L );
    <two-sided ideal in <Lie algebra of dimension 9 over Rationals>, (dimension 1)>
    gap> SizeScreen([ 80, ]);;

7 ▶ Cartan Subalgebra( L )

A Cartan subalgebra of a Lie algebra L is defined as a nilpotent subalgebra of L equal to its own Lie normalizer in L.

    gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
    gap> CartanSubalgebra( L );
    <Lie algebra of dimension 2 over Rationals>

61.4 Series of Ideals

1 ▶ Lie Derived Series( L )

is the (Lie) derived series of the Lie algebra L.

    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> ]
Chapter 61. Lie Algebras

61.5 Properties of a Lie Algebra

1. IsLieAbelian( L )
   
is true if L is a Lie algebra such that each product of elements in L is zero, and false otherwise.

2. IsLieNilpotent( L )
   
A Lie algebra L is defined to be (Lie) nilpotent when its (Lie) lower central series reaches the trivial subalgebra.

3. IsLieSolvable( L )
   
A Lie algebra L is defined to be (Lie) solvable when its (Lie) derived series reaches the trivial subalgebra.
61.6 Direct Sum Decompositions

In this section we describe two functions that calculate a direct sum decomposition of a Lie algebra; the so-called Levi decomposition and the decomposition into a direct sum of ideals.

1 ▶ LeviMalcevDecomposition( L )

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.

```gap
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)> ]
```

2 ▶ DirectSumDecomposition( L )

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 Lie algebras (semisimple or not).

```gap
gap> L := FullMatrixLieAlgebra( Rationals, 5 );;
gap> DirectSumDecomposition( L );
[ <two-sided ideal in <two-sided ideal in <Lie algebra of dimension 25 over Rationals>,
  (dimension 24)>,
  <Lie algebra of dimension 25 over Rationals>,
  (dimension 24)> ]
```

61.7 Semisimple Lie Algebras and Root Systems

This section contains some functions for dealing with semisimple Lie algebras and their root systems.

1 ▶ SemiSimpleType( L )

Let \( L \) be a semisimple Lie algebra, i.e., a direct sum of simple Lie algebras. Then \texttt{SemiSimpleType} returns the type of \( L \), i.e., a string containing the types of the simple summands of \( L \).

```gap
gap> L := SimpleLieAlgebra( "E", 8, Rationals );;
gap> b := BasisVectors( Basis( L ) );;
<Lie algebra of dimension 102 over Rationals>
gap> lev := LeviMalcevDecomposition( K );
<two-sided ideal in <Lie algebra of dimension 25 over Rationals>,
  (dimension 1)> ]
gap> SemiSimpleType( lev[1] );
"B3 A1"
```

2 ▶ ChevalleyBasis( L )

Here \( L \) must be a semisimple Lie algebra with a split Cartan subalgebra. Then \texttt{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.
 gap> L := SimpleLieAlgebra( "G", 2, Rationals );
 <Lie algebra of dimension 14 over Rationals>
 gap> ChevalleyBasis( L );
  [ v.13, v.14 ] ]

3▶ IsRootSystem( obj )
Category of root systems.

4▶ IsRootSystemFromLieAlgebra( obj )
Category of root systems that come from (semisimple) Lie algebras. They often have
special attributes such as UnderlyingLieAlgebra, PositiveRootVectors, NegativeRootVectors,
CanonicalGenerators.

5▶ RootSystem( L )
RootSystem calculates the root system of the semisimple Lie algebra $L$ with a split
Cartan subalgebra.

 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

6▶ UnderlyingLieAlgebra( R )
Here $R$ is a root system coming from a semisimple Lie algebra $L$. This function returns $L$.

7▶ PositiveRoots( R )
The list of positive roots of the root system $R$.

8▶ NegativeRoots( R )
The list of negative roots of the root system $R$.

9▶ PositiveRootVectors( R )
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 ). The root vector is a non-zero element of the root space
(in $L$) of the corresponding root.

10▶ NegativeRootVectors( R )
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 ). The root vector is a non-zero element of the root space
(in $L$) of the corresponding root.

11▶ SimpleSystem( R )
A list of simple roots of the root system $R$.

12▶ CartanMatrix( R )
The Cartan matrix of the root system $R$, relative to the simple roots in SimpleSystem( R ).
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 $\alpha_i$ are the simple roots of $R$.

Here $R$ must be a root system coming from a semisimple Lie algebra $L$. This function returns 3$\ell$ generators of $L$, $x_1, \ldots, x_\ell, y_1, \ldots, y_\ell, h_1, \ldots, h_\ell$, 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 $-i$-th simple root, and the $h_i$ are elements of the Cartan subalgebra. These elements satisfy the relations $h_i * h_j = 0$, $x_i * y_i = \delta_{ij} h_i$, $h_j * x_i = c_{ij} x_i$, $h_j * y_i = -c_{ij} y_i$, 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 $a$.

The next few sections deal 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). 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 (see below).

A Weyl group is generated by the simple reflections. So GeneratorsOfGroup( $W$ ) 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.

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.
Chapter 61. Lie Algebras

17 ▶ \texttt{WeylGroup( R )}

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 $w \cdot m$.

\begin{verbatim}
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 );
\[
\begin{bmatrix}
1 & 2 \\
3 & -1 \\
2 & 2 \\
1 & -1
\end{bmatrix},
\begin{bmatrix}
2 & 2 \\
3 & 2 \\
4 & -1
\end{bmatrix},
\begin{bmatrix}
1 & -1 \\
3 & 2 \\
4 & -1
\end{bmatrix},
\begin{bmatrix}
2 & -1 \\
3 & -2 \\
4 & 2
\end{bmatrix}
\]
gap> g := GeneratorsOfGroup( W );;
gap> [ 1, 1, 1, 1 ] \ast g[2];
\[1, -1, 1, 2\]
\end{verbatim}

18 ▶ \texttt{ApplySimpleReflection( SC, i, wt )}

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 $wt$.

\begin{verbatim}
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 );
w;
[1, -1, 1, 2]
\end{verbatim}

19 ▶ \texttt{LongestWeylWordPerm( W )}

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 $\alpha_i$ to $-\alpha_{\sigma(i)}$, where $\sigma$ is a permutation of $(1, \ldots, l)$. This function returns that permutation.

\begin{verbatim}
gap> L := SimpleLieAlgebra( "E", 6, Rationals );;
gap> W := WeylGroup( RootSystem( L ) );;
gap> LongestWeylWordPerm( W );
(1,6)(3,5)
\end{verbatim}

20 ▶ \texttt{ConjugateDominantWeight( W, wt )}

Here $W$ is a Weyl group and $wt$ a weight (i.e., a list of integers). This function returns the unique dominant weight conjugate to $wt$ under $W$.

\texttt{ConjugateDominantWeightWithWord( W, wt )} returns a list of two elements. The first of these is the dominant weight conjugate do $wt$. 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.
Section 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 \( \text{ad} x^p = (\text{ad} x)^p, (\alpha x)^p = \alpha^p x^p \) 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 \text{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 \( B \).

61.8 Restricted Lie algebras

Test whether \( L \) is restricted.

```gap
gap> L:= SimpleLieAlgebra( "W", [2], GF(5));;
<Lie algebra of dimension 25 over GF(5)>
gap> IsRestrictedLieAlgebra( L );
false
```

\[ 
\text{WeylOrbitIterator}( W, wt ) 
\]

Returns an iterator for the orbit of the weight \( wt \) under the action of the Weyl group \( W \).

```gap
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 ]
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;
51840  # i.e., the size of the Weyl group of E6.
```
Chapter 61. Lie Algebras

gap> IsRestrictedLieAlgebra( L );
true

2 ▶ PthPowerImages( B )

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.

gap> L := SimpleLieAlgebra( "W", [1], GF(11) );
<Lie algebra of dimension 11 over GF(11)>

3 ▶ PthPowerImage( B, x )

B is a basis of a restricted Lie algebra L. This function calculates for an element x of L the image of x under the p-map.

v.1 + v.11

4 ▶ JenningsLieAlgebra( G )

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 37.16.14). 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 is set.

5 ▶ PCentralLieAlgebra( G )

Here G is a nontrivial p-group. PCentralLieAlgebra( G ) does the same as JenningsLieAlgebra( G ) except that the p-central series is used instead of the Jennings series (see 37.16.13). This function also returns a graded Lie algebra. However, it is not necessarily restricted.
61.9 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 60.8.5 and 60.8.6).

1. \texttt{AdjointMatrix}( B, x )

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 the \( i \)-th basis vector of \( B \) under left multiplication by \( x \).

\[
\begin{bmatrix}
0 & 0 & -2 \\
0 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]

2. \texttt{AdjointAssociativeAlgebra}( L, K )

is the associative matrix algebra (with 1) generated by the matrices of the adjoint representation of the subalgebra \( K \) on the Lie algebra \( L \).

\[
\text{<algebra of dimension 9 over Rationals>}
\]

\[
\text{<algebra of dimension 3 over Rationals>}
\]

3. \texttt{KillingMatrix}( B )

is the matrix of the Killing form \( \kappa \) 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 \( B \).

\[
\begin{bmatrix}
0 & 4 & 0 \\
4 & 0 & 0 \\
0 & 0 & 8
\end{bmatrix}
\]

4. \texttt{KappaPerp}( L, U )

is the orthogonal complement of the subspace \( U \) of the Lie algebra \( L \) with respect to the Killing form \( \kappa \), 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 \( L \).

\[
\text{<vector space of dimension 1 over Rationals>}
\]

5. \texttt{IsNilpotentElement}( L, x )

\( x \) is nilpotent in \( L \) if its adjoint matrix is a nilpotent matrix.

\[
\text{true}
\]

6. \texttt{NonNilpotentElement}( L )

A non-nilpotent element of a Lie algebra \( L \) is an element \( x \) such that \( adx \) 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) \texttt{fail} is returned.
Chapter 61. Lie Algebras

```gap
gap> L := SimpleLieAlgebra( "G", 2, Rationals );;
gap> NonNilpotentElement( L );
v.13
gap> IsNilpotentElement( L, last );
false
```

7▷ FindSl2( L, x )  
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.

```gap
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>
```

61.10 Universal Enveloping Algebras

```gap
UniversalEnvelopingAlgebra( L )  
UniversalEnvelopingAlgebra( L, B )
```
Returns the universal enveloping algebra of the Lie algebra $L$. The elements of this algebra are written on a Poincare-Birkhoff-Witt basis.

In the second form $B$ must be a basis of $L$. If this second argument is given, then an isomorphic copy of the universal enveloping algebra is returned, generated by the images (in the universal enveloping algebra) of the elements of $B$.

```gap
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]
```

61.11 Finitely Presented Lie Algebras

Finitely presented Lie algebras can be constructed from free Lie algebras by using the / constructor, i.e., $FL/[r_1...r_k]$ is the quotient of the free Lie algebra $FL$ by the ideal generated by the elements $r_1...r_k$ 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), which returns a surjective homomorphism. The structure constants Lie algebra can then be accessed by calling Range for this map. Also limited computations with elements of the finitely presented Lie algebra are possible.
Section 11. Finitely Presented Lie Algebras

\[ \text{gap} \] \begin{verbatim}
L := FreeLieAlgebra( Rationals, "s", "t" );
\langle \text{Lie algebra over Rationals, with 2 generators} \rangle
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
gL := GeneratorsOfAlgebra( L );
s := gL[1];
t := gL[2];
K := L/\langle s*(s*t), t*(t*(s*t)), s*(t*(s*t))-t*(s*t) \rangle;
\langle \text{Lie algebra over Rationals, with 2 generators} \rangle
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
h := NiceAlgebraMonomorphism( K );
[ [ (1)*s ], [ (1)*t ] ] \rightarrow [ v.1, v.2 ]
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
U := Range( h );
\langle \text{Lie algebra of dimension 3 over Rationals} \rangle
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
IsLieNilpotent( U );
true
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
gK := GeneratorsOfAlgebra( K );
[ [ (1)*s ], [ (1)*t ] ]
gK[1]*(gK[2]*gK[1]) = Zero( K );
true
\end{verbatim}

\begin{verbatim}
FpLieAlgebraByCartanMatrix( C )
\end{verbatim}

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 \( C \).

\[ \text{gap} \] \begin{verbatim}
C := \{ [ 2, -1 ], [ -3, 2 ] \};
K := FpLieAlgebraByCartanMatrix( C );
\langle \text{Lie algebra over Rationals, with 6 generators} \rangle
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
h := NiceAlgebraMonomorphism( K );
[ [ (1)*x1 ], [ (1)*x2 ], [ (1)*x3 ], [ (1)*x4 ], [ (1)*x5 ], [ (1)*x6 ] ] \rightarrow
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
SemiSimpleType( Range( h ) );
"G2"
\end{verbatim}

\begin{verbatim}
NilpotentQuotientOfFpLieAlgebra( FpL, max )
\end{verbatim}

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 of \( FpL \) 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.

\[ \text{gap} \] \begin{verbatim}
L := FreeLieAlgebra( Rationals, "x", "y" );
g := GeneratorsOfAlgebra( L );
x := g[1];
y := g[2];
(1)*x
(1)*y
rr := [ ( (y*x)*x )*x - 6*(y*x)*y,
3*(((y*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*(x*y))*y))-20*(((x*(x*y))*x*y)) ]
K := L/rr;
\langle \text{Lie algebra over Rationals, with 2 generators} \rangle
\end{verbatim}

\[ \text{gap} \] \begin{verbatim}
h := NilpotentQuotientOfFpLieAlgebra( K, 50, [ 1, 2 ] );
[ [ (1)*x ], [ (1)*y ] ] \rightarrow [ v.1, v.2 ]
\end{verbatim}
61. Modules over Lie Algebras and Their Cohomology

Representations of Lie algebras are dealt with in the same way as representations of ordinary algebras (see 60.10). 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

$c : L \times \cdots \times L \rightarrow V$ ($s$ 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 $1 \leq i_1 < i_2 < \cdots < i_s \leq \dim L$. 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

$[[[1,2], 2], [1, 1], [1, 2, 1/2], [1]]$

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 \( c(x) = v_c \) 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 (see 61.12.3), if \( c \) is a cochain, then its corresponding list is returned by \( \text{ExtRepOfObj}(c) \).

2 \( \triangleright \) IsCochain( \text{obj} ) \hspace{1cm} C

\( \triangleright \) IsCochainCollection( \text{obj} ) \hspace{1cm} C

Categories of cochains and of collections of cochains.

3 \( \triangleright \) Cochain( \( V \), \( s \), \text{obj} ) \hspace{1cm} O

Constructs a \( s \)-cochain given by the data in \text{obj}, with respect to the Lie algebra module \( V \). If \( s \) is non-zero, then \text{obj} must be a list.

\[
gap > \text{L} := \text{SimpleLieAlgebra( "A", 1, Rationals );};
\]
\[
gap > \text{V} := \text{AdjointModule( L );};
\]
\[
<3\text{-dimensional left-module over Lie algebra of dimension 3 over Rationals}>>
\]
\[
gap > \text{c1} := \text{Cochain( V, 2, \{ [ [ 1, 3 ], -1 ] \}, [ ], [ [ 2, 3 ], 1/2 ] \});
\]
\[
<2\text{-cochain}>
\]
\[
gap > \text{ExtRepOfObj}(\text{c1});
\]
\[
[ [ [ 1, 3 ], -1 ] \}, [ ], [ [ 2, 3 ], 1/2 ] \] ]
\]
\[
gap > \text{c2} := \text{Cochain( V, 0, Basis( V )[1] );};
\]
\[
<0\text{-cochain}>
\]
\[
gap > \text{ExtRepOfObj}(\text{c2});
\]
\[
v.1
\]
\[
gap > \text{IsCochain( c2 );} \\
\text{true}
\]

4 \( \triangleright \) CochainSpace( \( V \), \( s \) ) \hspace{1cm} O

Returns the space of all \( s \)-cochains with respect to \( V \).

\[
gap > \text{L} := \text{SimpleLieAlgebra( "A", 1, Rationals );};
\]
\[
gap > \text{V} := \text{AdjointModule( L );};
\]
\[
<\text{vector space of dimension 9 over Rationals}>
\]
\[
gap > \text{BasisVectors( Basis( C ) );};
\]
\[
[ <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, <2\text{-cochain}, ]
\]
\[
gap > \text{ExtRepOfObj}( \text{last}[1] );
\]
\[
[ [ [ 1, 2 ], 1 ] \}, [ ], [ ] ]
\]

5 \( \triangleright \) ValueCochain( \( c \), \( y1 \), \( y2 \), ..., \( ys \) ) \hspace{1cm} F

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 \( y1 \),...,\( ys \).

\[
gap > \text{L} := \text{SimpleLieAlgebra( "A", 1, Rationals );};
\]
\[
gap > \text{V} := \text{AdjointModule( L );};
\]
\[
<\text{vector space of dimension 2 over Rationals}>
\]
\[
gap > \text{BasisVectors( Basis( C ) );};
\]
\[
[ [ [ 1, 2 ], 1 ] \}, [ ], [ ] ]
\]
\[
gap > \text{ValueCochain( c, Basis(L)[2], Basis(L)[1] );}
\]
\[
(-1)*v.1
\]

6 \( \triangleright \) LieCoboundaryOperator( \( c \) ) \hspace{1cm} V

This is a function that takes an \( s \)-cochain, and returns an \( s+1 \)-cochain. The coboundary operator is applied.
Chapter 61. Lie Algebras

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>

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.

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.

\[
\text{DominantWeights(} \, R, \, \text{maxw}\)\]

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 over the Lie algebra with root system \( R \). The \( i \)-th element of the second list is the level of the \( i \)-th dominant weight. (Where level is defined as follows. For a weight \( \mu \) we write \( \mu = \lambda - \sum k_i \alpha_i \), where the \( \alpha_i \) are the simple roots, and \( \lambda \) the highest weight. Then the level of \( \mu \) is \( \sum k_i \).

\[
\text{DominantCharacter(} \, L, \, \text{maxw}\)\]

For a highest weight \( \text{maxw} \) and a semisimple Lie algebra \( L \), this returns the dominant weights of the highest-weight module over \( L \), with highest weight \( \text{maxw} \). 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.
DecomposeTensorProduct( $L$, $w_1$, $w_2$ )

Here $L$ is a semisimple Lie algebra and $w_1$, $w_2$ 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).

DimensionOfHighestWeightModule( $L$, $w$ )

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.

Let $L$ be a semisimple Lie algebra over a field of characteristic 0, and let $R$ be its root system. For a positive root $\alpha$ we let $x_\alpha^{(n)}$ and $y_\alpha^{(n)}$ 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

$$x_\alpha^{(n)} = \frac{x_\alpha^n}{n!}, \quad y_\alpha^{(n)} = \frac{y_\alpha^n}{n!}.$$ 

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 $K = (k_1, \ldots, k_l)$ (where $n_i, m_i, k_i \geq 0$) set

$$x^N = x_{\alpha_1}^{(m_1)} \cdots x_{\alpha_s}^{(m_s)}, \quad y^M = y_{\alpha_1}^{(m_1)} \cdots y_{\alpha_s}^{(m_s)}, \quad h^K = (h_1^{k_1}) \cdots (h_l^{k_l}).$$

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 $\mathbb{Z}$-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.
is the category of elements of an admissible lattice in the universal enveloping algebra of a semisimple Lie algebra \( L \).

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 \( \text{ChevalleyBasis}( L ) \). 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 \( ( h_i/1 ) \), which means \( (h_i^1) \).

In general the printed form \( ( h_i/ k ) \) means \( (h_i^k) \).

Also \( y_{i}^{(m)} \) is printed as \( y_i^{-m} \), which means that entering \( y_i^{m} \) at the GAP prompt results in the output \( m!*y_i^{-m} \).

Products of lattice generators are collected using the following order: first come the \( y_{i}^{(m)} \) (in the same order as the positive roots), then the \( (h_i^k) \), and then the \( x_{i}^{(m)} \) (in the same order as the positive roots).

An \( \text{UEALattice} \) element is represented by a list of the form 
\[
[ m1, c1, m2, c2, \ldots ]
\]
where the \( c1, c2 \) etc. are coefficients, and the \( m1, m2 \) etc. monomials. A monomial is a list of the form \([ \text{ind1}, e1, \text{ind2}, e2, \ldots ]\) where \( \text{ind1}, \text{ind2} \) are indices, and \( e1, e2 \) etc. are exponents. Let \( N \) be the number of positive roots of the underlying Lie algebra. 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 \( \text{NegativeRoots}( R )[\text{ind}] \), where \( R \) is the root system of \( L \). This leads to a factor \( \text{yind1}^{-\text{e1}} \) in the printed form of the monomial (which equals \( z^{\text{e1}}/\text{e1!} \), 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 an index lies between \( 2N+1 \) and \( 2N+\text{rank} \), then it represents an element of the Cartan subalgebra. This is printed as \( ( h_1/1 ) \), meaning \( h_1 \) choose \( e_1 \) (\( h_1, \ldots, h_{\text{rank}} \) are the canonical Cartan generators).

The zero element is represented by the empty list, the identity element by the list \([ [], 1 ]\).

```gap
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> g:=LatticeGeneratorsInUEA( L );
gap> [ 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 ]
```
Section 14. Tensor Products and Exterior and Symmetric Powers

8 ▶ IsWeightRepElement( obj )
▶ IsWeightRepElementCollection( obj )
▶ IsWeightRepElementFamily( fam )

Is a category of vectors, that is used to construct elements of highest-weight modules (by HighestWeightModule).

WeightRepElements are represented by a list of the form [ v1, c1, v2, c2, .... ], where the vi are basis vectors, and the ci coefficients. Furthermore a basis vector v is a weight vector. It is represented by a list of 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) 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.

9 ▶ HighestWeightModule( L, wt )

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 (and in particular they do not lie in the category IsWeightRepElement). 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.

gap> SizeScreen([ 81, ]);;
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

61.14 Tensor Products and Exterior and Symmetric Powers

1 ▶ TensorProductOfAlgebraModules( list )
▶ TensorProductOfAlgebraModules( V, W )

Here the elements of list must be algebra modules. The tensor product is returned as an algebra module.
Chapter 61. Lie Algebras

```gap
L := SimpleLieAlgebra("G", 2, Rationals);
V := HighestWeightModule( L, [ 1, 0 ] );
W := TensorProductOfAlgebraModules( [ V, V ] );
<343-dimensional left-module over <Lie algebra of dimension 14 over Rationals>>

w := Basis(W)[1];
1*(1*v0<x1*v0<x1*v0)
gap> Basis(L)[1]^w;
<0-tensor>
gap> Basis(L)[7]^w;
1*(1*v0<x1*v0<y1*v0)+1*(1*v0<y1*v0<x1*v0)+1*(y1*v0<x1*v0)<1*v0)

2► ExteriorPowerOfAlgebraModule( V, k )
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.

```gap
L := SimpleLieAlgebra("G", 2, Rationals);
V := HighestWeightModule( L, [ 1, 0 ] );
W := ExteriorPowerOfAlgebraModule( V, 3 );
<35-dimensional left-module over <Lie algebra of dimension 14 over Rationals>>

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)

3► SymmetricPowerOfAlgebraModule( V, k )
Here V must be an algebra module. This function returns the k-th symmetric power of V (as an algebra
module).

```gap
L := SimpleLieAlgebra("G", 2, Rationals);
V := HighestWeightModule( L, [ 1, 0 ] );
W := SymmetricPowerOfAlgebraModule( V, 3 );
<84-dimensional left-module over <Lie algebra of dimension 14 over Rationals>>

w := Basis(W)[1];
1*(1*v0.1*v0.1*v0)
gap> Basis(L)[7]^w;
<0-symmetric element>
gap> Basis(L)[7]^w;
3*(1*v0.1*v0.y1*v0)

4► DirectSumOfAlgebraModules( list )
4► DirectSumOfAlgebraModules( V, W )
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 ] )

```gap
L := SimpleLieAlgebra( "C", 3, Rationals );
V := HighestWeightModule( L, [ 1, 1, 0 ] );
<64-dimensional left-module over <Lie algebra of dimension 21 over Rationals>>
W := HighestWeightModule( L, [ 0, 0, 2 ] );
<84-dimensional left-module over <Lie algebra of dimension 21 over Rationals>>
U := DirectSumOfAlgebraModules( V, W );
<148-dimensional left-module over <Lie algebra of dimension 21 over Rationals>>
SizeScreen([ 80, ]);
Currently the GAP library contains only few functions dealing with general finitely presented algebras, so this file is merely a placeholder.

The special case of finitely presented Lie algebras is described in 61.11, and there is also a GAP package fplsa for computing structure constants of finitely presented Lie (super)algebras.
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 64.14), where the magma is a free abelian monoid generated by the indeterminates,
- group rings (see 63.1.5), 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 a free Lie algebra is **not** a magma ring, because of the additional relations given by the Jacobi identity; see 63.4 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 63.3). 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. Consider the following example.

```gap
gap> a:= Algebra( GF(2), [ [ Z(2) ] ] );; Size( a );
2
gap> rm:= FreeMagmaRing( GF(2), a );

emb:= Embedding( a, rm );

z:= Zero( a );; o:= One( a );;

imz:= z ^ emb; IsZero( imz );
false

im1:= ( z + o ) ^ emb;
(Z(2)^0)*[ [ Z(2)^0 ] ]

im2:= z + emb + o + emb;
(Z(2)^0)*[ [ 0*Z(2) ] ]+(Z(2)^0)*[ [ Z(2)^0 ] ]

im1 = im2;
false
```
Section 1. Free Magma Rings

63.1 Free Magma Rings

1. `FreeMagmaRing( R, M )`

   is a free magma ring over the ring \( R \), free on the magma \( M \).

2. `GroupRing( R, G )`

   is the group ring of the group \( G \), over the ring \( R \).

3. `IsFreeMagmaRing( D )`

   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` (see 55.1.11) and `UnderlyingMagma` (see 63.1.6) 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.

```gap
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> 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
```

4. `IsFreeMagmaRingWithOne( obj )`

5. `IsGroupRing( obj )`

   A group ring is a magma ring where the underlying magma is a group.

6. `UnderlyingMagma( RM )`

7. `AugmentationIdeal( RG )`

   is the augmentation ideal of the group ring \( RG \), i.e., the kernel of the trivial representation of \( RG \).
Chapter 63. Magma Rings

63.2 Elements of Free Magma Rings

Elements of families in this category have trivial normalisation, i.e., efficient methods for \( \neq \) and \( \prec \).

In order to treat elements of free magma rings uniformly, also without an external representation, the attributes \texttt{CoefficientsAndMagmaElements} (see 63.2.3) and \texttt{ZeroCoefficient} (see 63.2.4) 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, \texttt{ElementOfMagmaRing} (see 63.2.5) can be used. (Of course one can also embed each magma element into the magma ring, see 63.3, and then form the linear combination, but many unnecessary intermediate elements are created this way.)

\texttt{CoefficientsAndMagmaElements( elm )}

is a list that contains at the odd positions the magma elements, and at the even positions their coefficients in the element \( elm \).

\texttt{ZeroCoefficient( elm )}

For an element \( elm \) of a magma ring (modulo relations) \( RM \), \texttt{ZeroCoefficient} returns the zero element of the coefficient ring \( R \).

\texttt{ElementOfMagmaRing( Fam, zeroeff, coeffs, mgmelms )}

\texttt{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 \( Fam \). \texttt{zeroeff} must be the zero of the coefficient ring containing the \( c_i \).

63.3 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 31.1.10) explicitly whenever one needs for example the magma ring element corresponding to a given magma element. Here is an example.

```gap
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) ]
(-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)
```
Section 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 61.2.3).

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 $\neq$ and $\prec$ 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 62.

1. IsElementOfMagmaRingModuloRelations( obj )
2. IsElementOfMagmaRingModuloRelationsCollection( obj )
3. NormalizedElementOfMagmaRingModuloRelations( F, descr )
4. IsMagmaRingModuloRelations( obj )

This category is used, e.g., for elements of free Lie algebras.

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 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.

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 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 62.
63.5 Magma Rings modulo the Span of a Zero Element

1. IsElementOfMagmaRingModuloSpanOfZeroFamily(Fam)

We need this for the normalization method, which takes a family as first argument.

2. IsMagmaRingModuloSpanOfZero(RM)

3. MagmaRingModuloSpanOfZero(R, M, z)

Let $R$ be a ring, $M$ a magma, and $z$ an element of $M$ with the property that $z \ast m = z$ 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 (see 30.10.3).

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 63.4.

63.6 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 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.

```gap
g := SymmetricGroup( 3 );
h := AlternatingGroup( 3 );
IsSubset( g, h );
true
f := GF(2);
gf := GroupRing( f, g );
<algebra-with-one over GF(2), with 2 generators>
fh := GroupRing( f, h );
<algebra-with-one over GF(2), with 1 generators>
IsSubset( fg, fh );
false
o1 := One( fh ); o2 := One( fg ); o1 = o2;
( Z(2)^0 )\ast ()
( Z(2)^0 )\ast ()
false
emb := Embedding( g, fg );
im := Image( emb, h );
<group of size 3 with 1 generators>
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, FreeAssociativeAlgebra, FreeAlgebraWithOne, and FreeAssociativeAlgebraWithOne 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 64) as well as the default representation `IsMagmaRingObjDefaultRep` 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 .
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, 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 a 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.

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 64.1 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.

64.1 Indeterminates

GAP implements a polynomial ring with countably many indeterminates. These indeterminates can be referred to by positive integers. If only a number $num$ of indeterminates is required they default to $[1..num]$.

It is possible to assign names to indeterminates. These names only provide a means for printing the indeterminates in a nice way, but have not necessary any relations to variable names. Indeterminates that have not been assigned a name will be printed as "$x_{nr}$".

It is possible to assign the same name to different indeterminates (though it is probably not a good idea to do so). Asking twice for an indeterminate with the name $nam$ will produce two different indeterminates!

When asking for indeterminates with certain names, GAP usually will take the first 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.

```gap
Indeterminate( R, [nr] )
Indeterminate( R, [avoid] )
Indeterminate( R, name[, avoid] )
Indeterminate( fam, nr )
```

returns 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 avoid. The third usage returns an indeterminate called name (also avoiding the indeterminates in avoid if given).
Section 1. Indeterminates

```
gap> a:=Indeterminate(GF(3));
x_1
gap> x:=Indeterminate(GF(3),"x");
x
gap> Indeterminate(GF(3),"x")=x;
false

gap> y:=Indeterminate(GF(3),"y");z:=Indeterminate(GF(3),"X");
y
X

gap> Indeterminate(GF(3),3);
y

gap> Indeterminate(GF(3),[y,z]);

x
```

2 ▶ IndeterminateNumberOfUnivariateRationalFunction( rfun )

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 of them is constant. (see also 64.1.5 and 64.19.7).

3 ▶ IndeterminateOfUnivariateRationalFunction( rfun )

returns the indeterminate in which the univariate rational function rfun is expressed. (cf. 64.1.2.)

```
gap> IndeterminateNumberOfUnivariateRationalFunction(z);
4

gap> IndeterminateOfUnivariateRationalFunction(z^5+z);
X
```

4 ▶ IndeterminateName( fam, nr )

A

▶ HasIndeterminateName( fam, nr )

O

▶ SetIndeterminateName( fam, nr, name )

O

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

```
gap> IndeterminateName(FamilyObj(x),3);
"y"

gap> HasIndeterminateName(FamilyObj(x),5);
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.
Chapter 64. Polynomials and Rational Functions

\[ \text{gap> a:=Indeterminate(GF(3),5);} \]
\[ \text{x_5} \]
\[ \text{gap> SetName(a,"ah");} \]
\[ \text{gap> a^5+a;} \]
\[ \text{ah+ah^5} \]

5 \text{\textup{\^{\textup{\textuml}}}} \text{CIUnivPols( upol, upol )}

This function (whose name stands for “CommonIndeterminateOfUnivariatePolynomials”) takes two univariate polynomials as arguments. If both polynomials are given in the same indeterminate number \textit{indnum} (in this case they are “compatible” as univariate polynomials) it returns \textit{indnum}. In all other cases it returns \textit{fail}. \text{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.

64.2 Operations for Rational Functions

The rational functions form a field, therefore all arithmetic operations are applicable to rational functions.

\[ \text{\texttt{f + g}} \]
\[ \text{\texttt{f - g}} \]
\[ \text{\texttt{f * g}} \]
\[ \text{\texttt{f / g}} \]

\[ \text{\texttt{gap> x:=Indeterminate(Rationals,1);;y:=Indeterminate(Rationals,2);;}} \]
\[ \text{\texttt{gap> f:=3+x*y+x^5;;g:=5+x^2*y+x*y^2;;}} \]
\[ \text{\texttt{gap> a:=g/f;}} \]
\[ \text{\texttt{(5+x_1^2*x_2+x_1*x_2^2)/(3+x_1*x_2+x_1^5)}} \]

Note that the quotient \texttt{f/g} of two polynomials might be represented as a rational function again. If \texttt{g} is known to divide \texttt{f} the call \texttt{Quotient(f,g)} (see 54.1.10) should be used instead.

\[ \text{\texttt{f mod g}} \]

For two Laurent polynomials \texttt{f} and \texttt{g}, \texttt{f mod g} is the Euclidean remainder (see 54.6.4) of \texttt{f} modulo \texttt{g}.

At the moment \texttt{GAP} does not contain a proper multivariate Gcd algorithm. Therefore it cannot be 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.

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.

64.3 Comparison of Rational Functions

\[ \text{\texttt{f = g}} \]

Two rational functions \texttt{f} and \texttt{g} are equal if the product \texttt{Numerator(f)*Denominator(g)} equals \texttt{Numerator(g)*Denominator(f)}. 

Section 4. Properties and Attributes of Rational Functions

The ordering of rational functions is defined in several steps. Monomials (products of indeterminates) are sorted first by total-degree and then lexicographically (see 64.16.2). Products of monomials with ring elements ("terms") are compared first by their monomials and then by their coefficients.

Polynomials are compared by comparing the largest terms in turn until they differ.

Rational functions are compared by comparing the polynomial $\text{Numerator}(f) \cdot \text{Denominator}(g)$ with the polynomial $\text{Numerator}(g) \cdot \text{Denominator}(f)$. (As the ordering of monomials used by GAP is invariant under multiplication this is independent of common factors in numerator and denominator.)

For univariate polynomials this reduces to an ordering first by total degree and then lexicographically on the coefficients.

64.4 Properties and Attributes of Rational Functions

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 64.4.2) and its denominator (see 64.4.3)

returns the nominator of the rational function `ratfun`.

As no proper multivariate gcd has been implemented yet, numerators and denominators are not guaranteed to be reduced!
DenominatorOfRationalFunction( ratfun )

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!

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

IsPolynomial( ratfun )

A polynomial is a rational functions 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.

gap> IsPolynomial((x*y+x^2*y^3)/y);
true
gap> IsPolynomial((x*y+x^2)/y);
false

AsPolynomial( poly )

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.

gap> AsPolynomial((x*y+x^-2*y^-3)/y);
x+x^2*y^2

IsUnivariateRationalFunction( ratfun )

A rational function is univariate if its numerator and its denominator are both polynomials in the same
one indeterminate. The attribute IndeterminateNumberOfUnivariateRationalFunction can be used to
obtain the number of this common indeterminate.

CoefficientsOfUnivariateRationalFunction( rfun )

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 · n/d where x is
the variable with the number given by 64.1.2. Numerator and Denominator are guaranteed to be cancelled.

IsUnivariatePolynomial( ratfun )

A univariate polynomial is a polynomial in only one indeterminate.

CoefficientsOfUnivariatePolynomial( pol )

CoefficientsOfUnivariatePolynomial returns the coefficient list of the polynomial pol, sorted in ascending
order.

IsLaurentPolynomial( ratfun )

A Laurent polynomial is a univariate rational function whose denominator is a monomial. Therefore every
univariate polynomial is a Laurent polynomial.

The attribute CoefficientsOfLaurentPolynomial (see 64.12.2) gives a compact representation as Laurent
polynomial.
Section 5. Univariate Polynomials

11  ▶  IsConstantRationalFunction(  \text{ratfun}  )

A constant rational function is a function whose numerator and denominator are polynomials of degree 0.

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 64.18 for details.

12  ▶  IsPrimitivePolynomial(  F,  \text{pol}  )

For a univariate polynomial \text{pol} of degree \(d\) in the indeterminate \(X\), with coefficients in a finite field \(F\) with \(q\) elements, say, \text{IsPrimitivePolynomial} returns \text{true} if

1. \text{pol} divides \(X^q^d - 1\), and
2. for each prime divisor \(p\) of \(q^d - 1\), \text{pol} does not divide \(X^{(q^d-1)/p} - 1\),

and \text{false} otherwise.

13  ▶  SplittingField(  \text{f}  )

returns the smallest field which contains the coefficients of \(f\) and the roots of \(f\).

64.5 Univariate Polynomials

Some of the operations are actually defined on the larger domain of Laurent polynomials (see 64.12). For this section you can simply ignore the word “Laurent” if it occurs in a description.

1  ▶  UnivariatePolynomial(  \text{ring},  \text{cofs}, [, \text{ind}]  )

constructs an univariate polynomial over the ring \text{ring} in the indeterminate \text{ind} with the coefficients given by \text{coefs}.

2  ▶  UnivariatePolynomialByCoefficients(  \text{fam},  \text{cofs}, \text{ind}  )

constructs an univariate polynomial over the coefficients family \text{fam} and in the indeterminate \text{ind} with the coefficients given by \text{coefs}. This function should be used in algorithms to create polynomials as it avoids overhead associated with \text{UnivariatePolynomial}.

3  ▶  DegreeOfLaurentPolynomial(  \text{pol}  )

The degree of a univariate (Laurent) polynomial \text{pol} is the largest exponent \(n\) of a monomial \(x^n\) of \text{pol}.

\[
\text{gap> p:=UnivariatePolynomial(Rationals,[1,2,3,4],1);}
1+2*x+3*x^2+4*x^3
\]
\[
\text{gap> UnivariatePolynomialByCoefficients(FamilyObj(1),[9,2,3,4],73);}
9+2*x_73+3*x_73^2+4*x_73^3
\]
\[
\text{gap> CoefficientsOfUnivariatePolynomial(p); [ 1, 2, 3, 4 ]}
\]
\[
\text{gap> DegreeOfLaurentPolynomial(p); 3}
\]
\[
\text{gap> IndeterminateNumberOfLaurentPolynomial(p); 1}
\]
\[
\text{gap> IndeterminateOfLaurentPolynomial(p); x}
\]

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 \text{Value}, \text{Discriminant}, \text{Derivative}, \text{LeadingCoefficient} and \text{LeadingMonomial}:
Chapter 64. Polynomials and Rational Functions

gap> Value(p,Z(5));
Z(5)^2
gap> LeadingCoefficient(p);
4
gap> Derivative(p);
2+6*x+12*x^2

This function returns a list of all roots of the univariate polynomial \textit{upol} in its default domain. If \textit{field} is given the roots over \textit{field} are taken, if the first parameter is the string "split" the field is taken to be the splitting field of the polynomial.

gap> RootsOfUPol(50-45*x-6*x^2+x^3);
[ 10, 1, -5 ]

The info class for univariate polynomials is \texttt{InfoPoly}.

64.6 Polynomials as Univariate Polynomials in one Indeterminate

returns the degree of the polynomial \textit{pol} in the indeterminate \textit{ind} (respectively indeterminate number \textit{inum}).

gap> f:=x^5+3*x*y+9*y^7+4*y^5*x+3*y+2;
2+3*y+3*x*y+x^5+4*x*y^5+9*y^7
gap> DegreeIndeterminate(f,1);
5
gap> DegreeIndeterminate(f,y);
7

PolynomialCoefficientsOfPolynomial returns the coefficient list (whose entries are polynomials not involving the indeterminate \textit{ind}) describing the polynomial \textit{pol} viewed as a polynomial in \textit{ind}. Instead of \textit{ind} also the indeterminate number \textit{inum} can be given.

gap> PolynomialCoefficientsOfPolynomial(f,2);
[ 2*x^5, 3+3*x, 0, 0, 0, 4*x, 0, 9 ]

returns the leading coefficient (that is the coefficient of the leading monomial, see 64.6.4) of the polynomial \textit{pol}. 

Section 7. Multivariate Polynomials

4 ▶ LeadingMonomial( pol )
returns the leading monomial (with respect to the ordering given by 64.16.2 of the polynomial pol as a list containing indeterminate numbers and exponents.

gap> LeadingCoefficient(f,1);
1
gap> LeadingCoefficient(f,2);
9
gap> LeadingMonomial(f);
[ 2, 7 ]
gap> LeadingCoefficient(f);
9

5 ▶ Derivative( ufun )
▶ Derivative( ratfun, ind )
returns the derivative $u_{pol}'$ of the univariate rational function ufun by its indeterminant. The second version returns the derivative of ratfun by the indeterminate ind (respectively indeterminate number ind) when viewing ratfun as univariate in ind.

gap> Derivative(f,2);
3+3*x+20*x*y^4+63*y^6

6 ▶ Discriminant( upol )
▶ Discriminant( pol, ind )
returns the discriminant disc(upoly) of the univariate polynomial upoly by its indeterminant. The second version returns the discriminant of pol by the indeterminate ind (respectively indeterminate number ind).

gap> Discriminant(f,1);
50000+300000*y+675000*y^2+675000*y^3+253125*y^4+62208*y^5+900000*y^7+4050000*y^8+
~8+6489720*y^9+3037500*y^10+1105920*y^11+6075000*y^12+18225000*y^13+13668750*y^14+
~16+1474560*y^17+19208040*y^21+27337500*y^22+262144*y^25+20503125*y^28

7 ▶ Resultant( pol1, pol2, inum )
▶ Resultant( pol1, pol2, ind )
computes the resultant of the polynomials pol1 and pol2 with respect to the indeterminate ind or indeterminate number inum. 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).

gap> Resultant(x^4+y,y^4+x,1);
y+y^16
gap> Resultant(x^4+y,y^4+x,2);
x+x^16

64.7 Multivariate Polynomials

1 ▶ Value( ratfun, indets, vals [, one] )
▶ Value( upol, values [, one] )
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 indeterminate indets, by vals[i]. 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 \( \text{ratfun} \) are defined. In this situation the identity element of \( R \) may be given by a further argument \( \text{one} \) 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 \( \text{val} \). Again, an optional argument \( \text{one} \) may be given.

\[
\text{gap> Value}(x*\text{y}+\text{y}^7, [x,y], [5,7]);
78167
\]

Note that the default values for \( \text{one} \) 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 \( \text{Value} \) defaults to the one of the coefficient ring, when evaluating Matrices in polynomials always the correct \( \text{one} \) should be given!

\[
\text{gap> OnIndeterminates}(x^7\ast y+x\ast y^4,(1,17)(2,28));
\]

\[
x_{17}\ast x_{28}^4+x_{17}^7\ast x_{28}
\]

\[
\text{gap> Stabilizer(\text{Group}((1,2,3,4),(1,2)),x\ast y,\text{OnIndeterminates});}
\]

\[
\text{Group}([ (1,2), (3,4) ])
\]

### 64.8 Minimal Polynomials

\[
\text{gap> MinimalPolynomial(Rationals,[[2,0],[0,2]]);}
\]

\[
-2+x
\]

### 64.9 Cyclotomic Polynomials

\[
\text{gap> CyclotomicPolynomial(Rationals,5);}
\]

\[
1+x+x^2+x^3+x^4
\]

### 64.10 Polynomial Factorization

At the moment GAP provides only methods to factorize univariate polynomials over finite fields (see Chapter 57) and over subfields of cyclotomic fields (see Chapter 58).

\[
\text{gap> Factors([[R],[upoly],[opt]]})
\]

returns a list of the irreducible factors of the univariate polynomial \( \text{upoly} \) in the polynomial ring \( R \). (That is factors over the \text{CoefficientsRing} of \( R \).)

It is possible to pass a record \( \text{opt} \) 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.

gap> f:= CyclotomicPolynomial( GF(2), 7 );
Z(2)^0+x_1+x_1^2+x_1^3+x_1^4+x_1^5+x_1^6
gap> Factors( f );
[ Z(2)^0+x_1+x_1^3, Z(2)^0+x_1^2+x_1^3 ]
gap> Factors( PolynomialRing( GF(8) ), f );
[ 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+x_1 ]
gap> f:= MinimalPolynomial( Rationals, E(4) );
1+x^2
gap> Factors( f );
[ 1+x^2 ]
gap> Factors( PolynomialRing( Rationals ), f );
[ 1+x^2 ]
gap> Factors( PolynomialRing( CF(4) ), f );
[ -E(4)+x, E(4)+x ]

2 ▶ FactorsSquarefree( pring, upol, opt )
returns a factorization of the squarefree, monic, univariate polynomial upoly 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.

64.11 Polynomials over the Rationals
The following functions are only available to polynomials with rations coefficients:

1 ▶ PrimitivePolynomial( f )
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 con-
tent (the Gcd of the coefficients). The operation returns a list [newpol,coeff] with rational
coeff such that coeff*newpol=f.
The following operations are used by GAP inside the factorization algorithm but might be of interest also
in other contexts.

2 ▶ BombieriNorm( pol )
computes weighted Norm [pol]_2 of pol which is a good measure for factor coefficietns (see [BTW93]).

3 ▶ MinimizedBombieriNorm( f )
This function applies linear Tschirnhaus transformations (x ↦ x + i) to the polynomial f, trying to get the
Bombieri norm of f small. It returns a list [new_polynomial, i_of_transformation].

4 ▶ HenselBound( pol, [minpol, den] )
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.

5 ▶ OneFactorBound( pol )
returns the coefficient bound for a single factor of the rational polynomial pol.
64.12 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).

1. \texttt{LaurentPolynomialByCoefficients( fam, cofs, val [, ind] )}
   
   constructs a Laurent polynomial over the coefficients family \( \text{fam} \) and in the indeterminate \( \text{ind} \) (defaulting to 1) with the coefficients given by \( \text{coefs} \) and valuation \( \text{val} \).

2. \texttt{CoefficientsOfLaurentPolynomial( laurent )}
   
   For a Laurent polynomial this function returns a pair \([\text{cof}, \text{val}]\), consisting of the coefficient list (in ascending order) \( \text{cof} \) and the valuation \( \text{val} \) of the Laurent polynomial \( \text{laurent} \).

```gap`
gap> p:=LaurentPolynomialByCoefficients(FamilyObj(1),
> [1,2,3,4,5],-2);
x^-2+2*x^-1+3+4*x+5*x^2

gap> NumeratorOfRationalFunction(p);DenominatorOfRationalFunction(p);
1+2*x+3*x^2+4*x^3+5*x^4
x^2

gap> CoefficientsOfLaurentPolynomial(p*p);
[ [ 1, 4, 10, 20, 35, 44, 46, 40, 25 ], -4 ]
```

3. \texttt{IndeterminateNumberOfLaurentPolynomial( pol )}
   
   Is a synonym for \texttt{IndeterminateNumberOfUnivariateRationalFunction} (see 64.1.2).

4. \texttt{QuotRemLaurpols( left, right, mode )}
   
   takes two Laurent polynomials \( \text{left} \) and \( \text{right} \) and computes their quotient. Depending on the integer variable \( \text{mode} \) it returns:
   
   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 \text{fail} otherwise.

64.13 Univariate Rational Functions

1. \texttt{UnivariateRationalFunctionByCoefficients( fam, ncof, dcof, val[, ind] )}
   
   constructs a univariate rational function over the coefficients family \( \text{fam} \) and in the indeterminate \( \text{ind} \) (defaulting to 1) with numerator and denominator coefficients given by \( \text{ncof} \) and \( \text{dcof} \) and valuation \( \text{val} \).
64.14 Polynomial Rings

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.

\[
\begin{align*}
\text{PolynomialRing}( & \text{ring, rank, [avoid]} ) & \text{O} \\
\text{PolynomialRing}( & \text{ring, names, [avoid]} ) & \text{O} \\
\text{PolynomialRing}( & \text{ring, indets} ) & \text{O}
\end{align*}
\]

creates a polynomial ring over \( \text{ring} \). If a positive integer \( \text{rank} \) is given, this creates the polynomial ring in \( \text{rank} \) indeterminates (differing from the indeterminates contained in \( \text{avoid} \) if given). The second usage takes a list \( \text{names} \) of strings and returns a polynomial ring in indeterminates labelled by \( \text{names} \). In the third use, a list of positive integers \( \text{indets} \) is given. This creates the polynomial ring in the indeterminates labelled by \( \text{indets} \).

\[
\begin{align*}
\text{IndeterminatesOfPolynomialRing}( & \text{pring} ) & \text{A} \\
\text{CoefficientsRing}( & \text{pring} ) & \text{A}
\end{align*}
\]

returns a list of the indeterminates of the polynomial ring \( \text{pring} \) and returns the ring of coefficients of the polynomial ring \( \text{pring} \), that is the ring over which \( \text{pring} \) was defined.

\[
\begin{align*}
gap> r:=\text{PolynomialRing}(\text{GF}(7)); \\
\text{PolynomialRing}(..., [x_1]) \\
gap> r:=\text{PolynomialRing}(\text{GF}(7),3); \\
\text{PolynomialRing}(..., [x_1, x_2, x_3]) \\
gap> \text{IndeterminatesOfPolynomialRing}(r); \\
[x_1, x_2, x_3] \\
gap> r2:=\text{PolynomialRing}(\text{GF}(7),[5,7,12]); \\
\text{PolynomialRing}(..., [x_5, x_7, x_12]) \\
gap> \text{CoefficientsRing}(r); \\
\text{GF}(7) \\
gap> r:=\text{PolynomialRing}(\text{GF}(7),3); \\
\text{PolynomialRing}(..., [x_1, x_2, x_3]) \\
gap> r2:=\text{PolynomialRing}(\text{GF}(7),3,\text{IndeterminatesOfPolynomialRing}(r)); \\
\text{PolynomialRing}(..., [x_4, x_5, x_6]) \\
gap> r:=\text{PolynomialRing}(\text{GF}(7),["x","y","z","z2"]); \\
\text{PolynomialRing}(..., [x, y, z, z2])
\end{align*}
\]

As GAP objects polynomials 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.)

\[
\begin{align*}
\text{IsPolynomialRing}( & \text{pring} ) & \text{C} \\
\text{IsFiniteFieldPolynomialRing}( & \text{pring} ) & \text{C} \\
\text{IsAbelianNumberFieldPolynomialRing}( & \text{pring} ) & \text{C}
\end{align*}
\]

is the category of polynomial rings is the category of polynomial rings over a finite field (see Chapter 57). is the category of polynomial rings over a field of cyclotomics (see the chapters 18 and 58).
7. IsRationalsPolynomialRing(pring)

is the category of polynomial rings over the rationals (see Chapter 16).

gap> IsPolynomialRing(r);
true
gap> IsFiniteFieldPolynomialRing(r);
true
gap> IsRationalsPolynomialRing(r);
false

64.15 Univariate Polynomial Rings

UnivariatePolynomialRing(R[, nr]) O
UnivariatePolynomialRing(R[, avoid]) O
UnivariatePolynomialRing(R, name[, avoid]) O
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. The function will return a ring in an indeterminate that is guaranteed to be different from all the indeterminates in avoid. The third usage returns a ring in an indeterminate called name (also avoiding the indeterminates in avoid if given).

IsUnivariatePolynomialRing(pring) C

is the category of polynomial rings with one indeterminate.

gap> r := UnivariatePolynomialRing(Rationals,"x");
PolynomialRing(..., [ x ])
gap> r2 := PolynomialRing(Rationals, [ "q" ]);
PolynomialRing(..., [ q ])
gap> IsUnivariatePolynomialRing(r);
true
gap> IsUnivariatePolynomialRing(r2);
true

64.16 Polynomial Reduction

The functions described in this section and their syntax might be subject to change in future releases.

PolynomialReduction(poly, gens, order) F

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.

The polynomial reduction requires an ordering on monomials. This must be a function that takes two monomials without coefficients (in the form of lists as described in 64.19, for example [3,1,4,1,5,9]) and returns whether one is smaller than the other.

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.

Currently two admissible orderings are available:
Section 17. Rational Function Families

2▶ MonomialTotalDegreeLess((a, b))

implements comparison of monomial by the total degree order. This is the order GAP naturally puts on monomials. 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 64.19 for details on the expanded form of monomials.

3▶ MonomialRevLexicoLess((a, b))

implements comparison of monomials by the position/lexicographic order. The function takes two monomials \( a \) and \( b \) in expanded form and returns whether the first is smaller than the second.

See section 64.19 for details on the expanded form of monomials.

4▶ LeadingMonomialPosExtRep(fam, ext, order)

This function takes an external representation \( ext \) of a polynomial in family \( fam \) and returns the position of the leading monomial in \( ext \) with respect to the monomial order implemented by the function \( order \).

See section 64.19 for details on the external representation.

At the moment GAP does not contain any facilities to compute Gröbner bases.

64.17 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”.

1▶ RationalFunctionsFamily(fam)

creates a family containing rational functions with coefficients in \( fam \). This family \( fam \) must be a UFD, that is to say, there are no zero divisors, the family must have a one, be commutative and the factorisation of an elements into irreducible elements of the family must be unique (up to units and order). All elements of the RationalFunctionsFamily are rational functions (see 64.4.1).

2▶ IsRationalFunctionsFamily(obj)

Is the category of a family of rational functions.

```gap
gap> fam:=RationalFunctionsFamily(FamilyObj(1));
NewFamily( "RationalFunctionsFamily(...)", [ 618, 620 ],
[ 82, 85, 89, 93, 97, 100, 103, 107, 111, 618, 620 ] )
```

3▶ CoefficientsFamily(rffam)

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.

```gap
gap> 1=Indeterminate(Rationals)^0;
false
```
64.18 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 64.4 will find this out.

Each representation is associated one or several “defining attributes” that give an “external” representation (see 64.19) 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.

64.19 The Defining Attributes of Rational Functions

In general, rational functions are given in terms of monomials. They are represented by lists, using numbers (see 64.1) 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 \([inum, exp, inum, exp, \ldots]\) 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 \(x^5z^8\) is \([1,5,3,8]\).

The representation of a polynomials is a list of the form \([mon, coeff, mon, coeff, \ldots]\) 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 (implemented by the function MonomialTotalDegreeLess). 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.)
Section 20. Creation of Rational Functions

1. \texttt{IsRationalFunctionDefaultRep( obj )}

   is the default representation of rational functions. A rational function in this representation is defined by the attributes \texttt{ExtRepNumeratorRatFun} and \texttt{ExtRepDenominatorRatFun} where \texttt{ExtRepNumeratorRatFun} and \texttt{ExtRepDenominatorRatFun} are both external representations of a polynomial.

2. \texttt{ExtRepNumeratorRatFun( ratfun )}

   returns the external representation of the numerator polynomial of the rational function \texttt{ratfun}. Numerator and Denominator are not guaranteed to be cancelled against each other.

3. \texttt{ExtRepDenominatorRatFun( ratfun )}

   returns the external representation of the denominator polynomial of the rational function \texttt{ratfun}. Numerator and Denominator are not guaranteed to be cancelled against each other.

4. \texttt{ZeroCoefficientRatFun( ratfun )}

   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.

5. \texttt{IsPolynomialDefaultRep( obj )}

   is the default representation of polynomials. A polynomial in this representation is defined by the components and \texttt{ExtRepNumeratorRatFun} where \texttt{ExtRepNumeratorRatFun} is the external representation of the polynomial.

6. \texttt{ExtRepPolynomialRatFun( polynomial )}

   returns the external representation of a polynomial. The difference to \texttt{ExtRepNumeratorRatFun} is that rational functions might know to be a polynomial but can still have a non-vanishing denominator. In this case \texttt{ExtRepPolynomialRatFun} has to call a quotient routine.

7. \texttt{IsLaurentPolynomialDefaultRep( obj )}

   This representation is used for Laurent polynomials and univariate polynomials. It represents a Laurent polynomial via the attributes \texttt{CoefficientsOfLaurentPolynomial} (see 64.12.2) and \texttt{IndeterminateNumberOfLaurentPolynomial} (see 64.12.3).

   The attributes that give a representation of a rational function as a Laurent polynomial are \texttt{CoefficientsOfLaurentPolynomial} (see 64.12.2) and \texttt{IndeterminateNumberOfUnivariateRationalFunction} (see 64.1.2).

   Algorithms should use only the attributes \texttt{ExtRepNumeratorRatFun}, \texttt{ExtRepDenominatorRatFun}, \texttt{ExtRepPolynomialRatFun}, \texttt{CoefficientsOfLaurentPolynomial} and – if the univariate function is not constant – \texttt{IndeterminateNumberOfUnivariateRationalFunction} as the low-level interface to work with a polynomial. They should not refer to the actual representation used.

64.20 Creation of Rational Functions

The operations \texttt{LaurentPolynomialByCoefficients} (see 64.12.1), \texttt{PolynomialByExtRep} and \texttt{RationalFunctionByExtRep} are used to construct objects in the three basic representations for rational functions.

1. \texttt{RationalFunctionByExtRep( rfam, num, den )}

   constructs a rational function (in the representation \texttt{IsRationalFunctionDefaultRep}) in the rational function family \texttt{rfam}, the rational function itself is given by the external representations \texttt{num} and \texttt{den} for numerator and denominator. No test for validity of the arguments is performed and no cancellation takes place.
2. \texttt{PolynomialByExtRep( rfam, extrep )}

constructs a polynomial (in the representation \texttt{IsPolynomialDefaultRep}) in the rational function family \texttt{rfam}, the polynomial itself is given by the external representation \texttt{extrep}. No test for validity of the arguments is performed.

\begin{verbatim}
gap> fam:=RationalFunctionsFamily(FamilyObj(1));;
gap> p:=PolynomialByExtRep(fam,[[1,2],1,[2,1,9,7],3]);
x^2+3*y*x_9^7
gap> q:=p/(p+1);
(x^2+3*y*x_9^7)/(1+x^2+3*y*x_9^7)
gap> ExtRepNumeratorRatFun(q);
[ [ 1, 2 ], 1, [ 2, 1, 9, 7 ], 3 ]
gap> ExtRepDenominatorRatFun(q);
[ [ ], 1, [ 1, 2 ], 1, [ 2, 1, 9, 7 ], 3 ]
\end{verbatim}

3. \texttt{LaurentPolynomialByExtRep( fam, cofs, val, ind )}

creates a Laurent polynomial in the family \texttt{fam} with \texttt{[cofs.val]} as value of \texttt{CoefficientsOfLaurentPolynomial}. 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, \texttt{LaurentPolynomialByCoefficients} (see 64.12.1) should be used.

### 64.21 Arithmetic for External Representations of Polynomials

The following operations are used internally to perform the arithmetic for polynomials in their “external” representation (see 64.19) as lists.

1. \texttt{ZippedSum( z1, z2, czero, funcs )}

computes the sum of two external representations of polynomials \texttt{z1} and \texttt{z2}. \texttt{czero} is the appropriate coefficient zero and \texttt{funcs} a list \texttt{[monomial less, coefficient sum]} containing a monomial comparison and a coefficient addition function. This list can be found in the component \texttt{fam!.zippedSum} of the rational functions family.

2. \texttt{ZippedProduct( z1, z2, czero, funcs )}

computes the product of two external representations of polynomials \texttt{z1} and \texttt{z2}. \texttt{czero} is the appropriate coefficient zero and \texttt{funcs} a list \texttt{[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 \texttt{fam!.zippedProduct} of the rational functions family.

3. \texttt{QuotientPolynomialsExtRep( fam, a, b )}

Let \texttt{a} and \texttt{b} the external representations of two polynomials in the rational functions family \texttt{fam}. This function computes the external representation of the quotient of both polynomials, it returns \texttt{fail} if \texttt{b} does not divide \texttt{a}.

Functions to perform arithmetic with the coefficient lists of Laurent polynomials are described in section 23.3.
Section 22. Cancellation Tests for Rational Functions

64.22 Cancellation Tests for Rational Functions

GAP does not contain a multivariate GCD algorithm. The following operations are used internally to try to keep the denominators as small as possible:

1. `RationalFunctionByExtRepWithCancellation(rfam, num, den)` constructs a rational function as `RationalFunctionByExtRep` does but tries to cancel out common factors of numerator and denominator, calling `TryGcdCancelExtRepPolynomials`.

2. `TryGcdCancelExtRepPolynomials(fam, a, b)`

   Let $f$ and $g$ be two polynomials given by the ext reps $a$ and $b$. This function tries to cancel common factors between $a$ and $b$ and returns a list $[ac, bc]$ of cancelled numerator and denominator ext rep. As there is no proper multivariate GCD cancellation is not guaranteed to be optimal.

3. `HeuristicCancelPolynomials(fam, ext1, ext2)`

   is called by `TryGcdCancelExtRepPol` to perform the actual work. It will return either fail or a new list $[num, den]$ of cancelled numerator and denominator. The cancellation performed is not necessarily optimal.
If we adjoin a root $\alpha$ 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.

### 65.1 Creation of Algebraic Extensions

1. **AlgebraicExtension( $K$, $f$ )** 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** is $K$. The polynomial $p$ is the **DefiningPolynomial** of $L$ and the attribute **RootOfDefiningPolynomial** of $L$ holds a root of $f$ in $L$ (see 56.2.8).

   ```gap
   gap> x:=Indeterminate(Rationals,"x");
   gap> p:=x^4+3*x^2+1;;
   gap> e:=AlgebraicExtension(Rationals,p);
   <field in characteristic 0>
   gap> IsField(e);
   true
   gap> a:=RootOfDefiningPolynomial(e);
   (a)
   ```

2. **IsAlgebraicExtension( $obj$ )** is the category of algebraic extensions of fields.

   ```gap
   gap> IsAlgebraicExtension(e);
   true
   gap> IsAlgebraicExtension(Rationals);
   false
   ```

### 65.2 Elements in Algebraic Extensions

According to Kronecker’s construction, the elements of an algebraic extension considered to be polynomials in the primitive element. The elements of the base field are represented as polynomials of degree 0. **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 0 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.
Section 2. Elements in Algebraic Extensions

```gap
gap> a^3/(a^2+a+1);
(-1/2*a+1/2*a^2-1/2*a^3)
gap> a*(1/a);
!1
```

The external representation of algebraic extension elements are the polynomial coefficients in the primitive element \( a \), the operations `ExtRepOfObj` and `ObjByExtRep` can be used for conversion.

```gap
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]);
(3+19*a-27*a^2+433*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 `One` of the extension which will embed all entries in the extension.

```gap
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 ] ]
```

\textbf{IsAlgebraicElement(} \textit{obj} \textbf{)}

is the category for elements of an algebraic extension.
In this chapter \( p \) is always a (fixed) prime.

The \( p \)-adic numbers \( \mathbb{Q}_p \) are the completion of the rational numbers with respect to the valuation \( \nu_p(p^{\frac{a}{b}}) = v \) 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 approximated 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, \( \frac{1}{2} \), and \( \frac{1}{4} \) are represented as 1(2), 0·1(2), 1·1(2), 0·01(2), 10(2) and 0·0101(2). Approximation means to ignore powers of \( p \), so for example with only 2 digits accuracy \( \frac{1}{4} \) would be approximated as 0·01(2). The important difference to the decimal approximation is that \( p \)-adic approximation is a ring homomorphism on the subrings of \( p \)-adic numbers whose valuation is bounded from below.

In GAP, \( p \)-adic numbers are represented by approximations. A family of (approximated) \( p \)-adic numbers consists of \( p \)-adic numbers with a certain precision and arithmetic with these numbers is done with this precision.

### 66.1 Pure \( p \)-adic Numbers

Pure \( p \)-adic numbers are the \( p \)-adic numbers described so far.

1. \texttt{PurePadicNumberFamily}( \( p \), \textit{precision} )

   returns the family of pure \( p \)-adic numbers over the prime \( p \) with \textit{precision} “digits”.

2. \texttt{PadicNumber}( \textit{fam}, \textit{rat} )

   returns the element of the \( p \)-adic number family \textit{fam} that is used to represent the rational number \textit{rat}.

\( p \)-adic numbers allow the usual operations for fields.

```gap
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)
```

3. \texttt{Valuation}( \textit{obj} )

   The Valuation is the \( p \)-part of the \( p \)-adic number.

4. \texttt{ShiftedPadicNumber}( \textit{padic}, \textit{int} )

   \texttt{ShiftedPadicNumber} takes a \( p \)-adic number \textit{padic} and an integer \textit{shift} and returns the \( p \)-adic number \( c \), that is \( \textit{padic} \cdot p^{\textit{shift}} \). The \textit{shift} is just added to the \( p \)-part.
Section 2. Extensions of the \( p \)-adic Numbers

### 66.2 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 this subfield.

An extended \( p \)-adic field \( L \) is given by two polynomials \( h \) and \( g \) with coeff.-lists \( \text{unram} \) (for the unramified part) and \( \text{ram} \) (for the ramified part). Then \( L \) is isomorphic to \( \mathbb{Q}_p[x,y]/(h(x),g(y)) \).

This function takes the prime number \( p \) and the two coefficient lists \( \text{unram} \) and \( \text{ram} \) for the two polynomials. The polynomial given by the coefficients in \( \text{unram} \) must be a cyclotomic polynomial and the polynomial given by \( \text{ram} \) an Eisenstein-polynomial (or \( 1+x \)). This is not checked by GAP.

Every number out of \( L \) is represented as a coeff.-list for the basis \( \{1, x, x^2, \ldots, y, xy, x^2y, \ldots\} \) of \( L \). The integer \( \text{precision} \) is the number of “digits” that all the coefficients have.

A general comment: the polynomials with which \text{PadicExtensionNumberFamily} is called define an extension of \( \mathbb{Q}_p \). It must be ensured that both polynomials are really irreducible over \( \mathbb{Q}_p \)!

For example \( x^2+x+1 \) is not irreducible over \( \mathbb{Q}_p \). Therefore the “extension” \text{PadicExtensionNumberFamily}(3, 4, [1,1,1], [1,1]) contains non-invertible “pseudo-\( p \)-adic numbers”. Conversely, if an “extension” contains noninvertible elements one of the polynomials was not irreducible.

A word of warning: Depending on the actual representation of quotients, precision may seem to “vanish”. For example in \text{PadicExtensionNumberFamily}(3, 5, [1,1,1], [1,1]) the number \((1.2000, 0.1210)\) 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.

3► IsPadicExtensionNumber( obj ) C
4► IsPadicExtensionNumberFamily( fam ) C
67 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.

67.1 MeatAxe Modules

\begin{verbatim}
GModuleByMats( gens, field )

GModuleByMats( emptygens, dim, field )
\end{verbatim}

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

\begin{description}
\item[generators] A list of matrices which represent a group operation on a finite dimensional row vector space.
\item[dimension] The dimension of the vector space (this is the common length of the row vectors (see 59.8.6)).
\item[field] The field over which the vector space is defined.
\end{description}

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 sub-
modules 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.

67.2 Selecting a Different MeatAxe

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.
67.3 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:

1. \( \text{MTX.Generators( module )} \)
   returns a list of matrix generators of \( \text{module} \).

2. \( \text{MTX.Dimension( module )} \)
   returns the dimension in which the matrices act.

3. \( \text{MTX.Field( module )} \)
   returns the field over which \( \text{module} \) is defined.

67.4 Irreducibility Tests

1. \( \text{MTX.IsIrreducible( module )} \)
   tests whether the module \( \text{module} \) is irreducible (i.e. contains no proper submodules.)

2. \( \text{MTX.IsAbsolutelyIrreducible( module )} \)
   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 \( K \).)

3. \( \text{MTX.DegreeSplittingField( module )} \)
   returns the degree of the splitting field as extension of the prime field.

67.5 Finding Submodules

1. \( \text{MTX.SubmoduleGModule( module, subspace )} \)
   \( \text{subspace} \) should be a subspace of (or a vector in) the underlying vector space of \( \text{module} \) i.e. the full row space of the same dimension and over the same field as \( \text{module} \). A normalized basis of the submodule of \( \text{module} \) generated by \( \text{subspace} \) is returned.

2. \( \text{MTX.ProperSubmoduleBasis( module )} \)
   returns the basis of a proper submodule of \( \text{module} \) and \( \text{fail} \) if no proper submodule exists.

3. \( \text{MTX.BasesSubmodules( module )} \)
   returns a list containing a basis for every submodule.

4. \( \text{MTX.BasesMinimalSubmodules( module )} \)
   returns a list of bases of all minimal submodules.

5. \( \text{MTX.BasesMaximalSubmodules( module )} \)
   returns a list of bases of all maximal submodules.

6. \( \text{MTX.BasisRadical( module )} \)
   returns a basis of the radical of \( \text{module} \).

7. \( \text{MTX.BasisSocle( module )} \)
   returns a basis of the socle of \( \text{module} \).
Section 6. Induced Actions

8 ▶ MTX.BasesMinimalSupermodules(module, sub)
returns a list of bases of all minimal supermodules of the submodule given by the basis sub.

9 ▶ MTX.BasesCompositionSeries(module)
returns a list of bases of submodules in a composition series in ascending order.

10 ▶ MTX.CompositionFactors(module)
returns a list of composition factors of module in ascending order.

11 ▶ MTX.CollectedFactors(module)
returns a list giving all irreducible composition factors with their frequencies.

67.6 Induced Actions

1 ▶ MTX.NormedBasisAndBaseChange(sub)
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).

2 ▶ MTX.InducedActionSubmodule(module, sub)
 ▶ MTX.InducedActionSubmoduleNB(module, sub)
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)

3 ▶ MTX.InducedActionFactorModule(module, sub[, compl])
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)

4 ▶ MTX.InducedActionMatrix(mat, sub)
 ▶ MTX.InducedActionMatrixNB(mat, sub)
 ▶ MTX.InducedActionFactorMatrix(mat, sub[, compl])
work the same way as the above functions for modules, but take as input only a single matrix.

5 ▶ MTX.InducedAction(module, sub[, type])
Computes induced actions on submodules or factormodules 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.
67.7 Module Homomorphisms

1 ▶ MTX.IsEquivalent( module1, module2 )  
   tests two irreducible modules for equivalence.

2 ▶ MTX.Isomorphism( module1, module2 )  
   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 module2 in the standard basis of module1. That is, conjugation of the generators of module2 with $M$ yields the generators of module1.

3 ▶ MTX.Homomorphism( module1, module2, mat )  
   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.

4 ▶ MTX.Homomorphisms( module1, module2 )  
   returns a basis of all homomorphisms from the irreducible module module1 to module2.

5 ▶ MTX.Distinguish( cf, nr )  
   Let cf be the output of MTX.CollectedFactors. This routine tries to find a group algebra element that has nullity zero on all composition factors except number nr.

67.8 The Smash MeatAxe

The standard MeatAxe provided in the GAP library is 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 is assigned by default. For the sake of completeness the remaining sections document more technical functions of this MeatAxe.

1 ▶ SMTX.RandomIrreducibleSubGModule( module )  
   returns the module action on a random irreducible submodule.

2 ▶ SMTX.GoodElementGModule( module )  
   finds an element with minimal possible nullspace dimension if module is known to be irreducible.

3 ▶ SMTX.SortHomGModule( module1, module2, homs )  
   Function to sort the output of Homomorphisms.

4 ▶ SMTX.MinimalSubGModules( module1, module2[, max] )  
   returns (at most max) bases of submodules of module2 which are isomorphic to the irreducible module module1.

5 ▶ SMTX.Setter( string )  
   returns a setter function for the component smashMeataxe.(string).

6 ▶ SMTX.Getter( string )  
   returns a getter function for the component smashMeataxe.(string).
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.

Tests for absolute irreducibility and sets splitting field degree. It neither sets an absolute irreducibility flag, nor tests it.

returns the basis of a minimal submodule of \( \text{module} \) containing the indicated composition factor. It assumes \text{Distinguish} \ has been called already.

creates the direct sum of two matrix lists.

extends the partial basis \( \text{pbasis} \) to a basis of the full space by action of \( \text{module} \). It returns whether it succeeded.

The following getter routines access internal flags. For each routine, the appropriate setter’s name is prefixed with \text{Set}.

Basis of a submodule.

list \([\text{newgens}, \text{coefflist}]\) giving an algebra element used for chopping.

matrix of \text{SMTX.AlgEl}.

minimal polynomial of \text{SMTX.AlgEl}.

uses factor of \text{SMTX.AlgEl}.

nullspace of the matrix evaluated under this factor.

dimension of the nullspace.


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 68.3). For several groups, the table of marks can be restored from the GAP library of tables of marks (see 68.14).

Given the table of marks of \( G \), one can display it (see 68.4) and derive information about \( G \) and its Burnside ring from it (see 68.7, 68.8, 68.9). Moreover, tables of marks in GAP provide an easy access to the classes of subgroups of their underlying groups (see 68.11).

### 68.1 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 \( m_{11} \) in the first column of \( M \) is equal to the index of \( H_1 \) in \( G \) since the trivial subgroup fixes all cosets of \( H_1 \). 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

\[
c_{ij} = \frac{m_{ij}m_{11}}{m_{i1}m_{j1}}.
\]

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].
68.2 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 \( \text{tom} \) in the category \text{IsTableOfMarks} (see 68.6.2), with defining attributes \text{SubsTom} (see 68.7.1) and \text{MarksTom} (see 68.7.1). These two attributes encode the matrix of marks in a compressed form. The \text{SubsTom} value of \( \text{tom} \) 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 \text{MarksTom} value of \( \text{tom} \), which is a list that contains for each entry in \text{SubsTom}( \text{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 68.3.1). 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 68.9), 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 68.9.4) and derived subgroups (see 68.9.2) 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 then \( i \leq j \).

The \text{MarksTom} information can be computed from the values of the attributes \text{NrSubsTom}, \text{LengthsTom}, \text{OrdersTom}, and \text{SubsTom} (see 68.7.2, 68.7.3, 68.7.2). \text{NrSubsTom} stores a list containing for each entry in the \text{SubsTom} value the corresponding number of conjugates that are contained in a subgroup, \text{LengthsTom} a list containing for each conjugacy class of subgroups its length, and \text{OrdersTom} a list containing for each class of subgroups their order. So the \text{MarksTom} value of \( \text{tom} \) may be missing provided that the values of \text{NrSubsTom}, \text{LengthsTom}, and \text{OrdersTom} are stored in \( \text{tom} \).

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 68.9.4, 68.9.2).

If \( \text{tom} \) stores its group \( G \) and a bijection from the rows and columns of the matrix of marks of \( \text{tom} \) 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 \( \text{tom} \) might have been constructed from a generic table of marks (see 68.13), or as table of marks of a factor group from a given table of marks (see 68.9.11). Access to the group \( G \) is provided by the attribute \text{UnderlyingGroup} (see 68.7.7) 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 \( \text{tom} \)– is signalled by the filter \text{IsTableOfMarksWithGens} (see 68.11).

68.3 Constructing Tables of Marks

- \text{TableOfMarks}( G )
- \text{TableOfMarks}( \text{string} )
- \text{TableOfMarks}( \text{matrix} )

In the first form, \( G \) must be a finite group, and \text{TableOfMarks} constructs the table of marks of \( G \). This computation requires the knowledge of the complete subgroup lattice of \( G \) (see 37.19.1). 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 \text{IsTableOfMarksWithGens} value \text{true} (see 68.11).

In the second form, \text{string} must be a string, and \text{TableOfMarks} gets the table of marks with name \text{string} from the GAP library (see 68.14). If no table of marks with this name is contained in the library then \text{fail} is returned.
In the third form, \texttt{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 \texttt{matrix}, \texttt{TableOfMarks} returns a table of marks object (see 68.2) for which \texttt{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 \texttt{IsInternallyConsistent} (see 68.9), and \texttt{fail} is returned if \texttt{matrix} is proved not to describe a matrix of marks; but if \texttt{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.

```gap
gap> tom := TableOfMarks( AlternatingGroup( 5 ) );
TableOfMarks( AlternatingGroup( [ 1 .. 5 ] ) )
gap> TableOfMarks( "J5" );
fail
```  

```gap
gap> a5 := TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> mat :=
> [ [ 60, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
> [ 20, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0, 0, 0 ],
> [ 12, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0, 0, 0 ],
> [ 6, 2, 0, 0, 1, 0, 1, 0, 0, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0, 0, 0 ],
> [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ];
gap> TableOfMarks( mat );
TableOfMarks( <9 classes> )
```  

The following \texttt{TableOfMarks} methods for a group are installed.

- If the group is known to be cyclic then \texttt{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 \texttt{TableOfMarks} computes the table of marks from the lattice (see 68.3.2).
- If the group is known to be solvable then \texttt{TableOfMarks} takes the lattice of subgroups (see 37.19.1) 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 \texttt{TableOfMarksByLattice} (see 68.3.2).
- 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 68.3.3).

Conversely, the lattice of subgroups of a group with known table of marks can be computed using the table of marks, via the function \texttt{LatticeSubgroupsByTom}. This is also installed as a method for \texttt{LatticeSubgroups}.

\begin{verbatim}
2 TableOfMarksByLattice( G )
TableOfMarksByLattice computes the table of marks of the group G from the lattice of subgroups of G. This lattice is computed via LatticeSubgroups (see 37.19.1) if it is not yet stored in G. The function TableOfMarksByLattice is installed as a method for TableOfMarks 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.

3 LatticeSubgroupsByTom( G )
LatticeSubgroupsByTom computes the lattice of subgroups of G from the table of marks of G, using RepresentativeTom (see 68.11.4).
\end{verbatim}
68.4 Printing Tables of Marks

The default ViewObj (see 6.2.3) method for tables of marks prints the string "TableOfMarks", followed by –if known– the identifier (see 68.7.9) 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.

The default PrintObj (see 6.2.3) method for tables of marks does the same as ViewObj, except that the group is Print-ed instead of View-ed.

The default Display (see 6.2.4) 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 (see 6.11.1).

The optional second argument arec of Display can be used to change the default style for displaying a character as shown above. 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.

```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

### 68.5 Sorting Tables of Marks

**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 68.2). 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 (see 68.6.4).

```gap
  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
```

**PermutationTom**

For the table of marks *tom* of the group *G* stored as UnderlyingGroup value of *tom* (see 68.7.7), PermutationTom is a permutation *π* such that the *i*-th conjugacy class of subgroups of *G* belongs to the *π*-*i*-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 (see 68.5.1), 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*. 
Section 7. Attributes of Tables of Marks

gap> MarksTom( tom )[2];
[ 3, 3 ]
gap> MarksTom( sorted )[2];
[ 2, 2 ]
gap> HasPermutationTom( sorted );
true
gap> PermutationTom( sorted );
(2,3)

68.6 Technical Details about Tables of Marks

1 ▶ InfoTom

is the info class for computations concerning tables of marks.

2 ▶ IsTableOfMarks( obj )

Each table of marks belongs to this category.

3 ▶ TableOfMarksFamily

Each table of marks belongs to this family.

4 ▶ TableOfMarksComponents

The list TableOfMarksComponents is used when a table of marks object is created from a record via ConvertToTableOfMarks (see 68.6.5). TableOfMarksComponents contains at position $2i - 1$ a name of an attribute and at position $2i$ the corresponding attribute getter function.

5 ▶ ConvertToTableOfMarks( record )

ConvertToTableOfMarks converts a record with components from TableOfMarksComponents into a table of marks object with the corresponding attributes.

```gap
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> )
```

68.7 Attributes of Tables of Marks

1 ▶ MarksTom( tom )

A

2 ▶ SubsTom( tom )

A

The matrix of marks (see 68.1) 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 68.7.10) 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 68.2). There is no default method for computing the SubsTom value, and the default MarksTom method needs the values of NrSubsTom and OrdersTom (see 68.7.2, 68.7.2).
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 ] ]

2 \triangleright \text{NrSubsTom}( \text{tom} ) \quad A
\triangleright \text{OrdersTom}( \text{tom} ) \quad A

Instead of storing the marks (see 68.7.1) of the table of marks \text{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 \text{NrSubsTom} value in the same way as the marks in the \text{MarksTom} value.

\text{OrdersTom} returns a list that contains at position \(i\) the order of a representative of the \(i\)-th conjugacy class of subgroups of \text{tom}.

One can compute the \text{NrSubsTom} and \text{OrdersTom} values from the \text{MarksTom} value of \text{tom} and vice versa.

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 ]

3 \triangleright \text{LengthsTom}( \text{tom} ) \quad A

For a table of marks \text{tom}, \text{LengthsTom} returns a list of the lengths of the conjugacy classes of subgroups.

gap> LengthsTom( a5 );
[ 1, 15, 10, 5, 6, 10, 6, 5, 1 ]

4 \triangleright \text{ClassTypesTom}( \text{tom} ) \quad A

\text{ClassTypesTom} distinguishes isomorphism types of the classes of subgroups of the table of marks \text{tom} as far as this is possible from the \text{SubsTom} and \text{MarksTom} values of \text{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. \text{ClassTypesTom} returns the list which contains for each class of subgroups its corresponding type.

gap> a6 := TableOfMarks( "A6" );
TableOfMarks( "A6" )
gap> ClassTypesTom( a6 );
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 11, 12, 12, 13, 13, 14, 15, 15, 16 ]

5 \triangleright \text{ClassNamesTom}( \text{tom} ) \quad A

\text{ClassNamesTom} constructs generic names for the conjugacy classes of subgroups of the table of marks \text{tom}. In general, the generic name of a class of non-cyclic subgroups consists of three parts and has the form \("(o)_{tl}\"), 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 68.7.4). 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.

\begin{verbatim}
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)" ]
\end{verbatim}

\subsection*{FusionsTom( tom )}

For a table of marks \textit{tom}, \textbf{FusionsTom} is a list of fusions into other tables of marks. Each fusion is a list of length two, the first entry being the \textbf{Identifier} (see 68.7.9) value of the image table, the second entry being the list of images of the class positions of \textit{tom} in the image table.

This attribute is mainly used for tables of marks in the \textbf{GAP} library (see 68.14).

\begin{verbatim}
gap> fus:= FusionsTom( a6 );;
gap> fus[1];
[ "L3(4)" , [ 1, 2, 3, 3, 14, 5, 9, 7, 15, 24, 26, 27, 32, 33, 50, 57, 55, 63, 73, 77, 90 ] ]
\end{verbatim}

\subsection*{UnderlyingGroup( tom )}

\textbf{UnderlyingGroup} is used to access an underlying group that is stored on the table of marks \textit{tom}. There is no default method to compute an underlying group if it is not stored.

\begin{verbatim}
gap> UnderlyingGroup( a6 );
Group([ (1,2)(3,4), (1,2,4,5)(3,6) ])
\end{verbatim}

\subsection*{IdempotentsTom( tom )}

\textbf{IdempotentsTom} encodes the idempotents of the integral Burnside ring described by the table of marks \textit{tom}. The return value is a list \textit{l} of positive integers such that each row vector describing a primitive idempotent has value 1 at all positions with the same entry in \textit{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 68.9.7).

\textbf{IdempotentsTomInfo} returns a record with components \textit{fixpointvectors} and \textit{primidems}, both bound to lists. The \textit{i}-th entry of the \textit{fixpointvectors} list is the 0−1-vector describing the \textit{i}-th primitive idempotent, and the \textit{i}-th entry of \textit{primidems} is the decomposition of this idempotent in the rows of \textit{tom}.

\begin{verbatim}
gap> IdempotentsTom( a5 );
[ 1, 1, 1, 1, 1, 1, 1, 1, 9 ]
gap> IdempotentsTomInfo( a5 );
rec(
  primidems := [ [ 1, -2, -1, 0, 0, 1, 1, 1 ], [ -1, 2, 1, 0, 0, -1, -1, -1, 1 ] ],
  fixpointvectors := [ [ 1, 1, 1, 1, 1, 1, 1, 1, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 ] ] )
\end{verbatim}

\subsection*{Identifier( tom )}

The identifier of a table of marks \textit{tom} is a string. It is used for printing the table of marks (see 68.4) and in fusions between tables of marks (see 68.7.6).
If \textit{tom} is a table of marks from the GAP library of tables of marks (see 68.14) then it has an identifier, and if \textit{tom} was constructed from a group with \texttt{Name} value (see 12.8.2) then this name is chosen as \texttt{Identifier} value. There is no default method to compute an identifier in all other cases.

\begin{verbatim}
gap> Identifier( a5 );
"A5"
\end{verbatim}

\texttt{MatTom( \textit{tom} )}

\texttt{MatTom} returns the square matrix of marks (see 68.1) of the table of marks \textit{tom} which is stored in a compressed form using the attributes \texttt{MarksTom} and \texttt{SubsTom} (see 68.7.1). This may need substantially more space than the values of \texttt{MarksTom} and \texttt{SubsTom}.

\begin{verbatim}
gap> MatTom( a5 );
\[ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
    [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
    [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
    [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 1, 0, 0 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] \]
\end{verbatim}

\texttt{MoebiusTom( \textit{tom} )}

\texttt{MoebiusTom} computes the Möbius values both of the subgroup lattice of the group \textit{G} with table of marks \textit{tom} and of the poset of conjugacy classes of subgroups of \textit{G}. It returns a record where the component \texttt{mu} contains the Möbius values of the subgroup lattice, and the component \texttt{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 \texttt{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 \texttt{hyp}. For the computation of these values, the position of the derived subgroup of \textit{G} is needed (see 68.9.2). If it is not uniquely determined then the result does not have the components \texttt{ex} and \texttt{hyp}.

\begin{verbatim}
gap> MoebiusTom( a5 );
rec( mu := [ -60, 4, 2,, -1, -1, -1, -1, 1 ],
    nu := [ -1, 2, 1,, -1, -1, -1, -1, 1 ],
    ex := [ -60, 4, 2,, -1, -1, -1, -1, 1 ],
    hyp := [ ] )
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
\end{verbatim}

\texttt{WeightsTom( \textit{tom} )}

\texttt{WeightsTom} extracts the \texttt{weights} from the table of marks \textit{tom}, i.e., the diagonal entries of the matrix of marks (see 68.7.1), indicating the index of a subgroup in its normalizer.

\begin{verbatim}
gap> wt:= WeightsTom( a5 );
[ 60, 2, 2, 3, 2, 1, 1, 1, 1 ]
\end{verbatim}

This information may be used to obtain the numbers of conjugate supergroups from the marks.

\begin{verbatim}
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 ] ]
\end{verbatim}
68.8 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.

1. $IsAbelianTom(tom[, sub])$
2. $IsCyclicTom(tom[, sub])$
3. $IsNilpotentTom(tom[, sub])$
4. $IsPerfectTom(tom[, sub])$
5. $IsSolvableTom(tom[, sub])$

```gap
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
```

68.9 Other Operations for Tables of Marks

1. $IsInternallyConsistent(tom)$

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.

2. $DerivedSubgroupTom(tom, sub)$
3. $DerivedSubgroupsTom(tom)$

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 $DerivedSubgroupsTomPossible$ (see 68.9.3).

$DerivedSubgroupsTom$ is just the list of $DerivedSubgroupTom$ values for all values of $sub$.

4. $DerivedSubgroupsTomPossible(tom)$
5. $DerivedSubgroupsTomUnique(tom)$

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 $DerivedSubgroupTom$ (see 68.9.2).

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 \textit{tom}. The derived subgroups are in general not uniquely determined by the table of marks if no \texttt{UnderlyingGroup} value is stored, so there is no default method for \texttt{DerivedSubgroupsTomUnique}. But in some cases the derived subgroups are explicitly set when the table of marks is constructed. The \texttt{DerivedSubgroupsTomUnique} value is automatically set when the last missing unique value is entered in the \texttt{DerivedSubgroupsTomPossible} list by \texttt{DerivedSubgroupTom}.

\begin{verbatim}
gap> a5:= TableOfMarks( "A5" ); TableOfMarks( "A5" )
gap> DerivedSubgroupsTomPossible( a5 ); [,,1,,,,,,9]
gap> DerivedSubgroupTom( a5, 2 ); 1
\end{verbatim}

\begin{verbatim}
4\textarrow{NormalizerTom( tom, sub )}
\textarrow{NormalizersTom( tom )}
\end{verbatim}

Let \textit{tom} be the table of marks of a group \( G \), say. \texttt{NormalizerTom} tries to find the conjugacy class of the normalizer \( N \) in \( G \) of a subgroup \( U \) in the \texttt{sub}-th class of \textit{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 \textit{tom} knows the subgroup lattice of \( G \) (see \texttt{68.11.3}) then all normalizers are uniquely determined. \texttt{NormalizerTom} should never return an empty list.

\texttt{NormalizersTom} returns the list of positions of the classes of normalizers of subgroups in \textit{tom}. In addition to the criteria for a single class of subgroup used by \texttt{NormalizerTom}, the approximations of normalizers for several classes are used and thus \texttt{NormalizersTom} may return better approximations than \texttt{NormalizerTom}.

\begin{verbatim}
4\textarrow{NormalizerTom( a5, 4 )}
8
\end{verbatim}

\begin{verbatim}
4\textarrow{NormalizerTom( a5, 4 )}
8
\end{verbatim}

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 \( A_4 \).

\begin{verbatim}
5\textarrow{ContainedTom( tom, sub1, sub2 )}
\end{verbatim}

\texttt{ContainedTom} returns the number of subgroups in class \texttt{sub1} of the table of marks \textit{tom} that are contained in one fixed member of the class \texttt{sub2}.

\begin{verbatim}
6\textarrow{ContainedTom( a5, 3, 5 )}; ContainedTom( a5, 3, 8 ); 0
4
\end{verbatim}

The example shows that a subgroup with class number 3 in \( A_5 \) is contained in one fixed member of the class \texttt{sub1}.

\begin{verbatim}
5\textarrow{ContainedTom( a5, 3, 5 )}; ContainedTom( a5, 3, 8 ); 0
4
\end{verbatim}
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 \( p \).

In the second form, CyclicExtensionsTom returns the classes of this equivalence relation. In the third 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. In the first form, the result is the same as in the third form, with second argument the set of prime divisors of the size of the group of tom.

(This information is not used by NormalizerTom (see 68.9.4) although it might give additional restrictions in the search of normalizers.)

\[
gap> \text{CyclicExtensionsTom}(\text{a5, 2 });
[ [ 1, 2, 4 ], [ 3, 6 ], [ 5, 7 ], [ 8 ], [ 9 ] ]
\]

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.

\[
gap> \text{DecomposedFixedPointVector}(\text{a5, [ 16, 4, 1, 0, 1, 1, 1 ] });
[ 0, 0, 0, 0, 1, 1 ]
\]

The vector fix may be any vector of integers. The resulting decomposition, however, will not be integral, in general.

\[
gap> \text{DecomposedFixedPointVector}(\text{a5, [ 0, 0, 0, 1, 1 ] });
[ 2/5, -1, -1/2, 0, 1/2, 1 ]
\]

In the first form EulerianFunctionByTom computes the Eulerian function (see 37.15.3) of the underlying group \( G \) of the table of marks tom, that is, the number of \( n \)-tuples of elements in \( G \) that generate \( G \). In the second form 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 (see 37.15.3).

\[
gap> \text{EulerianFunctionByTom}(\text{a5, 2 });
2280
\]
\[
gap> \text{EulerianFunctionByTom}(\text{a5, 3 });
200160
\]
\[
gap> \text{EulerianFunctionByTom}(\text{a5, 2, 3 });
8
\]

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.
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.

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 $tom$, $\text{FactorGroupTom}$ returns the table of marks of the factor group $G/N$.

11. $\text{MaximalSubgroupsTom}( \text{tom} )$

In the first form $\text{MaximalSubgroupsTom}$ 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 the table of marks $tom$, the second entry being the list of class lengths of these groups. In the second form the same information for the $sub$-th class of subgroups is returned.

12. $\text{MinimalSupergroupsTom}( \text{tom}, \text{sub} )$

For a table of marks $tom$, $\text{MinimalSupergroupsTom}$ 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.

68.10 Standard Generators of Groups

An $s$-tuple of standard generators of a given group $G$ is a vector $(g_1, g_2, \ldots, g_s)$ of elements $g_i \in G$ satisfying certain conditions (depending on the isomorphism type of $G$) such that

1. $(g_1, g_2, \ldots, g_s) = G$ and
2. the vector is unique up to automorphisms of $G$, i.e., for two vectors $(g_1, g_2, \ldots, g_s)$ and $(h_1, h_2, \ldots, h_s)$ of standard generators, the map $g_i \mapsto h_i$ extends to an automorphism of $G$. 
For details about standard generators, see [Wil96].

1 ▶ \texttt{StandardGeneratorsInfo( G )}

When called with the group \( G \), \texttt{StandardGeneratorsInfo} returns a list of records with at least one of the components \texttt{script} and \texttt{description}. Each such record defines \textbf{standard generators} of groups isomorphic to \( G \), the \( i \)-th record is referred to as the \( i \)-th set of standard generators for such groups. The value of \texttt{script} is a dense list of lists, each encoding a command that has one of the following forms.

A \textbf{definition} \([i, n, k]\) or \([i, n]\)

means to search for an element of order \( n \), and to take its \( k \)-th power as candidate for the \( i \)-th standard generator (the default for \( k \) is 1),

a \textbf{relation} \([i_1, k_1, i_2, k_2, \ldots, i_m, k_m, n]\) with \( m > 1 \)

means a check whether the element \( g_{i_1}^{k_1} g_{i_2}^{k_2} \cdots g_{i_m}^{k_m} \) has order \( n \); if \( g_j \) occurs then of course the \( j \)-th generator must have been defined before,

a \textbf{relation} \([[[i_1, i_2, \ldots, i_m]], slp, n]\)

means a check whether the result of the straight line program \( slp \) (see 35.8) applied to the candidates \( g_{i_1}, g_{i_2}, \ldots, g_{i_m} \) has order \( n \), where the candidates \( g_j \) for the \( j \)-th standard generators must have been defined before,

a \textbf{condition} \([[[i_1, k_1, i_2, k_2, \ldots, i_m, k_m]], f, v]\)

means a check whether the \GAP function in the global list \texttt{StandardGeneratorsFunctions} (see 68.10.3) that is followed by the list \( f \) of strings returns the value \( v \) when it is called with \( G \) and \( g_{i_1}^{k_1} g_{i_2}^{k_2} \cdots g_{i_m}^{k_m} \).

Optional components of the returned records are

\textbf{generators}

a string of names of the standard generators,

\textbf{description}

a string describing the \texttt{script} information in human readable form, in terms of the \texttt{generators} value,

\textbf{classnames}

a list of strings, the \( i \)-th entry being the name of the conjugacy class containing the \( i \)-th standard generator, according to the \texttt{ATLAS} character table of the group (see 69.8.10), and

\textbf{ATLAS}

a boolean; \texttt{true} means that the standard generators coincide with those defined in Rob Wilson’s \texttt{ATLAS} of Group Representations (see [Wil]), and \texttt{false} means that this property is not guaranteed.

There is no default method for an arbitrary isomorphism type, since in general the definition of standard generators is not obvious.

The function \texttt{StandardGeneratorsOfGroup} (see 68.10.5) can be used to find standard generators of a given group isomorphic to \( G \).

The \texttt{generators} and \texttt{description} values, if not known, can be computed by \texttt{HumanReadableDefinition} (see 68.10.2).
Chapter 68. Tables of Marks

gap> StandardGeneratorsInfo( TableOfMarks( "L3(3)" ) );
[ rec( generators := "a, b",
description := "|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4",
script := [ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(, , )|" ], 9 ],
[ 1, 1, 2, 1, 1, 2, 1, 2, 1, 4 ] ],
ATLAS := true ) ]

1► HumanReadableDefinition( info )

Let info be a record that is valid as value of StandardGeneratorsInfo (see 68.10.1). HumanReadableDefinition returns a string that describes the definition of standard generators given by the script component of info in human readable form. The names of the generators are taken from the generators component (default names "a", "b" etc. are computed if necessary), and the result is stored in the description component.

ScriptFromString does the converse of HumanReadableDefinition, i.e., it takes a string string as returned by HumanReadableDefinition, and returns a corresponding script list.

If “condition” lines occur in the script (see 68.10.1) then the functions that occur must be contained in StandardGeneratorsFunctions (see 68.10.3).

gap> scr:= ScriptFromString( "|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4" );
[ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(, , )|" ], 9 ],
[ 1, 1, 2, 1, 1, 2, 1, 2, 1, 4 ] ]

gap> info:= rec( script:= scr );
rec( script := [ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(, , )|" ], 9 ],
[ 1, 1, 2, 1, 1, 2, 1, 2, 1, 4 ] ],
generators := "a, b",
description := "|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4" )

2► ScriptFromString( string )

3► StandardGeneratorsFunctions

StandardGeneratorsFunctions is a list of even length. At position $2i-1$, a function of two arguments is stored, which are expected to be a group and a group element. At position $2i$ a list of strings is stored such that first inserting a generator name in all holes and then forming the concatenation yields a string that describes the function at the previous position; this string must contain the generator enclosed in round brackets ( and ).

This list is used by the functions StandardGeneratorsInfo (see 68.10.1), HumanReadableDefinition, and ScriptFromString (see 68.10.2). Note that the lists at even positions must be pairwise different.

gap> StandardGeneratorsFunctions{ [ 1, 2 ] };
[ function( G, g ) ... end, [ "|C(, , )|" ] ]

4► IsStandardGeneratorsOfGroup( info, G, gens )

Let info be a record that is valid as value of StandardGeneratorsInfo (see 68.10.1), G a group, and gens a list of generators for G. In this case, IsStandardGeneratorsOfGroup returns true if gens satisfies the conditions of the script component of info, and false otherwise.

Note that the result true means that gens is a list of standard generators for G only if G has the isomorphism type for which info describes standard generators.
Section 11. Accessing Subgroups via Tables of Marks

5 StandardGeneratorsOfGroup( info, G[, randfunc] )

Let info be a record that is valid as value of StandardGeneratorsInfo (see 68.10.1), and G a group of the isomorphism type for which info describes standard generators. In this case, StandardGeneratorsOfGroup returns a list of standard generators (see Section 68.10) of G.

The optional argument randfunc must be a function that returns an element of G when called with G; the default is PseudoRandom.

In each call to StandardGeneratorsOfGroup, the script component of info is scanned line by line. randfunc is used to find an element of the prescribed order whenever a definition line is met, and for the relation and condition lines in the script list, the current generator candidates are checked; if a condition is not fulfilled, all candidates are thrown away, and the procedure starts again with the first line. When the conditions are fulfilled after processing the last line of the script list, the standard generators are returned.

Note that if G has the wrong isomorphism type then StandardGeneratorsOfGroup returns a list of elements in G that satisfy the conditions of the script component of info if such elements exist, and does not terminate otherwise. In the former case, obviously the returned elements need not be standard generators of G.

gap> a5:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
gap> info:= StandardGeneratorsInfo( TableOfMarks( "A5" ) )[1];
rec( generators := "a, b", description := "|a|=2, |b|=3, |ab|=5",
    script := [ [ 1, 2 ], [ 2, 3 ], [ 1, 1, 2, 1, 5 ] ], ATLAS := true )
gap> IsStandardGeneratorsOfGroup( info, a5, [ (1,3)(2,4), (3,4,5) ] );
true
gap> IsStandardGeneratorsOfGroup( info, a5, [ (1,2,3) ] );
false
gap> s5:= SymmetricGroup( 5 );;
gap> RepresentativeAction( s5, [ (1,3)(2,4), (1,2,3) ] ),
> StandardGeneratorsOfGroup( info, a5 ), OnPairs ) <> fail;
true

68.11 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 value (see 68.7.7). 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 value of tom is true (see 68.11.3). In this case, RepresentativeTom (see 68.11.4) 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 (see 68.11.1), which uses explicit generators. The second, more general, possibility is implemented by the attributes StraightLineProgramsTom (see 68.11.2) and StandardGeneratorsInfo (see 68.11.5). The StraightLineProgramsTom value encodes the generators as straight line programs (see 35.8) 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. 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 68.11.4 for details and an example).
GeneratorsSubgroupsTom( tom )

Let tom be a table of marks with IsTableOfMarksWithGens 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 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 (see 68.3.1) from a group, and there is a method to compute the value for a table of marks that admits RepresentativeTom (see 68.11.4).

StraightLineProgramsTom( tom )

For a table of marks tom with IsTableOfMarksWithGens value true, StraightLineProgramsTom returns a list that contains at position \( i \) either a list of straight line programs or a straight line program (see 35.8), 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 68.14).

IsTableOfMarksWithGens( tom )

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 68.9.2, 68.9.4).

```
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> Display( progs[4][1] );  # first generator of an el. ab group of order 4
# input:
r[1]:=[ 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];
```
Let \( \text{tom} \) be a table of marks with \texttt{IsTableOfMarksWithGens} value \texttt{true} (see 68.11.3), and \( \text{sub} \) a positive integer. \texttt{RepresentativeTom} returns a representative of the \( \text{sub} \)-th conjugacy class of subgroups of \( \text{tom} \).

\texttt{RepresentativeTomByGenerators} and \texttt{RepresentativeTomByGeneratorsNC} return a representative of the \( \text{sub} \)-th conjugacy class of subgroups of \( \text{tom} \), as a subgroup of the group generated by \( \text{gens} \). This means that the standard generators of \( \text{tom} \) are replaced by \( \text{gens} \).

\texttt{RepresentativeTomByGenerators} checks whether mapping the standard generators of \( \text{tom} \) to \( \text{gens} \) extends to a group isomorphism, and returns \texttt{fail} if not. \texttt{RepresentativeTomByGeneratorsNC} omits all checks. So \texttt{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 \texttt{RepresentativeTom} under this isomorphism. The idea behind \texttt{RepresentativeTomByGeneratorsNC}, however, is to avoid the overhead of using isomorphisms when \( \text{gens} \) are known to be standard generators.

\[
\begin{align*}
gap> \text{ RepresentativeTom( } \text{a5, 4 } \text{ );} \\
& \text{Group([ (2,3)(4,5), (2,4)(3,5) ])}
\end{align*}
\]

For a table of marks \( \text{tom} \), a stored value of \texttt{StandardGeneratorsInfo} equals the value of this attribute for the underlying group (see 68.7.7) of \( \text{tom} \), cf. Section 68.10.

In this case, the \texttt{GeneratorsOfGroup} value of the underlying group \( G \) of \( \text{tom} \) is assumed to be in fact a list of standard generators for \( G \); so one should be careful when setting the \texttt{StandardGeneratorsInfo} value by hand.

There is no default method to compute the \texttt{StandardGeneratorsInfo} value of a table of marks if it is not yet stored.

\[
\begin{align*}
gap> \text{ std := StandardGeneratorsInfo( a5 );} \\
& \text{ [ rec( generators := "a, b", description := "|a|=2, |b|=3, |ab|=5",} \\
& \text{ script := [ [ 1, 2 ], [ 2, 3 ], [ 1, 1, 2, 1, 5 ] ], ATLAS := true ) ]} \\
\end{align*}
\]
```gap
repeat
> y := PseudoRandom( g );
> until Order( y ) = 3 and Order( x*y ) = 5;
> RepresentativeTomByGenerators( a5, 4, [ x, y ]);
> Group([ [ Z(2^2), Z(2^2)^2 ], [ Z(2^2)^2, Z(2^2) ] ],
>        [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] ));
> RepresentativeTomByGenerators( a5, 4, [ x, y ] );
Group([ [ [ Z(2^2), Z(2^2)^2 ], [ Z(2^2)^2, Z(2^2) ] ],
       [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] ] ));
> # Show that the new generators are really good.
> iso := GroupGeneralMappingByImages( grp, g,
        GeneratorsOfGroup( grp ), [ x, y ] );
> IsGroupHomomorphism( iso );
true
> IsBijective( iso );
true
```

### 68.12 The Interface between Tables of Marks and Character Tables

#### FusionCharTableTom

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 68.14 and the manual of the GAP Character Table Library).

2. The UnderlyingGroup 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

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 quick, a boolean, indicates whether a unique fusion shall be returned as soon as it has been determined, without further checks. The default value of quick is false.

```gap
> fus := FusionCharTableTom( a5c, a5 );
[ 1, 2, 3, 5, 5 ]
```

#### PermCharsTom

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 68.12.1). 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 68.12.1) 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 70).

```gap
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 ]
```

### 68.13 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.

1. ▶ **TableOfMarksCyclic( $n$ )**

   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 $d$.

2. ▶ **TableOfMarksDihedral( $n$ )**

   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 $d$.

3. ▶ **TableOfMarksFrobenius( $p$, $q$ )**

   TableOfMarksFrobenius computes the table of marks of a Frobenius group of order $pq$, where $p$ is a prime and $q$ divides $p − 1$.

   ```gap
gap> Display( TableOfMarksCyclic( 6 ) );
1: 6
2: 3 3
3: 2 . 2
4: 1 1 1

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
```
8: 2 . 2 . 2 . . 2
9: 2 . . 2 2 . . . 2
10: 1 1 1 1 1 1 1 1 1

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

### 68.14 The Library of Tables of Marks

@The description of the contents of the library of tables of marks is currently missing.@
This chapter describes operations for **character tables of finite groups**.

Operations for **characters** (or, more general, **class functions**) are described in Chapter 70.

For a description of the **GAP** library of character tables, see the separate manual for this library.

### 69.1 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 $\chi$, $\psi$ of $G$ given by

$$ [\chi, \psi] = \frac{1}{|G|} \sum_{g \in G} \chi(g)\psi(g^{-1}) $$

can be written as
\text{Sum( [1..n], i -> SizesConjugacyClasses( t )[i] * chi[i] * ComplexConjugate( psi[i] ) );}

where \( t \) is the character table of \( G \), and \( \chi, \psi \) are the lists of values of \( \chi, \psi \), 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 \text{GAP}.

\text{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 \text{SizesConjugacyClasses( t )} is the list of class lengths of the character table \( t \). 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 \text{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 \text{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 \text{Irr} (see 69.8.2).

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 \text{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 71.

So class functions, power maps, and fusion maps are represented by lists in \text{GAP}. If they are plain lists then they are regarded as class functions etc. of an appropriate character table when they are passed to \text{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 70). 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.

### 69.2 History of Character Theory Stuff in \text{GAP}

This section still has to be written (by Thomas Breuer who promised to do this for \text{GAP} 4.3).

### 69.3 Creating Character Tables

There are in general five different ways to get a character table in \text{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 69.12. 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 2.3 in the manual of the GAP Character Table Library. Several occurrences of 4. are described in 69.17. 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 (see 69.3.1) can be used for the cases 1.–3.

1. \texttt{CharacterTable( \textit{G} )}
2. \texttt{BrauerTable( \textit{ordtbl}, \textit{p} )}
3. \texttt{CharacterTable( \textit{ordtbl}, \textit{p} )}
4. \texttt{CharacterTable( \textit{name}, \textit{param} )}

Called with a group \textit{G}, \texttt{CharacterTable} calls the attribute \texttt{OrdinaryCharacterTable} (see 69.8.4). Called with first argument a group \textit{G} or an ordinary character table \textit{ordtbl}, and second argument a prime \textit{p}, \texttt{CharacterTable} calls the operation \texttt{BrauerTable} (see 69.3.2). Called with a string \textit{name} and perhaps optional parameters \textit{param}, \texttt{CharacterTable} delegates to \texttt{CharacterTableFromLibrary}, which tries to access the GAP Character Table Library (see the manual of this library for an overview of admissible strings \textit{name}).

Probably the most interesting information about the character table is its list of irreducibles, which can be accessed as the value of the attribute \texttt{Irr} (see 69.8.2). If the argument of \texttt{CharacterTable} is a string \textit{name} then the irreducibles are just read from the library file, therefore the returned table stores them already. However, if \texttt{CharacterTable} is called with a group \textit{G} or with an ordinary character table \textit{ordtbl}, the irreducible characters are not computed by \texttt{CharacterTable}. They are only computed when the \texttt{Irr} value is accessed for the first time, for example when \texttt{Display} is called for the table (see 69.11). This means for example that \texttt{CharacterTable} returns its result very quickly, and the first call of \texttt{Display} 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 \texttt{HasIrr} indicates whether the irreducible characters have been computed already.

The reason why \texttt{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 70.12.1) 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 69.12.

If the group \textit{G} is given as an argument, \texttt{CharacterTable} accesses the conjugacy classes of \textit{G} and therefore causes that these classes are computed if they were not yet stored (see 69.6).

1. \texttt{BrauerTable( \textit{ordtbl}, \textit{p} )}
2. \texttt{BrauerTable( \textit{G}, \textit{p} )}
3. \texttt{BrauerTable( \textit{ordtbl}, \textit{p} )}
4. \texttt{ComputedBrauerTables( \textit{ordtbl} )}

Called with an ordinary character table \textit{ordtbl} or a group \textit{G}, \texttt{BrauerTable} returns its \textit{p}-modular character table if GAP can compute this table, and \texttt{fail} otherwise. The \textit{p}-modular table can be computed for \textit{p}-solvable groups (using the Fong-Swan Theorem) and in the case that \textit{ordtbl} is a table from the GAP character table library for which also the \textit{p}-modular table is contained in the table library.

The default method for a group and a prime delegates to \texttt{BrauerTable} for the ordinary character table of this group. The default method for \textit{ordtbl} uses the attribute \texttt{ComputedBrauerTables} for storing the computed Brauer table at position \textit{p}, and calls the operation \texttt{BrauerTableOp} for computing values that are not yet known.
Chapter 69. Character Tables

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 69.7) delegates to `BrauerTable`.

```gap
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> CharacterTable( "A5" );
CharacterTable( "A5" )
gap> CharacterTable( "Symmetric", 4 );
CharacterTable( "Sym(4)" )
gap> ComputedBrauerTables( tbl );
[ , BrauerTable( Sym( [ 1 .. 4 ] ), 2 ) ]
```

3 ▶ `SupportedCharacterTableInfo`

`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` (see 69.3.4).

New attributes and properties can be notified to `SupportedCharacterTableInfo` by creating them with `DeclareAttributeSuppCT` and `DeclarePropertySuppCT` instead of `DeclareAttribute` and `DeclareProperty`.

4 ▶ `ConvertToCharacterTable( record )` F
▶ `ConvertToCharacterTableNC( record )` F

Let `record` be a record. `ConvertToCharacterTable` converts `record` into a component object (see 3.9 in “Programming in GAP”) representing a character table. The values of those components of `record` whose names occur in `SupportedCharacterTableInfo` (see 69.3.3) 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 (see 69.8.9), since this decides about whether the returned character table lies in `IsOrdinaryTable` or in `IsBrauerTable` (see 69.4.1, 69.4.1).

`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 69.11.2.
69.4 Character Table Categories

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 69.8.4).

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 69.6.1) 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 69.18 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 19) or lists of possible values in certain positions, the same may happen also for power maps and class fusions (see 71.3). @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.

```gap
g:= SymmetricGroup( 4 );
tbl:= CharacterTable( g );
modtbl:= tbl mod 2;
CharacterTable( Sym( [ 1 .. 4 ] ) )
BrauerTable( Sym( [ 1 .. 4 ] ), 2 )
IsCharacterTable( tbl );
IsCharacterTable( modtbl );
true
true
IsBrauerTable( modtbl );
IsBrauerTable( tbl );
true
false
IsOrdinaryTable( tbl );
IsOrdinaryTable( modtbl );
true
false
IsCharacterTable( g );
IsCharacterTable( Irr( g ) );
false
```
false

2► InfoCharacterTable

is the info class (see 7.4) for computations with character tables.

3► NearlyCharacterTablesFamily

Every character table like object lies in this family (see 13.1).

69.5 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 (see 69.8.2); in these cases, the Irr value is automatically computed if it was not yet known.
– For a stored class fusion, the image table is denoted by its Identifier value (see 69.8.11); each library table has a unique identifier by which it can be accessed (see 2.2 in the manual for the GAP Character Table Library), tables constructed from groups get an identifier that is unique in the current GAP session.

69.6 The Interface between Character Tables and Groups

For a character table with underlying group (see 69.6.1), the interface between table and group consists of three attribute values, namely the group, the conjugacy classes stored in the table (see ConjugacyClasses below) and the identification of the conjugacy classes of table and group (see IdentificationOfConjugacyClasses 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 use the given table as the character table of $G$ (see 69.6.4).

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.

1► UnderlyingGroup( ordtbl )

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 69.8.4).

2► ConjugacyClasses( tbl )

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 69.18). 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 69.5).

If \( tbl \) was constructed from \( G \) then the conjugacy classes have been stored at the same time when \( G \) was stored. If \( G \) and \( tbl \) were connected later than in the construction of \( tbl \), the recommended way to do this is via \texttt{ConnectGroupAndCharacterTable} (see 69.6.4). So there is no method for \texttt{ConjugacyClasses} that computes the value for \( tbl \) if it is not yet stored.

Brauer tables do not store the \((p\text{-regular})\) conjugacy classes, they access them via the ordinary table (see 69.8.4) if necessary.

\[\text{IdentificationOfConjugacyClasses( } tbl \text{ )} \]

For an ordinary character table \( tbl \) with known underlying group \( G \), \texttt{IdentificationOfConjugacyClasses} returns a list of positive integers that contains at position \( i \) the position of the \( i \)-th conjugacy class of \( tbl \) in the list \texttt{ConjugacyClasses(G)}.

\[
gap> g:=\text{SymmetricGroup}(4);;
\]
\[
gap> \text{repres:= [ (1,2), (1,2,3), (1,2,3,4), (1,2)(3,4), () ];};
\]
\[
gap> \text{ccl:= List( repres, x -> ConjugacyClass( g, x ) );};
\]
\[
gap> \text{SetConjugacyClasses( g, ccl );}
\]
\[
gap> tbl:= \text{CharacterTable( g );}; \quad \# \text{the table stores already the values}
\]
\[
gap> \text{HasConjugacyClasses( tbl ); HasUnderlyingGroup( tbl );}
\]
\[
\text{true true}
\]
\[
gap> \text{UnderlyingGroup( tbl ) = g;}
\]
\[
\text{true}
\]
\[
gap> \text{HasIdentificationOfConjugacyClasses( tbl );}
\]
\[
\text{true}
\]
\[
gap> \text{IdentificationOfConjugacyClasses( tbl );}
\]
\[
[ 5, 1, 2, 3, 4 ]
\]

\[\text{ConnectGroupAndCharacterTable( } G, tbl[, \text{arec}] \text{) } \quad \text{F}
\]
\[\text{ConnectGroupAndCharacterTable( } G, tbl, \text{bijection} \text{) } \quad \text{F}
\]

Let \( G \) be a group and \( tbl \) a character table of (a group isomorphic to) \( G \), such that \( G \) does not store its \texttt{OrdinaryCharacterTable} value and \( tbl \) does not store its \texttt{UnderlyingGroup} value. \texttt{ConnectGroupAndCharacterTable} calls \texttt{CompatibleConjugacyClasses}, trying to identify the classes of \( G \) with the columns of \( tbl \).

If this identification is unique up to automorphisms of \( tbl \) (see 69.8.8) then \( tbl \) is stored as \texttt{CharacterTable} value of \( G \), in \( tbl \) the values of \texttt{UnderlyingGroup}, \texttt{ConjugacyClasses}, and \texttt{IdentificationOfConjugacyClasses} are set, and \texttt{true} is returned.

Otherwise, i.e., if \texttt{GAP} cannot identify the classes of \( G \) up to automorphisms of \( G \), \texttt{false} is returned.

If a record \texttt{arec} is present as third argument, its meaning is the same as for \texttt{CompatibleConjugacyClasses} (see 69.6.5).

If a list \texttt{bijection} is entered as third argument, it is used as value of \texttt{IdentificationOfConjugacyClasses}, relative to \texttt{ConjugacyClasses( } G \text{ )} without further checking, and \texttt{true} is returned.

\[\text{CompatibleConjugacyClasses( } G, \text{ccl}, tbl[, \text{arec}] \text{) } \quad \text{O}
\]
\[\text{CompatibleConjugacyClasses( } tbl[, \text{arec}] \text{) } \quad \text{O}
\]

In the first form, \texttt{ccl} must be a list of the conjugacy classes of the group \( G \), and \( tbl \) the ordinary character table of \( G \). Then \texttt{CompatibleConjugacyClasses} returns a list \( l \) of positive integers that describes an identification of the columns of \( tbl \) with the conjugacy classes \texttt{ccl} in the sense that \( l[i] \) is the position in \texttt{ccl} of the class
corresponding to the $i$-th column of $tbl$, if this identification is unique up to automorphisms of $tbl$ (see 69.8.8); if GAP cannot identify the classes, `fail` is returned.

In the second form, $tbl$ 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 69.8.5),
2. class lengths (see 69.8.7),
3. power maps (see 71.1.1, 71.1.1),
4. symmetries of the table (see 69.8.8).

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 70.7.2) 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 `cl`.

```gap
g:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
tbl:= CharacterTable( "A5" );
CharacterTable( "A5" )
HasUnderlyingGroup( tbl ); HasOrdinaryCharacterTable( g );
false false
CompatibleConjugacyClasses( tbl ); # unique identification [ ]
ConnectGroupAndCharacterTable( g, tbl );
true
HasConjugacyClasses( tbl ); HasUnderlyingGroup( tbl );
true true
IdentificationOfConjugacyClasses( tbl );
[ 1, 2, 3, 4, 5 ]
# Here is an example where the identification is not unique.
CompatibleConjugacyClasses( CharacterTable( "J2" ) );
[ [ 17, 18 ], [ 9, 10 ] ]
```
69.7 Operators for Character Tables

The following infix operators are defined for character tables.

\[ tbl1 \ast tbl2 \]
the direct product of two character tables (see 69.17.1),

\[ tbl \div list \]
the table of the factor group modulo the normal subgroup spanned by the classes in the list list (see 69.17.2),

\[ tbl \mod p \]
the \( p \)-modular Brauer character table corresponding to the ordinary character table tbl (see 69.3.1),

\[ tbl.name \]
the position of the class with name name in tbl (see 69.8.10).

69.8 Attributes and Properties of Character Tables

Several attributes for groups are valid also for character tables. These are on one hand those that have the same meaning for both group and character table, and whose values can be read off or computed, respectively, from the character table, such as Size, IsAbelian, or IsSolvable. On the other hand, there are attributes whose meaning for character tables is different from the meaning for groups, such as ConjugacyClasses.

\[ \text{CharacterDegrees}( G ) \]
\[ \text{CharacterDegrees}( G, p ) \]
\[ \text{CharacterDegrees}( tbl ) \]

In the first two forms, 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 third 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].

\[
\text{gap} > \text{CharacterDegrees}( \text{SymmetricGroup}(4) ); \\
\[
[ [ 1, 2 ], [ 2, 1 ], [ 3, 2 ] ]
\]
\[
\text{gap} > \text{CharacterDegrees}( \text{SymmetricGroup}(4), 2 ) ; \\
\[
[ [ 1, 1 ], [ 2, 1 ] ]
\]
\[
\text{gap} > \text{CharacterDegrees}( \text{CharacterTable}( "A5" ) ); \\
\[
[ [ 1, 1 ], [ 3, 2 ], [ 4, 1 ], [ 5, 1 ] ]
\]
\[
\text{gap} > \text{CharacterDegrees}( \text{CharacterTable}( "A5" ) \mod 2 ) ; \\
\[
[ [ 1, 1 ], [ 2, 2 ], [ 4, 1 ] ]
\]
\]

\[ \text{Irr}( G ) \]
\[ \text{Irr}( G, p ) \]
\[ \text{Irr}( tbl ) \]

Calleed 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 37.14.18) 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 CharacterTable( \( G \) ), which may differ from the ordering of conjugacy classes in \( G \) (see 69.6). As an extreme example, for a character table obtained from sorting the classes of CharacterTable( \( G \) ), the ordering of columns in the Irr matrix respects the sorting of classes (see 69.18), so the irreducibles of such a table will in general not coincide with the irreducibles stored as Irr( \( G \) ) although also the sorted table stores the group \( G \).

In the following example we temporarily increase the line length limit from its default value 80 to 85 in order to get a nicer output format.

```gap
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 ), 2 ), [ 1, 1 ] ),
  Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 2, -1 ] ) ]

gap> SizeScreen([ 85, ]);;

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> SizeScreen([ 80, ]);;

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 ), [ 4, 1, -1, -1 ] ) ]
```

\( \text{LinearCharacters}\) returns the linear (i.e., degree 1) characters in the Irr (see 69.8.2) list of the group \( G \) or the character table \( tbl \), respectively. In the second form, \( \text{LinearCharacters}\) returns the \( p \)-modular linear characters of the group \( G \).

For a character table \( tbl \) with underlying group, \( \text{LinearCharacters}\) may delegate to the group. For a group \( G \), \( \text{LinearCharacters}\) 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 69.8.2).

```gap
gap> LinearCharacters( SymmetricGroup( 4 ) );
[ Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, -1, 1, 1, -1 ] ) ]

gap> LinearCharacters( "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 ), [ 4, 1, -1, -1 ] ) ]
```

\( \text{OrdinaryCharacterTable}\) 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. 69.4).

```gap
gap> OrdinaryCharacterTable( SymmetricGroup( 4 ) );
CharacterTable( Sym( [ 1 .. 4 ] ) )
gap> tbl := CharacterTable( "A5" );
> modtbl := tbl mod 2;
> BrauerTable( "A5", 2 )
gap> OrdinaryCharacterTable( modtbl ) = tbl;
true
```

The following operations for groups are applicable to character tables and mean the same for a character table as for the group; see the chapter about groups for the definition.

AbelianInvariants
CommutatorLength
Exponent
IsAbelian
IsCyclic
IsFinite
IsMonomial
IsNilpotent
IsPerfect
IsSimple
IsSolvable
IsSupersolvable
NrConjugacyClasses
Size

These operations are mainly useful for selecting character tables with certain properties, also for character tables without access to a group.

```gap
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, 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 ]
```
The following three attributes for character tables 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 value of \texttt{ConjugacyClasses}, and this value may differ for a group and its character table (see 69.6). 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.

5\hspace{1em}| 5\hspace{1em} OrdersClassRepresentatives( \textit{tbl} ) \hspace{1em} A

is a list of orders of representatives of conjugacy classes of the character table \textit{tbl}, in the same ordering as the conjugacy classes of \textit{tbl}.

6\hspace{1em}| 6\hspace{1em} SizesCentralizers( \textit{tbl} ) \hspace{1em} A

is a list that stores at position \textit{i} the size of the centralizer of any element in the \textit{i}-th conjugacy class of the character table \textit{tbl}.

7\hspace{1em}| 7\hspace{1em} SizesConjugacyClasses( \textit{tbl} ) \hspace{1em} A

is a list that stores at position \textit{i} the size of the \textit{i}-th conjugacy class of the character table \textit{tbl}.

The following attributes apply only to character tables, not to groups.

8\hspace{1em}| 8\hspace{1em} AutomorphismsOfTable( \textit{tbl} ) \hspace{1em} A

is the permutation group of all column permutations of the character table \textit{tbl} that leave the set of irreducibles and each power map of \textit{tbl} invariant (see also 69.19.2).

9\hspace{1em}| 9\hspace{1em} UnderlyingCharacteristic( \textit{tbl} ) \hspace{1em} A
\hspace{1em} | 9\hspace{1em} UnderlyingCharacteristic( \textit{psi} ) \hspace{1em} A

For an ordinary character table \textit{tbl}, the result is 0, for a \textit{p}-modular Brauer table \textit{tbl}, it is \textit{p}. The underlying characteristic of a class function \textit{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 (see 30.10.1) to denote this, since of course each Brauer character is an element of characteristic zero in the sense of GAP (see Chapter 70).

\[
gap> \text{tbl} := \text{CharacterTable( "A5" );;}
\]
\[
gap> \text{UnderlyingCharacteristic( tbl ));
0
\]
\[
gap> \text{UnderlyingCharacteristic( tbl mod 17 ));
17
\]

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 \( [ \text{"X.1"}, \text{"X.2"}, \ldots ] \), whose length is the number of irreducible characters of \( tbl \).

The position of the class with name \( \text{name} \) in \( tbl \) can be accessed as \( tbl.\text{name} \).

When ClassNames is called with two arguments, the second being the string "ATLAS", the class names returned obey the convention used in Chapter 7, Section 5 of the ATLAS of Finite Groups [CCN+85].

\[
gap> \text{tbl} := \text{CharacterTable( "A5" );;}
\]
\[
gap> \text{ClassNames( tbl ));
\[ \text{"1a", \"2a", \"3a", \"5a", \"5b" } \]
\]
\[
gap> \text{tbl.2a};
2
\]

Identifier 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 "CTn", 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.

\[
gap> \text{Identifier( CharacterTable( "A5" ) );}
\"A5"
\]
\[
gap> \text{tbl} := \text{CharacterTable( Group( () ) );;}
\gap> \text{Identifier( tbl ); Identifer( tbl mod 2 );}
\"CT8"
\"CT8mod2"
\]

InfoText 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.

```gap
gap> Print( InfoText( CharacterTable( "A5" ) ), "\n" );
origin: ATLAS of finite groups, tests: 1.o.r., pow[2,3,5]
```

For a character table \( tbl \), \texttt{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 71.1.1).

```gap
13 \texttt{InverseClasses( \( tbl \))}

For a character table \( tbl \), \texttt{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 71.1.1).

14 \texttt{RealClasses( \( tbl \))}

For a character table \( tbl \), \texttt{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 \( x^{-1} = x^{a(x)-1} \).

```gap
gap> InverseClasses( CharacterTable( "A5" ) );
[ 1, 2, 3, 4, 5 ]
gap> InverseClasses( CharacterTable( "Cyclic", 3 ) );
[ 1, 3, 2 ]
gap> RealClasses( CharacterTable( "A5" ) );
[ 1, 2, 3, 4, 5 ]
gap> RealClasses( CharacterTable( "Cyclic", 3 ) );
[ 1 ]
```

15 \texttt{ClassOrbit( \( tbl \), \( cc \))}

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 \texttt{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 \( tbl \).

16 \texttt{ClassRoots( \( tbl \))}

For a character table \( tbl \), \texttt{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 \texttt{Size( \( tbl \))}.

This information is computed from the power maps of \( tbl \).

```gap
gap> ClassOrbit( CharacterTable( "A5" ), 4 );
[ 4, 5 ]
gap> ClassRoots( CharacterTable( "A5" ) );
[ [ 2, 3, 4, 5 ], [ ], [ ], [ ], [ ] ]
gap> ClassRoots( CharacterTable( "Cyclic", 6 ) );
[ [ 2, 3, 5 ], [ 2 ], [ 2, 6 ], [ 6 ], [ ] ]
```

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 \texttt{ConjugacyClasses( \( tbl \))}– of those classes forming the normal subgroup in question.
section 8. attributes and properties of character tables

17 ▶ ClassPositionsOfNormalSubgroups(ordtbl)
▶ ClassPositionsOfMaximalNormalSubgroups(ordtbl)

correspond to NormalSubgroups and MaximalNormalSubgroups for the group of the ordinary character table ordtbl (see 37.18.7, 37.18.8).

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.)

18 ▶ ClassPositionsOfAgemo(ordtbl, p)

corresponds to Agemo (see 37.13.2) for the group of the ordinary character table ordtbl.

19 ▶ ClassPositionsOfCentre(ordtbl)

corresponds to Centre (see 33.4.5) for the group of the ordinary character table ordtbl.

20 ▶ ClassPositionsOfDirectProductDecompositions(tbl)
▶ ClassPositionsOfDirectProductDecompositions(tbl, nclasses)

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 [l1, l2] where l1 and l2 are lists of class positions of normal subgroups N1, N2 of G such that G is their direct product and |N1| ≤ |N2| 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.

21 ▶ ClassPositionsOfDerivedSubgroup(ordtbl)

corresponds to DerivedSubgroup (see 37.11.3) for the group of the ordinary character table ordtbl.

22 ▶ ClassPositionsOfElementaryAbelianSeries(ordtbl)

corresponds to ElementaryAbelianSeries (see 37.16.9) for the group of the ordinary character table ordtbl.

23 ▶ ClassPositionsOfFittingSubgroup(ordtbl)

corresponds to FittingSubgroup (see 37.11.5) for the group of the ordinary character table ordtbl.

24 ▶ ClassPositionsOfLowerCentralSeries(tbl)

corresponds to LowerCentralSeries (see 37.16.11) for the group of the ordinary character table ordtbl.

25 ▶ ClassPositionsOfUpperCentralSeries(ordtbl)

corresponds to UpperCentralSeries (see 37.16.12) for the group of the ordinary character table ordtbl.

26 ▶ ClassPositionsOfSupersolvableResiduum(ordtbl)

corresponds to SupersolvableResiduum (see 37.11.11) for the group of the ordinary character table ordtbl.

27 ▶ ClassPositionsOfNormalClosure(ordtbl, classes)

is the sorted list of the positions of all conjugacy classes of the ordinary character table ordtbl that form the normal closure (see 37.10.4) of the conjugacy classes at positions in the list classes.
Chapter 69. Character Tables

gap> tbla5 := CharacterTable( "A5" );;
gap> tbls4 := CharacterTable( "Symmetric", 4 );;
gap> tbld8 := CharacterTable( "Dihedral", 8 );;
gap> ClassPositionsOfNormalSubgroups( tbls4 );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1, 2, 3, 4, 5 ] ]
gap> ClassPositionsOfAgemo( tbls4, 2 );
[ 1, 3, 4 ]
gap> ClassPositionsOfCentre( tbld8 );
[ 1, 3 ]
gap> ClassPositionsOfDerivedSubgroup( tbld8 );
[ 1, 3 ]
gap> ClassPositionsOfElementaryAbelianSeries( tbls4 );
[ [ 1 .. 5 ], [ 1, 3, 4 ], [ 1, 3 ], [ 1 ] ]
gap> ClassPositionsOfElementaryAbelianSeries( tbla5 );
fail

69.9 Operations Concerning Blocks

For an ordinary character table \textit{ordtbl} and a prime integer \textit{p}, \texttt{PrimeBlocks} returns a record with the following components.

\textbf{block}

- A list, the value \( j \) at position \( i \) means that the \( i \)-th irreducible character of \textit{ordtbl} lies in the \( j \)-th \( p \)-block of \textit{ordtbl},

\textbf{defect}

- A list containing at position \( i \) the defect of the \( i \)-th block,

\textbf{height}

- A list containing at position \( i \) the height of the \( i \)-th irreducible character of \textit{ordtbl} in its block,

\textbf{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,

\textbf{exponents}

- A list containing at the positions in the component \textit{relevant} an integer \( n \) such that the \( n \)-th power of a difference of characters is divisible by \( p \) if the two characters lie in the same block, and
Section 9. Operations Concerning Blocks

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, exponents, and centralcharacters are used by SameBlock (see 69.9.2).

If InfoCharacterTable has level at least 2, the defects of the blocks and the heights of the characters are printed.

The default method uses the attribute ComputedPrimeBlocks 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 \( \chi, \psi \) of a group \( G \) are said to lie in the same \( p \)-block if the images of their central characters \( \omega_\chi, \omega_\psi \) (see 70.8.17) under the ring homomorphism \( *: R \rightarrow R/M \) 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 \( \chi \) in the block is defined as the largest exponent \( h \) for which \( p^h \) divides \( \chi(1)/p^{a-d} \).

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> pbl:= PrimeBlocks( tbl, 2 );
rec( block := [ 1, 1, 1, 1, 1, 2 ], defect := [ 3, 0 ],
    height := [ 0, 0, 0, 1, 0, 0 ], relevant := [ 3, 5 ],
    exponents := [ ,, 2,, 6 ],
    centralcharacter := [ [ ,, 56,, 24 ], [ ,, -7,, 3 ] ] )
```

2 ▶ SameBlock( tbl, p, omega1, omega2, relevant, exponents ) F

Let \( tbl \) be an ordinary character table, \( p \) a prime integer, \( omega1 \) and \( omega2 \) two central characters (or their values lists) of \( tbl \). The remaining arguments relevant and exponents are lists as stored in the components relevantclasses and exponents of a record returned by PrimeBlocks (see 69.9.1).

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.

```
gap> omega:= List( Irr( tbl ), CentralCharacter );;
gap> SameBlock( tbl, 2, omega[1], omega[2], pbl.relevant, pbl.exponents );
true
```

3 ▶ BlocksInfo( modtbl ) A

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
Chapter 69. Character Tables

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 is called for the block (see 69.9.4),

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.

gap> BlocksInfo( CharacterTable( "L3(2)" ) mod 2 );
[ rec( defect := 3, ordchars := [ 1, 2, 3, 4, 5 ], modchars := [ 1, 2, 3 ],
  decinv := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  basicset := [ 1, 2, 3 ] ),
  rec( defect := 0, ordchars := [ 6 ], modchars := [ 4 ], decinv := [ [ 1 ] ],
  basicset := [ 6 ] ) ]

DecompositionMatrix( modtbl )
DecompositionMatrix( modtbl, blocknr )

Let modtbl be a Brauer character table.

In the first version 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.

In the second version 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 list (see 69.9.3) 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 (see 25.2.1).

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 ],
  [ 1, 1, 0, 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 ] ]

LaTeXStringDecompositionMatrix( modtbl[, blocknr][, options] )

is a string that contains \LaTeX code to print a decomposition matrix (see 69.9.4) 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, rrowlabels (subscripts for the labels). The defaults for phi and chi are \"\tt Y\" and \"\tt X\", the defaults for collabels and rrowlabels 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 \texttt{nrows} and \texttt{ncols} denote the maximal number of rows and columns per array; if they are present then each portion of \texttt{nrows} rows and \texttt{ncols} columns forms an array of its own which is enclosed in $[\,,\,]$.

If the component \texttt{decmat} is bound in \texttt{options} then it must be the decomposition matrix in question, in this case the matrix is not computed from the information in \texttt{modtbl}.

For those character tables from the GAP table library that belong to the ATLAS of Finite Groups [CCN+85], \texttt{AtlasLabelsOfIrreducibles} constructs character labels that are compatible with those used in the ATLAS (see 2.5 and 2.5.1 in the manual of the GAP Character Table Library).

```gap
gap> modtbl:= CharacterTable( "L3(2)" ) mod 2;;
gap> Print( LaTeXStringDecompositionMatrix( modtbl, 1 ) );
\[
\begin{array}{r|rrr} \hline 
& \varphi_{1} & \varphi_{2} & \varphi_{3} \\
\hline
\chi_{1} & 1 & . & . \\
\chi_{2} & . & 1 & . \\
\chi_{3} & . & . & 1 \\
\chi_{4} & . & 1 & 1 \\
\chi_{5} & 1 & 1 & 1 \\
\hline
\end{array}
\]
```

69.10 Other Operations for Character Tables

In the following, we list operations for character tables that are not attributes.

\begin{itemize}
\item [1:\texttt{IsInternallyConsistent( tbl )}]\texttt{O}
\end{itemize}

For an \texttt{ordinary} character table \texttt{tbl}, \texttt{IsInternallyConsistent} checks the consistency of the following attribute values (if stored).

- \texttt{Size}, \texttt{SizesCentralizers}, and \texttt{SizesConjugacyClasses}.
SizesCentralizers and OrdersClassRepresentatives.
- ComputedPowerMaps and OrdersClassRepresentatives.
- SizesCentralizers and Irr.
- Irr (first orthogonality relation).

For a Brauer table \( tbl \), IsInternallyConsistent checks the consistency of the following attribute values (if stored).
- Size, SizesCentralizers, and SizesConjugacyClasses.
- SizesCentralizers and OrdersClassRepresentatives.
- ComputedPowerMaps and OrdersClassRepresentatives.
- Irr (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 is at least 1 (see 7.4), and false is returned at the end.

\[
\begin{align*}
&\text{IsPSolvableCharacterTable}(\ tbl, \ p ) & \text{O} \\
&\text{IsPSolvableCharacterTableOp}(\ tbl, \ p ) & \text{O} \\
&\text{ComputedIsPSolvableCharacterTables}(\ tbl ) & \text{A}
\end{align*}
\]

\( \text{IsPSolvableCharacterTable} \) for the ordinary character table \( tbl \) corresponds to \( \text{IsPSolvable} \) for the group of \( tbl \) (see 37.14.18). \( p \) must be either a prime integer or 0.

The default method uses the attribute \( \text{ComputedIsPSolvableCharacterTables} \) for storing the computed value at position \( p \), and calls the operation \( \text{IsPSolvableCharacterTableOp} \) for computing values that are not yet known.

\[
\begin{align*}
gap> \ tbl:= \text{CharacterTable}(\ "Sz(8)" );; \\
gap> \ \text{IsPSolvableCharacterTable}(\ tbl, \ 2 ); \\
false \\
gap> \ \text{IsPSolvableCharacterTable}(\ tbl, \ 3 ); \\
true
\end{align*}
\]

\[
\begin{align*}
&\text{IsClassFusionOfNormalSubgroup}(\ substbl, \ fus, \ tbl ) & \text{F} \\
\end{align*}
\]

For two ordinary character tables \( tbl \) and \( substbl \) of a group \( G \) and its subgroup \( U \), say, and a list \( fus \) of positive integers that describes the class fusion of \( U \) into \( G \), \( \text{IsClassFusionOfNormalSubgroup} \) returns true if \( U \) is a normal subgroup of \( G \), and false otherwise.

\[
\begin{align*}
gap> \ tblc2:= \text{CharacterTable}(\ "Cyclic", \ 2 );; \\
gap> \ tbld8:= \text{CharacterTable}(\ "Dihedral", \ 8 );; \\
gap> \ fus:= \text{PossibleClassFusions}(\ tblc2, \ tbld8 ); \\
[ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ] \\
gap> \ \text{List}(\ fus, \ map \rightarrow \text{IsClassFusionOfNormalSubgroup}(\ tblc2, \ map, \ tbld8 ) ); \\
[ \text{true, false, false} ]
\end{align*}
\]

\[
\begin{align*}
&\text{Indicator}(\ tbl, \ n ) & \text{O} \\
&\text{Indicator}(\ tbl[, \ characters], \ n ) & \text{O} \\
&\text{Indicator}(\ modtbl, \ 2 ) & \text{O} \\
&\text{IndicatorOp}(\ tbl, \ characters, \ n ) & \text{O} \\
&\text{ComputedIndicators}(\ tbl ) & \text{A}
\end{align*}
\]

If \( tbl \) is an ordinary character table then \( \text{Indicator} \) returns the list of \( n \)-th Frobenius-Schur indicators of the characters in the list \( characters \); the default of \( characters \) is \( \text{Irr}(\ tbl ) \).
The \( n \)-th Frobenius-Schur indicator \( \nu_n(\chi) \) of an ordinary character \( \chi \) of the group \( G \) is given by \( \nu_n(\chi) = \frac{1}{|G|} \sum_{g \in G} \chi(g^n) \).

If \( tbl \) is a Brauer table in characteristic \( \neq 2 \) and \( n = 2 \) then \texttt{Indicator} returns the second indicator.

The default method uses the attribute \texttt{ComputedIndicators} for storing the computed value at position \( n \), and calls the operation \texttt{IndicatorOp} for computing values that are not yet known.

\[
\text{gap> tbl} := \text{CharacterTable( "L3(2)" )};
\text{gap> Indicator( tbl, 2 );}
\text{[ 1, 0, 0, 1, 1, 1 ]}
\]

\texttt{NrPolyhedralSubgroups( tbl, c1, c2, c3 )} 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 p. 233 in [NPP84], 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 69.10.6) and the number of Galois conjugates of a class (see 69.8.15).

The classes \( c1, c2 \) and \( c3 \) in the parameter list must be ordered according to the order of the elements in these classes.

\[
\text{gap> NrPolyhedralSubgroups( tbl, 2, 2, 4 );}
\text{rec( number := 21, type := "D8" )}
\]

\texttt{ClassMultiplicityCoefficient( tbl, i, j, k )} 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 by the following formula.

\[
c_{ijk} = \frac{|C_i||C_j|}{|G|} \sum_{\chi \in \text{Irr}(G)} \frac{\chi(g_i)\chi(g_j)\chi(g_k)}{\chi(1)}.
\]

On the other hand the knowledge of the class multiplication coefficients admits the computation of the irreducible characters of \( G \). (see 69.12.1).

\texttt{ClassStructureCharTable( tbl, classes )} returns the so-called class structure of the classes in the list \( \texttt{classes} \), for the character table \( tbl \) of the group \( G \). The length of \( \texttt{classes} \) 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 \texttt{ClassMultiplicationCoefficient} (see 69.10.6).

\[ n(C_1, C_2, \ldots, C_n) \text{ is computed using the formula} \]

\[ n(C_1, C_2, \ldots, C_n) = \frac{|C_1||C_2|\cdots|C_n|}{|G|} \sum_{\chi \in \text{Irr}(G)} \frac{\chi(g_1)\chi(g_2)\cdots\chi(g_n)}{\chi(1)^{n-2}} \]

For an ordinary character table \( tbl \) and a class position \( i \), \texttt{MatClassMultCoeffsCharTable} returns the matrix \( [a_{ijk}]_{j,k} \) of structure constants (see 69.10.6).

\cadex
\begin{verbatim}
gap> tbl:= CharacterTable( "L3(2)" );;
gap> ClassMultiplicationCoefficient( tbl, 2, 2, 4 );
4
gap> ClassStructureCharTable( tbl, [ 2, 2, 4 ] );
168
gap> ClassStructureCharTable( tbl, [ 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 ] ]
\end{verbatim}

### 69.11 Printing Character Tables

The default \texttt{ViewObj} (see 6.2.3) method for ordinary character tables prints the string "CharacterTable", followed by the identifier (see 69.8.11) or, if known, the group of the character table enclosed in brackets. \texttt{ViewObj} for Brauer tables does the same, except that the first string is replaced by "BrauerTable", and that the characteristic is also shown.

The default \texttt{PrintObj} (see 6.2.3) method for character tables does the same as \texttt{ViewObj}, except that the group is is \texttt{Print}-ed instead of \texttt{View}-ed.

The default \texttt{Display} (see 6.2.4) method for a character table \( tbl \) prepares the data contained in \( tbl \) for a pretty 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 \texttt{SizeScreen} (see 6.11.1).

\texttt{Display} 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 71.1.1). Each displayed character is given a 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 \( tbl \), \texttt{ClassNames} is called (see 69.8.10).

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 19) 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 58.2.4, 18.5.1) and the star of an irrationality (see 18.5.2) 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 together with the corresponding ATLAS-notation (see [CCN+85]).

The optional second argument arec of Display can be used to change the default style (mentioned above) for displaying a character. arec must be a record, its relevant components are the following.

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,

centralizers
- suppresses the printing of the orders of the centralizers if false,

powermap
- an integer or a list of integers to select a subset of the available power maps, or false to suppress the printing of power maps,

letter
- a single capital letter (e. g. "P" for permutation characters) to replace "X",

indicator
- true enables the printing of the second Frobenius Schur indicator, a list of integers enables the printing of the corresponding indicators (see 69.10.4),

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); the default StringEntry function is CharacterTableDisplayStringEntryDefault,

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; the default StringEntryData function is CharacterTableDisplayStringEntryDataDefault,

PrintLegend
- a function that is called with the result of the StringEntryData call after the character table has been displayed; the default PrintLegend function is CharacterTableDisplayPrintLegendDefault.

If the value of DisplayOptions (see 69.11.1) is stored on tbl, it is used as default value for arec in the one argument call of Display.

There is no default method to compute a value, one can set a value with SetDisplayOptions.

```gap
gap> tbl:= CharacterTable( "A5" );;
gap> Display( tbl );
A5

  2  2  2 . . .
  3  1 .  1 . .
  5  1 . . 1 1
```
Chapter 69. Character Tables

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>2a</th>
<th>3a</th>
<th>5a</th>
<th>5b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2P</td>
<td>1a</td>
<td>1a</td>
<td>3a</td>
<td>5b</td>
<td>5a</td>
</tr>
<tr>
<td>3P</td>
<td>1a</td>
<td>2a</td>
<td>1a</td>
<td>5b</td>
<td>5a</td>
</tr>
<tr>
<td>5P</td>
<td>1a</td>
<td>2a</td>
<td>3a</td>
<td>1a</td>
<td>1a</td>
</tr>
</tbody>
</table>

\[\begin{array}{cccc}
X.1 & 1 & 1 & 1 & 1 \\
X.2 & 3 & -1 & . & A *A \\
X.3 & 3 & -1 & . & *A A \\
X.4 & 4 & . & 1 & -1 -1 \\
X.5 & 5 & 1 & -1 & . . \\
\end{array}\]

\[A = -E(5)-E(5)^4\]
\[= (1-ER(5))/2 = -b5\]

gap> options:= rec( chars:= 4, classes:= [ tbl.3a .. tbl.5a ],
  > centralizers:= false, indicator:= true,
  > powermap:= [ 2 ] );;
gap> Indicator( tbl, 2 );;
gap> Display( tbl, options );

\[A5\]

\[\begin{array}{ccc}
3a & 5a \\
2P & 3a & 5b \\
2 & \\
X.4 & + & 1 -1 \\
\end{array}\]

\[\text{F}\]

PrintCharacterTable( \[tbl\], \[varname\] )

Let \(tbl\) be a nearly character table, and \(varname\) a string. PrintCharacterTable prints those values of the supported attributes (see 69.3.3) 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.

gap> PrintCharacterTable( CharacterTable( "Cyclic", 2 ), "tbl" );
tbl:= function()
  local tbl;
  tbl:=rec();
  tbl.Irr:=
    [ [ 1, 1 ], [ 1, -1 ] ];
  tbl.NrConjugacyClasses:=
    2;
  tbl.Size:=
    2;
  tbl.OrdersClassRepresentatives:=
    [ 1, 2 ];
Section 12. Computing the Irreducible Characters of a Group

715

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();

69.12 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 69.12.1). For supersolvable groups, Conlon’s algorithm can be used (see 69.12.2). For abelian-by-supersolvable groups, the Baum-Clausen algorithm for computing the irreducible representations (see 69.12.4) can be used to compute the irreducible characters (see 69.12.3).

These functions are installed in methods for Irr (see 69.8.2), but explicitly calling one of them will not set the Irr value of $G$.

1 ▶ IrrDixonSchneider( G )

computes the irreducible characters of the finite group $G$, using the Dixon-Schneider method (see 69.13). It calls DixonInit and DixonSplit, and finally returns the list returned by DixontinI (see 69.14, 69.15, 69.16).

2 ▶ IrrConlon( G )

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 72.3.1) is stored.

3 ▶ IrrBaumClausen( G )

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.

In the following example we temporarily increase the line length limit from its default value 80 to 87 in order to get a nicer output format.
Chapter 69. Character Tables

```plaintext
gap> g := SL(2,3);;
gap> SizeScreen([ 87, ]);;
gap> irr1 := IrrDixonSchneider( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3), E(3)^2, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3)^2, E(3), E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, -2, 1, 1, -1, -1, 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, -2, E(3), E(3)^2, -E(3)^2, -E(3), 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, -2, E(3)^2, E(3), -E(3), -E(3)^2, 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 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, 1, E(3), E(3)^2, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3)^2, E(3), E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 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, 1, E(3), E(3)^2, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3)^2, E(3), E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] ) ]
gap> SizeScreen([ 80, ]);;
gap> chi := irr2[4];;
true
```

```plaintext
4\textbf{IrreducibleRepresentations( G )} \quad A
4\textbf{IrreducibleRepresentations( G, F )} \quad O

Called with a finite group $G$ and a field $F$, \textbf{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 \textbf{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 69.12.5) is used.

For other cases no methods are implemented yet.

See also \textbf{IrreducibleModules}, which provides efficient methods for solvable groups.
```

```plaintext
gap> g := AlternatingGroup( 4 );;
gap> repr := IrreducibleRepresentations( g );
[ Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
  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 ] ] ),
```
Section 13. The Dixon-Schneider Algorithm

\[ \begin{bmatrix} 1, 0, 0 \end{bmatrix}, \begin{bmatrix} 0, -1, 0 \end{bmatrix}, \begin{bmatrix} 0, 0, -1 \end{bmatrix} \] \]

\texttt{gap> ForAll( repr, IsGroupHomomorphism );}
\texttt{true}
\texttt{gap> Length( repr );}
\texttt{4}
\texttt{gap> gens:= GeneratorsOfGroup( g );}
\texttt{\([1,2,3], (2,3,4)\]}
\texttt{gap> List( gens, x -> x^repr[1] );}
\texttt{\([1\times1], [1\times1]\]}
\texttt{gap> List( gens, x -> x^repr[4] );}
\texttt{\([\begin{bmatrix} 0, 0, -1 \end{bmatrix}, \begin{bmatrix} 1, 0, 0 \end{bmatrix}, \begin{bmatrix} 0, -1, 0 \end{bmatrix}, \begin{bmatrix} 0, 1, 0 \end{bmatrix}, \begin{bmatrix} 0, 0, 1 \end{bmatrix}, \begin{bmatrix} 1, 0, 0 \end{bmatrix}\]}

\textbf{69.13 The Dixon-Schneider Algorithm}

The \texttt{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 \texttt{CharacterTable} (see 69.3.1) 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 69.6). Thus it is possible to combine computations using the group with character theoretic computations (see 69.14 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 \texttt{IsDxLargeGroup} (see 69.14.8).

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 69.14.1) in the group that helps routines that identify classes, for
example \texttt{FusionConjugacyClasses}, to work much faster. Note that interrupting Dixon-Schneider calcula-
tions will prevent \texttt{GAP} from cleaning up the Dixon record; when the computation by \texttt{IrrDixonSchneider}
is complete, the possibly large record is shrunk to an acceptable size.

### 69.14 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 \texttt{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 \texttt{InfoCharacterTable} if positive, information about the progress of splitting is printed.
(The default value is zero.)

1. \texttt{DixonRecord( G )} AM

The \texttt{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 .\texttt{conjugacyClasses} which contains the classes of $G$ ordered as the algorithm
needs them.

2. \texttt{DixonInit( G )} F

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 69.14.1,
69.15) that can be used when calculating the irreducible characters of $G$ interactively.

3. \texttt{DixontinI( D )} F

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.

4. \texttt{DixonSplit( D )} F

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 \texttt{BestSplittingMatrix} and the options described for this function
apply here.

\texttt{DixonSplit} returns \texttt{true} if a split was performed and \texttt{fail} otherwise.
Section 15. Components of a Dixon Record

5 ▶ BestSplittingMatrix( D )
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) 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.

6 ▶ DxIncludeIrreducibles( D, new[, newmod] )
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.

7 ▶ SplitCharacters( D, list )
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.

8 ▶ IsDxLargeGroup( G )
returns true if the order of the group G is smaller than the current value of the global variable DXLARGE-GROUPORDER, 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.

69.15 Components of a Dixon Record

The “Dixon record” D returned by DixonInit (see 69.14.2) 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 69.16).

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 value of G (whose characters are not yet known),

ClassElement( D, el ):
a function that returns the number of the class of G that contains the element el.
69.16 An Example of Advanced Dixon-Schneider Calculations

First, we set the appropriate info level higher

```gap
SetInfoLevel(InfoCharacterTable, 1);
```

for printout of some internal results. We now define our group, which is isomorphic to PSL₄(3).

```gap
g:= PrimitiveGroup(40,5);
PSL(4,3)
Size(g);
6065280
d:= DixonInit( g );
#I 29 classes
#I choosing prime 28081
c:= d.characterTable;;
```

After the initialisation, one structure matrix is evaluated, yielding smaller spaces and several irreducible characters.

```gap
DixonSplit( d );
#I Matrix 18, Representative of Order 3, Centralizer: 5832
#I Dimensions: [ 1, 12, 2, 2, 4, 2, 1, 1, 1, 1 ]
#I Two-dim space split
#I Two-dim space split
#I Two-dim space split
18
```

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.

```gap
asp:= AntiSymmetricParts( c, d.irreducibles, 2 );
ro:= ReducedOrdinary( c, d.irreducibles, asp );
Length( ro.irreducibles );
3
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.

```gap
nlc:= Filtered( d.irreducibles, i -> i[1] > 1 );
t:= Tensored( nlc, nlc );
ro:= ReducedOrdinary( c, d.irreducibles, t );
ro.remainders;
[ ]
List( ro.remainders, i -> ScalarProduct( c, i, i ) );
[ 2, 2, 4, 4, 4, 4, 13, 13, 18, 18, 18, 18, 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 ]
t:= SplitCharacters( d, ro.remainders );
```
Finally we calculate the characters induced from all cyclic subgroups and obtain the missing irreducibles by applying the LLL-algorithm to them.

```gap
gap> ic:= InducedCyclic( c, "all" );;
gap> ro:= ReducedOrdinary( c, d.irreducibles, ic );;
gap> Length( ro.irreducibles );
0
```

The LLL returns class function objects (see Chapter 70), 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.

```gap
gap> l:= LLL( c, ro.remainders );;
gap> Length( l.irreducibles );
13
```

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.

```gap
gap> irrs:= DixontinI( d );;
#I Total:1 matrices,[ 18 ]
gap> SetIrr(g,irrs);
gap> Length(Irr(c));
29
```

### 69.17 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` (see 69.3.2); note that the return value of `BrauerTable` 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` values are updated.
Chapter 69. Character Tables

(The interactive construction of character tables using character theoretic methods and incomplete tables is not described here.) \textit{Currently it is not supported and will be described in a chapter of its own when it becomes available.}

\begin{verbatim}
1 \texttt{CharacterTableDirectProduct( tbl1, tbl2 )}

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 24.4.8) 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 tbl1 and tbl2 may be computed in order to construct the direct product.

The embeddings of tbl1 and tbl2 into the direct product are stored, they can be fetched with \texttt{GetFusionMap} (see 71.2.3); if tbl1 is equal to tbl2 then the two embeddings are distinguished by their \texttt{specification} components "1" and "2", respectively.

Analogously, the projections from the direct product onto tbl1 and tbl2 are stored, and can be distinguished by the \texttt{specification} components.

The \texttt{*} operator for two character tables (see 69.7) delegates to \texttt{CharacterTableDirectProduct}.

\begin{verbatim}
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

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  

2 \texttt{CharacterTableFactorGroup( tbl, classes )}

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 \texttt{classes}.

The \texttt{\*} operator for a character table and a list of class positions (see 69.7) delegates to \texttt{CharacterTableFactorGroup}.

\begin{verbatim}
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfNormalSubgroups( s4 );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1, 2, 3, 4, 5 ] ]
gap> f:= CharacterTableFactorGroup( s4, [ 3 ] );
CharacterTable( "Sym(4)/[ 1, 3 ]" )
gap> Display( f );
Sym(4)/[ 1, 3 ]
\end{verbatim}
\end{verbatim}
Section 17. Constructing Character Tables from Others

\[ \begin{array}{cccc}
2 & 1 & 1 & . \\
3 & 1 & . & 1 \\
1a & 2a & 3a \\
2P & 1a & 1a & 3a \\
3P & 1a & 2a & 1a \\
\end{array} \]

\[ \begin{array}{cccc}
X.1 & 1 & -1 & 1 \\
X.2 & 2 & . & -1 \\
X.3 & 1 & 1 & 1 \\
\end{array} \]

3 \texttt{CharacterTableIsoclinic( tbl )} \quad \text{O}

3 \texttt{CharacterTableIsoclinic( tbl, classes )} \quad \text{O}

3 \texttt{CharacterTableIsoclinic( tbl, classes, centre )} \quad \text{O}

If \( tbl \) is the character table of a group with structure \( 2.G.2 \) with a central subgroup \( Z \) of order 2 and a normal subgroup \( N \) of index 2 that contains \( Z \) then \texttt{CharacterTableIsoclinic} returns the character 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 \texttt{classes}. If \( Z \) is not unique in \( N \) then the position of the class consisting of the involution in \( Z \) must be entered as \texttt{centre}.

\texttt{gap> d8:= CharacterTable( "Dihedral", 8 );;}
\texttt{gap> nsg:= ClassPositionsOfNormalSubgroups( d8 );}
\texttt{[ [ 1 ], [ 1, 3 ], [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ], [ 1, 2, 3, 4, 5 ] ]}
\texttt{gap> Display( CharacterTableIsoclinic( d8, nsg[3] ) );}
\texttt{Isoclinic(D8)}

\[ \begin{array}{cccccccc}
2 & 3 & 2 & 3 & 2 & 2 \\
1a & 4a & 2a & 4b & 4c \\
2P & 1a & 2a & 1a & 2a & 2a \\
\end{array} \]

\[ \begin{array}{cccc}
X.1 & 1 & 1 & 1 & 1 \\
X.2 & 1 & 1 & 1 & -1 & -1 \\
X.3 & 1 & -1 & 1 & 1 & -1 \\
X.4 & 1 & -1 & 1 & -1 & 1 \\
X.5 & 2 & . & -2 & . & . \\
\end{array} \]

4 \texttt{CharacterTableWreathSymmetric( tbl, n )} \quad \text{F}

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 \texttt{ClassParameters} and \texttt{CharacterParameters} (see 2.3.2, 2.3.2 in the manual for the GAP Character Table Library) 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.

\texttt{gap> c3:= CharacterTable( "Cyclic", 3 );;}
\texttt{gap> wr:= CharacterTableWreathSymmetric( c3, 2 );;}
\texttt{gap> Display( wr );}
\texttt{C3wrS2}

\[ \begin{array}{cccccccc}
2 & 1 & . & . & 1 & 1 & 1 & 1 \\
3 & 2 & 2 & 2 & 2 & 2 & 1 & 1 \\
\end{array} \]
### Chapter 69. Character Tables

#### 69.18 Sorted Character Tables

1. `CharacterTableWithSortedCharacters(tbl)`
2. `CharacterTableWithSortedCharacters(tbl, perm)`

is a character table that differs from `tbl` only by the succession of its irreducible characters. This affects the values of the attributes `Irr` (see 69.8.2) and `CharacterParameters` (see 2.3.2 in the manual for the GAP Character Table Library). 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 69.18.2.

The result has all those attributes and properties of `tbl` that are stored in `SupportedCharacterTableInfo` and do not depend on the ordering of characters (see 69.3.3).

3. `SortedCharacters(tbl, chars)`
4. `SortedCharacters(tbl, chars, "norm")`
5. `SortedCharacters(tbl, chars, "degree")`

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 (parameter "norm"), to ascending degree (parameter "degree"), or both (no third parameter), 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.

6. `CharacterTableWithSortedClasses(tbl)`
7. `CharacterTableWithSortedClasses(tbl, "centralizers")`
8. `CharacterTableWithSortedClasses(tbl, "representatives")`
9. `CharacterTableWithSortedClasses(tbl, permutation)`

is a character table obtained by permutation of the classes of `tbl`. If the second argument is the string "centralizers" then the classes of the result are sorted according to descending centralizer orders. If the
Section 18. Sorted Character Tables

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 $perm$ 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 $SupportedCharacterTableInfo$ and do not depend on the ordering of classes (see 69.3.3).

\[\text{SortedCharacterTable( } tbl, \text{ kernel } ) \quad F\]
\[\text{SortedCharacterTable( } tbl, \text{ normalseries } ) \quad F\]
\[\text{SortedCharacterTable( } tbl, \text{ facttbl, kernel } ) \quad F\]

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 $\text{Irr( } tbl )$ 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 $\text{TransformingPermutationsCharacterTables}$ (see 69.19.4), 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 $facttbl$. $\text{Irr( } tbl )$ is sorted as with $\text{SortCharTable( } tbl, \text{ kernel } )$.

(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 $SupportedCharacterTableInfo$ and do not depend on the ordering of classes and characters (see 69.3.3).

The $\text{ClassPermutation}$ value of $tbl$ is changed if necessary, see 69.5.

$\text{SortedCharacterTable}$ uses $\text{CharacterTableWithSortedClasses}$ and $\text{CharacterTableWithSortedCharacters}$ (see 69.18.3, 69.18.1).

\[\text{ClassPermutation( } tbl ) \quad A\]

is a permutation $\pi$ 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 $\pi$ 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 $\text{CharacterTableWithSortedClasses}$ or $\text{SortedCharacterTable}$, (see 69.18.3, 69.18.4).

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.

\[\text{gap> tbl:= CharacterTable( "Symmetric", 4 );}\]
\[\text{CharacterTable( "Sym(4)" )}\]
\[\text{gap> Display( tbl );}\]
\[\text{Sym(4)}\]
\[
\begin{array}{cccccccc}
2 & 3 & 2 & 3 & . & 2 \\
3 & 1 & . & 1 & \\
\end{array}
\]
69.19 Automorphisms and Equivalence of Character Tables

For a matrix \( \text{mat} \), \text{MatrixAutomorphisms} returns the group of those permutations of the columns of \( \text{mat} \) that leave the set of rows of \( \text{mat} \) 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, 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 69.8.5) may be an entry in maps.

2 TableAutomorphisms( tbl, characters[, "closed"] )

TableAutomorphisms returns the permutation group of those matrix automorphisms (see 69.19.1) of the list characters that leave the element orders (see 69.8.5) and all stored power maps (see 71.1.1) of the character table tbl.

If characters is closed under Galois conjugacy –this is always fulfilled for ordinary character tables– the string "closed" may be entered as third argument.

The attribute AutomorphismsOfTable (see 69.8.8) can be used to compute and store the table automorphisms for the case that characters equals Irr( tbl ).

```gap> tbld8:= CharacterTable( "Dihedral", 8 );;
<character table of size 8, order 32, degree 4>
```

```gap> irrd8:= Irr( tbld8 );
[ Character( CharacterTable( "D8" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "D8" ), [ 1, 1, -1, -1 ] ),
  Character( CharacterTable( "D8" ), [ 1, -1, 1, -1 ] ),
  Character( CharacterTable( "D8" ), [ 1, -1, 1, 1 ] ),
  Character( CharacterTable( "D8" ), [ 2, 0, -2, 0 ] ) ]
```

```gap> orders:= OrdersClassRepresentatives( tbld8 );
[ 1, 4, 2, 2, 2 ]
```

```gap> MatrixAutomorphisms( irrd8 );
Group([ (2,4) ])
```

```gap> MatrixAutomorphisms( irrd8, [ orders ], Group( () ) );
Group([ (2,4) ])
```

```gap> TableAutomorphisms( tbld8, irrd8 );
Group([ (2,4) ])
```

3 TransformingPermutations( mat1, mat2 )

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 components columns, rows, and group is returned, otherwise fail. For TransformingPermutations(mat1, mat2) = r ≠ fail, we have mat2 = Permed( List( mat1, x -> Permed( x, r.columns ) ),r.rows ).

r.group is the group of matrix automorphisms of mat2 (see 69.19.1). 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.

4 TransformingPermutationsCharacterTables( tbl1, tbl2 )

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 69.19.3), 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 69.8.8) is returned. If no such permutation exists, fail is returned.
Chapter 69. Character Tables

gap> tblq8 := CharacterTable( "Quaternionic", 8 );;
gap> irrq8 := Irr( tblq8 );
[ Character( CharacterTable( "Q8" ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( "Q8" ), [ 1, 1, 1, -1 ] ),
  Character( CharacterTable( "Q8" ), [ 1, -1, 1, 1 ] ),
  Character( CharacterTable( "Q8" ), [ 1, -1, 1, -1 ] ),
  Character( CharacterTable( "Q8" ), [ 2, 0, -2, 0 ] ) ]
gap> OrdersClassRepresentatives( tblq8 );
[ 1, 4, 2, 4, 4 ]
gap> TransformingPermutations( irrd8, irrq8 );
rec( columns := (), rows := (), group := Group([ (4,5), (2,4) ]) )
gap> TransformingPermutationsCharacterTables( tbld8, tblq8 );
fail
gap> tbld6 := CharacterTable( "Dihedral", 6 );
gap> tbls3 := CharacterTable( "Symmetric", 3 );
gap> TransformingPermutationsCharacterTables( tbld6, tbls3 );
rec( columns := (2,3), rows := (1,3,2), group := Group(()) )

5 FamiliesOfRows( mat, maps )

F 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( mat, maps ) returns a record with 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).

69.20 Interface to the CAS System

CASString( tbl )

F is a string that encodes the CAS library format of the character table tbl. This string can be printed to a file which then can be read into the CAS system using its get command (see [NPP84]).

The used line length is SizeScreen()[1] (see 6.11.1).

Only the known values of the following attributes are used. ClassParameters (for partitions only), ComputedClassFusions, ComputedPowerMaps, Identifier, InfoText, Irr, ComputedPrimeBlocks, ComputedIndicators, OrdersClassRepresentatives, Size, SizesCentralizers.

gap> Print( CASString( CharacterTable( "Cyclic", 2 ) ), "\n" );
'O0/00/00. 00.00.00.
(2,2,0,2,-1,0)
text:
(#computed using generic character table for cyclic groups#),
Section 21. Interface to the MOC System

Additionally it provides an alternative representation of (virtual) characters.

The MOC 3 code of a 5 digit number in MOC 2 code is given by the following list. (Note that the code must contain only lower case letters.)

<table>
<thead>
<tr>
<th>MOC 2</th>
<th>MOC 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>for 0ABCD</td>
</tr>
<tr>
<td>a</td>
<td>for 10000</td>
</tr>
<tr>
<td>b</td>
<td>for 10001</td>
</tr>
<tr>
<td>c</td>
<td>for 10002</td>
</tr>
<tr>
<td>d</td>
<td>for 10003</td>
</tr>
<tr>
<td>e</td>
<td>for 10004</td>
</tr>
<tr>
<td>f</td>
<td>for 10005</td>
</tr>
<tr>
<td>g</td>
<td>for 10006</td>
</tr>
<tr>
<td>h</td>
<td>for 10007</td>
</tr>
<tr>
<td>i</td>
<td>for 10008</td>
</tr>
<tr>
<td>j</td>
<td>for 10009</td>
</tr>
<tr>
<td>tAB</td>
<td>for 100AB</td>
</tr>
<tr>
<td>uAB</td>
<td>for 200AB</td>
</tr>
<tr>
<td>vABCD</td>
<td>for 1ABCD</td>
</tr>
<tr>
<td>wABCD</td>
<td>for 2ABCD</td>
</tr>
<tr>
<td>yABC</td>
<td>for 30ABC</td>
</tr>
<tr>
<td>z</td>
<td>for 31000</td>
</tr>
</tbody>
</table>

Note that any long number in MOC 2 format is divided into packages of length 4, the beginning (!) filled with leading zeros if necessary. Such a number with decimals \( d_1, d_2, \ldots, d_{4n+k} \) is the sequence

\[
0d_1d_2d_3d_4 \ldots 0d_{4n-3}d_{4n-2}d_{4n-1}d_{4n}xd_{4n+1} \ldots d_{4n+k}
\]

where \( 0 \leq k \leq 3 \), the first digit of \( x \) is 1 if the number is positive and 2 if the number is negative, and then follow \((4 - k)\) zeros. A brief description of the MOC system can be found in [LP91].

\[\text{MAKElb11}( \ \text{listofns} \ )\]

\text{MAKElb11} prints field information for all number fields with conductor \( n \) where the positive integer \( n \) is in the list \text{listofns}. 
Chapter 69. Character Tables

The output of \texttt{MAKElb11} is used by the MOC system. \texttt{MAKElb11([ 3 .. 189 ])} will print something very similar to Richard Parker’s file lb11.

\begin{verbatim}
gap> MAKElb11([ 3, 4 ]); 3 2 0 1 0
   4 2 0 1 0
\end{verbatim}

\texttt{MOCTable( gaptbl )} \qquad F

\texttt{MOCTable( gaptbl, basicset )} \qquad F

\texttt{MOCTable} returns the MOC table record of the GAP character table \texttt{gaptbl}.

The first form can be used only if \texttt{gaptbl} is an ordinary (\texttt{G.0}) table. For Brauer (\texttt{G.p}) tables one has to specify a basic set \texttt{basicset} of ordinary irreducibles. \texttt{basicset} must be a list of positions of the basic set characters in the \texttt{Irr} list of the ordinary table of \texttt{gaptbl}.

The result is a record that contains the information of \texttt{gaptbl} in a format similar to the MOC 3 format. This record can e.g. easily be printed out or be used to print out characters using \texttt{MOCString} (see 69.21.3).

The components of the result are

\begin{itemize}
  \item \texttt{identifier}  
    the string \texttt{MOCTable(name)} where \texttt{name} is the \texttt{Identifier} value of \texttt{gaptbl},
  
  \item \texttt{GAPtbl}  
    \texttt{gaptbl},
  
  \item \texttt{prime}  
    the characteristic of the field (label 30105 in MOC),
  
  \item \texttt{centralizers}  
    centralizer orders for cyclic subgroups (label 30130)
  
  \item \texttt{orders}  
    element orders for cyclic subgroups (label 30140)
  
  \item \texttt{fieldbases}  
    at position \texttt{i} the Parker basis of the number field generated by the character values of the \texttt{i}-th cyclic subgroup. The length of \texttt{fieldbases} is equal to the value of label 30110 in MOC.
  
  \item \texttt{cycsubgps}  
    \texttt{cycsubgps}[\texttt{i}] = \texttt{j} means that class \texttt{i} of the GAP table belongs to the \texttt{j}-th cyclic subgroup of the GAP table,
  
  \item \texttt{repcycsub}  
    \texttt{repcycsub}[\texttt{j}] = \texttt{i} means that class \texttt{i} of the GAP table is the representative of the \texttt{j}-th cyclic subgroup of the GAP table. \textbf{Note} that the representatives of GAP table and MOC table need not agree!
  
  \item \texttt{galconjinfo}  
    a list \([r_1, c_1, r_2, c_2, \ldots, r_n, c_n]\) which means that the \texttt{i}-th class of the GAP table is the \texttt{c_i}-th conjugate of the representative of the \texttt{r_i}-th cyclic subgroup on the MOC table. (This is used to translate back to GAP format, stored under label 30160)

  \item 30170  
    (power maps) for each cyclic subgroup (except the trivial one) and each prime divisor of the representative order store four values, namely the number of the subgroup, the power, the number of the cyclic subgroup containing the image, and the power to which the representative must be raised to yield the image class. (This is used only to construct the 30230 power map/embedding information.) In 30170 only a list of lists (one for each cyclic subgroup) of all these values is stored, it will not be used by GAP.
\end{itemize}
Section 21. Interface to the MOC System

731

tensinfo
tensor product information, used to compute the coefficients of the Parker base for tensor products of characters (label 30210 in MOC). For a field with vector space basis \( (v_1, v_2, \ldots, v_n) \) the tensor product information of a cyclic subgroup in MOC (as computed by \( \text{fct} \)) is either 1 (for rational classes) or a sequence

\[ n x_1, y_1, z_1, \ldots, x_m, y_m, z_m 0 \cdots z_{n,m} 0 \]

which means that the coefficient of \( v_k \) in the product

\[ \left( \sum_{i=1}^{n} a_i v_i \right) \left( \sum_{j=1}^{n} b_j v_j \right) \]

is equal to

\[ \sum_{i=1}^{m_k} x_{k,i} a_{y_k,i} b_{z_k,i}. \]

On a MOC table in GAP the tensinfo component is a list of lists, each containing exactly the sequence mentioned above.

invmap
inverse map to compute complex conjugate characters, label 30220 in MOC.

powerinfo
field embeddings for \( p \)-th symmetrizations, \( p \) prime in \([ 2 \ldots 19 ]\); note that the necessary power maps must be stored on \( \text{gaptbl} \) to compute this component. (label 30230 in MOC)

30900
basic set of restricted ordinary irreducibles in the case of nonzero characteristic, all ordinary irreducibles otherwise.

\[ 3 \xrightarrow{\text{MOCString}} \text{MOCString}( \text{moctbl} ) \quad \text{F} \]
\[ 3 \xrightarrow{\text{MOCString}} \text{MOCString}( \text{moctbl}, \text{chars} ) \quad \text{F} \]

Let \( \text{moctbl} \) be a MOC table record as returned by \( \text{MOCTable} \) (see 69.21.2). \text{MOCString} returns a string describing the MOC 3 format of \( \text{moctbl} \).

If the second argument \( \text{chars} \) is specified, it must be a list of MOC format characters as returned by \( \text{MOCChars} \) (see 69.21.6). In this case, these characters are stored under label 30900. If the second argument is missing then the basic set of ordinary irreducibles is stored under this label.

\[ \text{gap> moca5:= MOCTable( CharacterTable( "A5" ) );} \]
\[ \text{rec( identifier := "MOCTable(A5)", prime := 0, fields := [ ], GAPtbl := CharacterTable( "A5" ), cycsubgps := [ 1, 2, 3, 4, 4 ], repcycsub := [ 1, 2, 3, 4 ], galconjinfo := [ 1, 1, 2, 1, 3, 1, 4, 1, 4, 2 ], centralizers := [ 60, 4, 3, 5 ], orders := [ 1, 2, 3, 5 ], fieldbases := [ CanonicalBasis( Rationals ), CanonicalBasis( Rationals ), CanonicalBasis( Rationals ), Basis( NF(5,[ 1, 1, 4 ]), [ 1, E(5)+E(5)^4 ] ), 30170 := [ [ ], [ 2, 2, 1, 1 ], [ 3, 3, 1, 1 ], [ 4, 5, 1, 1 ] ], tensinfo := [ [ 1 ], [ 1 ], [ 1 ], [ 2, 1, 1, 1, 1, 2, 2, 0, 1, 1, 2, 1, 2, 1, -1, 2, 2, 0 ] ], invmap := [ [ 1, 1, 0 ], [ 1, 2, 0 ], [ 1, 3, 0 ], [ 1, 4, 0, 1, 5, 0 ] ], powerinfo := [ [ 1, 1, 0 ], [ 1, 1, 0 ], [ 1, 3, 0 ], [ 1, 4, -1, 5, 0, -1, 5, 0 ] ] } \]
![OCR-scanned text](https://i.imgur.com/4Zd5B.jpg)
69.22 Storing Normal Subgroup Information

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

\begin{itemize}
  \item $nsg$: list of normal subgroups of $G$, may be incomplete,
  \item $nsgclasses$: at position $i$, the list of positions of conjugacy classes of $tbl$ forming the $i$-th entry of the $nsg$ component,
  \item $nsgfactors$: at position $i$, if bound, the factor group modulo the $i$-th entry of the $nsg$ component.
\end{itemize}

$NormalSubgroupClasses$, $FactorGroupNormalSubgroupClasses$, and $ClassPositionsOfNormalSubgroup$ 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 $NormalSubgroupClasses$. 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 $KernelOfCharacter$ function you make use of this feature because $KernelOfCharacter$ calls $NormalSubgroupClasses$.

\begin{itemize}
  \item $ClassPositionsOfNormalSubgroup( tbl, N )$ 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 $tbl$.
  \item $NormalSubgroupClasses( tbl, classes )$ 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 $NormalSubgroupClassesInfo( tbl ).normalSubgroups$.
  \item $FactorGroupNormalSubgroupClasses( tbl, classes )$ 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 $classes$.
\end{itemize}
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 ] ),
  Character( CharacterTable( S4 ), [ 3, -1, -1, 0 ],
  Character( CharacterTable( S4 ), [ 2, 0, 2, -1, 0 ] ),
  Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] ),
  Character( CharacterTable( S4 ), [ 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 ] ) ] )
This chapter describes operations for class functions of finite groups. For operations concerning character tables, see Chapter 69.

\begin{verbatim}
IsClassFunction( obj )
\end{verbatim}

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 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 69.6); 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 (see 70.2.1). 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. 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 (see 31.3.6) and PreImage (see 31.4.6) play no important role for class functions.

\section{70.1 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 “$\chi$ is a character of a group $G$” then this object $\chi$ carries a lot of information. $\chi$ has certain properties such as being irreducible or not. Several subgroups of $G$ are related to $\chi$, such as the kernel and the centre of $\chi$. 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 \( \cong \), 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 \textbf{Irr} lists (see 69.8.2) consist of class function objects, and \textbf{TrivialCharacter} (see 70.7.1) returns a class function object.

Here is an example that shows both approaches. First we define some groups.

```gap
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.

```gap
gap> irrS4:= Irr( S4 );;
gap> irrD8:= Irr( D8 );;
gap> chi:= irrD8[3];
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, 0, 0, 1, 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.

```gap
gap> tS4:= CharacterTable( S4 );;
gap> tD8:= CharacterTable( D8 );;
gap> chi:= ValuesOfClassFunction( irrD8[3] );
[ 1, 1, 1, -1, -1 ]
gap> Tensored( [ chi ], [ chi ] )[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, 0, 0, 1, 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.)

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 ] )
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 ] )

Functions that compute normal subgroups related to characters have counterparts that return the list of class positions corresponding to these groups.

gap> ClassPositionsOfKernel( irrs5[2] );
[ 1, 2, 3, 4 ]

Non-normal subgroups cannot be described this way, so for example inertia subgroups (see 70.8.13) can in general not be computed from character tables without access to their groups.

70.2 Basic Operations for Class Functions

Basic operations for class functions are UnderlyingCharacterTable (see 70.2.1), ValuesOfClassFunction (see 70.2.2), and the basic operations for lists (see 21.2).

1 UnderlyingCharacterTable( psi )

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 70.2.2) 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 69.8.4), 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 69.3.2). So the underlying characteristic of psi can be read off from the underlying character table.

2 ValuesOfClassFunction( psi )

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 70.2.1).
Chapter 70. Class Functions

With respect to \( = \) and \( < \), class functions behave equally to their lists of values (see 70.2.2). 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.

```gap
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[3] < irr1[2];
true
```

70.3 Comparison of Class Functions

With respect to \( = \) and \( < \), class functions behave equally to their lists of values (see 70.2.2). 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.
Section 4. Arithmetic Operations for Class Functions

70.4 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; see 70.8.5.) In terms of filters, the arithmetic of class functions is based on the decision that they lie in IsGeneralizedRowVector, with additive nesting depth 1, but they do not lie in IsMultiplicativeGeneralizedRowVector (see 21.12).

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 70.2.1) 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.

```gap
gap> g := SymmetricGroup( 4 );;
gap> tbl := CharacterTable( g );;
gap> SetName( tbl, "S4" );;
gap> 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
gap> irr[1] / 7;
ClassFunction( S4, [ 1/7, -1/7, 1/7, 1/7, -1/7 ] )
```

```gap
VirtualCharacter( S4, [ 2, -2, 2, -1, -2 ] )
```

```gap
gap> lincomb := lincomb + 2 * irr[5];
```

```gap
gap> IsCharacter( lincomb );
true
```

```gap
gap> lincomb;
Character( S4, [ 4, 0, 4, 1, 0 ] )
```

```gap
gap> irr[5] + 2;
[ 3, 3, 3, 3, 3 ]
```

```gap
gap> irr[5] + [ 1, 2, 3, 4, 5 ];
[ 2, 3, 4, 5, 6 ]
```

```gap
gap> zero := 0 * irr[1];
VirtualCharacter( S4, [ 0, 0, 0, 0, 0 ] )
```

```gap
gap> zero + Z(3);
[ Z(3), Z(3), Z(3), Z(3), Z(3) ]
```

```gap
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.
Chapter 70. Class Functions

```gap
tens := irr[3] * irr[4];
Character( S4, [ 6, 0, -2, 0, 0 ] )
tens / irr[1];
Character( S4, [ 6, 0, -2, 0, 0 ] )
```

Class functions without zero values are invertible, the `inverse` is defined pointwise. As a consequence, for example groups of linear characters can be formed.

```gap
4
```

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.

```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.

```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.

```gap
irr * irr;
Character( S4, [ 24, 4, 8, 3, 4 ] )
```

The `powering` operator `^` 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 `^`; instead one can use `Permuted`, see 21.20.15.) The power of a class function by a group or a character table is the induced class function (see 70.9.3). The power of a group element by a class function is the class function value at (the conjugacy class containing) this element.

```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) ] ) ]
```

```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) ] ) ]
```

```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) ] ) ]
```

```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) ] ) ]
```
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 (see 69.8.9) value of the character table.

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.

By definition of Order for arbitrary monoid elements, the determinantal order (see 70.8.18) of characters cannot be the return value of Order for characters. One can use Order( Determinant( chi ) ) to compute the determinantal order of the class function chi.

The default ViewObj (see 6.2.3) 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 70.8.1, 70.8.2), followed by the ViewObj output for the underlying character table (see 69.11), and the list of values. The table is chosen (and not the group) in order to distinguish class functions of different underlying characteristic (see 69.8.9).

The default PrintObj (see 6.2.3) method for class functions does the same as ViewObj, except that the character table is is Print-ed instead of View-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 12.8.2) to the underlying character table.

The default Display (see 6.2.4) method for a class function $chi$ calls Display for its underlying character table (see 69.11), with $chi$ as the only entry in the chars list of the options record.

```gap
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
```

### 70.6 Creating Class Functions from Values Lists

- **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 69.8.9).

- **VirtualCharacter** returns the virtual character (see 70.8.2) 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.

- **Character** returns the character (see 70.8.1) 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.

```gap
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, 5/8, 9/8, 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 ] )

ClassFunctionSameType( tbl, chi, values )

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 (see 70.6.1).

If chi is known to be a (virtual) character then ψ is also known to be a (virtual) character.

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 ] )

70.7 Creating Class Functions using Groups

TrivialCharacter( tbl )

TrivialCharacter( G )

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.

gap> TrivialCharacter( CharacterTable( "A5" ) );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
gap> TrivialCharacter( SymmetricGroup( 3 ) );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 1, 1, 1 ] )

NaturalCharacter( G )

NaturalCharacter( hom )

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 40.2.3), 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.

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> ),

\section*{Section 7. Creating Class Functions using Groups}
PermutationCharacter\( (G, D, \text{opr})\)

PermutationCharacter\( (G, U)\)

Called with a group \(G\), an action domain or proper set \(D\), and an action function \(\text{opr}\) (see Chapter 39), \texttt{PermutationCharacter} returns the permutation character of the action of \(G\) on \(D\) via \(\text{opr}\), 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 \(\text{opr}\).

If the arguments are a group \(G\) and a subgroup \(U\) of \(G\) then \texttt{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 \texttt{NaturalCharacter(\( G \))} can be used instead of \texttt{PermutationCharacter(\( G, U \))}.

More facilities concerning permutation characters are the transitivity test (see Section 70.8) and several tools for computing possible permutation characters (see 70.13, 70.14).

\begin{verbatim}
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 ] )
\end{verbatim}

### 70.8 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.

\begin{verbatim}
1\textgreater IsCharacter( [tbl, ]\chi )
\end{verbatim}

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 \(p\).

\begin{verbatim}
2\textgreater IsVirtualCharacter( [tbl, ]\chi )
\end{verbatim}

A virtual character is a class function that can be written as the difference of two proper characters (see 70.8.1).

\begin{verbatim}
3\textgreater IsIrreducibleCharacter( [tbl, ]\chi )
\end{verbatim}

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 70.8.5). For Brauer characters there is no generic method for checking irreducibility.
Section 8. Operations for Class Functions

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

4\> DegreeOfCharacter( chi )

is the value of the character \textit{chi} on the identity element. This can also be obtained as \textit{chi}[1].

\begin{verbatim}
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
\end{verbatim}

5\> ScalarProduct( [ tbl, ] chi, psi )

For two class functions \textit{chi} and \textit{psi} of the same ordinary character table \textit{tbl}, \texttt{ScalarProduct} returns the scalar product.

If \textit{G} is the underlying group of \textit{tbl} and the two class functions are \textit{\chi} and \textit{\psi} then this is defined as
\[
\frac{1}{|G|} \sum_{g \in G} \chi(g)\psi(g^{-1}).
\]

6\> MatScalarProducts( [ tbl, ] list1, list2 )

The first form returns the matrix of scalar products (see above) of the class functions in the list \textit{list1} with the class functions in the list \textit{list2}. More precisely, the matrix contains in the \textit{i}-th row the list of scalar products of \textit{list2[i]} with the entries of \textit{list1}.

The second form returns a lower triangular matrix of scalar products, containing for \((j \leq i)\) in the \textit{i}-th row in column \textit{j} the value \texttt{ScalarProduct(tbl, list[j], list[i])}.

7\> Norm( [ tbl, ] chi )

For an ordinary class function \textit{chi} of the group \textit{G}, say, we have \textit{chi} = \sum_{\chi \in Irr(G)} a_{\chi} \chi, with complex coefficients \(a_{\chi}\). The \texttt{norm} of \textit{chi} is defined as \(\sum_{\chi \in Irr(G)} a_{\chi} \overline{a_{\chi}}\).

\begin{verbatim}
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> chars:= Irr( tbl ){ [ 2 .. 4 ] };;
gap> chars:= Set( Tensored( chars, chars ) );
\end{verbatim}
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, 5 ] ]
gap> List( chars, Norm );
[ 2, 3, 3, 3, 3, 5 ]

8 ▶ ConstituentsOfCharacter( [ tbl ], chi )
is the set of irreducible characters that occur in the decomposition of the (virtual) character \( \text{chi} \) with nonzero coefficient.

gap> nat := NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> ConstituentsOfCharacter( nat );
[ Character( CharacterTable( S4 ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] ) ]

9 ▶ KernelOfCharacter( [ tbl ], chi )
For a class function \( \text{chi} \) of the group \( G \), say, \( \text{KernelOfCharacter} \) returns the normal subgroup of \( G \) that is formed by those conjugacy classes for which the value of \( \text{chi} \) equals the degree of \( \text{chi} \). If the underlying character table of \( \text{chi} \) does not store the group \( G \) then an error is signalled. (See 70.8.10 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 \( \text{chi} \).

10 ▶ ClassPositionsOfKernel( chi )
is the list of positions of those conjugacy classes that form the kernel of the character \( \text{chi} \), that is, those positions with character value equal to the character degree.

gap> List( Irr( S4 ), KernelOfCharacter );
[ Group([ [ () ], (1,2)(3,4), (1,2,3) ]), Group([()]),
  Group([ (1,2)(3,4), (1,4)(2,3) ]), Group([()]),
  Group([ (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ] ) ]
gap> List( Irr( S4 ), ClassPositionsOfKernel );
[ [ 1, 3, 4 ], [ 1 ], [ 1, 3 ], [ 1 ], [ 1, 2, 3, 4, 5 ] ]

11 ▶ CentreOfCharacter( [ tbl ], chi )
For a character \( \text{chi} \) of the group \( G \), say, \( \text{CentreOfCharacter} \) returns the centre of \( \text{chi} \), that is, the normal subgroup of all those elements of \( G \) for which the quotient of the value of \( \text{chi} \) by the degree of \( \text{chi} \) is a root of unity.
If the underlying character table of \( \text{psi} \) does not store the group \( G \) then an error is signalled. (See 70.8.12 for a way to handle the centre implicitly, by listing the positions of conjugacy classes in the centre.)

12 ▶ ClassPositionsOfCentre( chi )
is the list of positions of classes forming the centre of the character \( \text{chi} \) (see 70.8.11).
InertiaSubgroup([tbl,]G,chi)

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 70.2.1) is used instead. Furthermore, let \( G \) be a group that contains \( H \) as a normal subgroup.

\[ \text{InertiaSubgroup} \text{ returns the stabilizer in } G \text{ of } \chi \text{, w.r.t. the action of } G \text{ on the classes of } H \text{ via conjugation.} \]

For a permutation character \( \chi \) of the group \( G \) that corresponds to an action on the \( G \)-set \( \Omega \) (see 70.7.3), \text{IsTransitive} returns \text{true} if the action of \( G \) on \( \Omega \) is transitive, and \text{false} otherwise.

CentralCharacter([tbl,]chi)

For a character \( \chi \) of the group \( G \), say, \text{CentralCharacter} returns the \text{central character} of \( \chi \).

The central character of \( \chi \) is the class function \( \omega_\chi \) defined by \( \omega_\chi(g) = |g|^G \cdot \chi(g)/\chi(1) \) for each \( g \in G \).

DeterminantOfCharacter([tbl,]chi)

\text{DeterminantOfCharacter} returns the \text{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 \text{EigenvaluesChar} (see 70.8.19).
Chapter 70. Class Functions

It is also possible to call \texttt{Determinant} instead of \texttt{DeterminantOfCharacter}.

Note that the determinant character is well-defined for virtual characters.

\begin{verbatim}
    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 ] )
\end{verbatim}

\texttt{CentralCharacter( \text{TrivialCharacter}( S4 ) )};
\texttt{DeterminantOfCharacter( \text{Irr}( S4 )[3] )};

\begin{verbatim}
    EigenvaluesChar( [ tbl, ]chi, class )
\end{verbatim}

Let \texttt{chi} be a character of the group \textit{G}, say. For an element \texttt{g \in G} in the \texttt{class}-th conjugacy class, of order \texttt{n}, let \texttt{M} be a matrix of a representation affording \texttt{chi}.

\texttt{EigenvaluesChar( tbl, chi, class )} is the list of length \texttt{n} where at position \texttt{k} the multiplicity of \( E(n)^k = \exp(\frac{2\pi i k}{n}) \) as an eigenvalue of \texttt{M} is stored.

We have \texttt{chi[ class ] = List( [ 1 .. n ], k -> E(n)^k ) \ast EigenvaluesChar( tbl, chi, class )}.

It is also possible to call \texttt{Eigenvalues} instead of \texttt{EigenvaluesChar}.

\begin{verbatim}
    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 ] ]
\end{verbatim}

\begin{verbatim}
    Tensored( chars1, chars2 )
\end{verbatim}

Let \texttt{chars1} and \texttt{chars2} be lists of (values lists of) class functions of the same character table. \texttt{Tensored} returns the list of tensor products of all entries in \texttt{chars1} with all entries in \texttt{chars2}.

\begin{verbatim}
    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" ),
      Character( CharacterTable( "A5" ),
        [ 9, 1, 0, -1, -1 ] ),
      Character( CharacterTable( "A5" ),
\end{verbatim}

\texttt{70.9 Restricted and Induced Class Functions}

For restricting a class function of a group \textit{G} to a subgroup \textit{H} and for inducing a class function of \textit{H} to \textit{G}, the \texttt{class fusion} from \textit{H} to \textit{G} must be known (see 71.2).

If \textit{F} is the factor group of \textit{G} by the normal subgroup \textit{N} then each class function of \textit{F} can be naturally regarded as a class function of \textit{G}, with \texttt{N} in its kernel. For a class function of \textit{F}, the corresponding class function of \textit{G} is called the \texttt{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 \textit{G} and \textit{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 70.8.

1. $\text{RestrictedClassFunction}([tbl, \chi, H])$
2. $\text{RestrictedClassFunction}([tbl, \chi, \text{hom}])$
3. $\text{RestrictedClassFunction}([tbl, \chi, \text{subtbl}])$

For a class function $\chi$ of the group $G$, say, and either a subgroup $H$ of $G$ or a homomorphism from $H$ to $G$ or the character table $\text{subtbl}$ of this subgroup, $\text{RestrictedClassFunction}$ returns the class function of $H$ obtained by restricting $\chi$ to $H$.

In the situation that $\chi$ is a class function of a factor group $F$ of $H$, the variant where $\text{hom}$ is a homomorphism can be always used, the calls with argument $H$ or $\text{subtbl}$ work only if the factor fusion is stored on the character table.

1. $\text{RestrictedClassFunctions}([tbl, \chi, H])$
2. $\text{RestrictedClassFunctions}([tbl, \chi, \text{hom}])$
3. $\text{RestrictedClassFunctions}([tbl, \chi, \text{subtbl}])$

$\text{RestrictedClassFunctions}$ is similar to $\text{RestrictedClassFunction}$ (see 70.9.1), the only difference is that it takes a list $\chi$ of class functions instead of one class function, and returns the list of restricted class functions.

4. $\text{InducedClassFunction}([tbl, \chi, H])$
5. $\text{InducedClassFunction}([tbl, \chi, \text{hom}])$
6. $\text{InducedClassFunction}([tbl, \chi, \text{suptbl}])$

For a class function $\chi$ of the group $G$, say, and either a supergroup $H$ of $G$ or a homomorphism from $G$ to $H$ or the character table $\text{suptbl}$ of this supergroup, $\text{InducedClassFunction}$ returns the class function of $H$ obtained by inducing $\chi$ to $H$.

1. $\text{InducedClassFunctions}([tbl, \chi, H])$
2. $\text{InducedClassFunctions}([tbl, \chi, \text{hom}])$
3. $\text{InducedClassFunctions}([tbl, \chi, \text{suptbl}])$

$\text{InducedClassFunctions}$ is similar to $\text{InducedClassFunction}$ (see 70.9.3), the only difference is that it takes a list $\chi$ of class functions instead of one class function, and returns the list of induced class functions.
Chapter 70. Class Functions

```gap
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 ] ) ]
```

5► InducedCyclic( tbl )  O
6► InducedCyclic( tbl, "all" )  O
7► InducedCyclic( tbl, classes )  O
8► InducedCyclic( tbl, classes, "all" )  O

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 "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.

```gap
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 ] ) ]
```

70.10 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.

```gap
1► ReducedClassFunctions( [ tbl, ] constituents, reducibles )  O
2► ReducedClassFunctions( [ tbl, ] reducibles )  O
```

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 components remainders and irreducibles, both lists of virtual characters of $G$. These virtual characters are computed as follows.

Let $\text{rems}$ be the set of nonzero class functions obtained by subtraction of

$$\sum_{\chi} \frac{[\text{reducibles}[i], \chi]}{[\chi, \chi]} \cdot \chi$$

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 \texttt{rems} into the orthogonal space of \texttt{irrs} and all those irreducibles found this way until no new irreducibles arise. Then the \texttt{irreducibles} list is the set of all found irreducible characters, and the \texttt{remainders} list is the set of all nonzero remainders.

\texttt{ReducedCharacters} is similar to \texttt{ReducedClassFunctions}, the only difference is that \texttt{constituents} and \texttt{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 \texttt{constituents} does not exceed the degree of the character in \texttt{reducibles}.

\begin{verbatim}
gap> tbl:= CharacterTable( "A5" );;
gap> chars:= Irr( tbl ){ [ 2 .. 4 ] };;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> red:= ReducedClassFunctions( chars );;
rec( remainders := [ ], 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 ] ) ] )
\end{verbatim}

\texttt{IrreducibleDifferences} returns the list of irreducible characters which occur as difference of two elements of \texttt{reducibles} (if "triangle" is specified) or of an element of \texttt{reducibles} and an element of \texttt{reducibles2}. If \texttt{scprmat} is not specified then it will be calculated, otherwise we must have \texttt{scprmat = MatScalarProducts( tbl, reducibles, reducibles2 )}, respectively.

\begin{verbatim}
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 ] ) ]
\end{verbatim}

\texttt{LLL} calls the LLL algorithm (see 25.3.1) in the case of lattices spanned by the virtual characters \texttt{characters} of the ordinary character table \texttt{tbl} (see 70.8.5). By finding shorter vectors in the lattice spanned by \texttt{characters}, i.e., virtual characters of smaller norm, in some cases \texttt{LLL} is able to find irreducible characters.

\texttt{LLL} returns a record with at least components \texttt{irreducibles} (the list of found irreducible characters), \texttt{remainders} (a list of reducible virtual characters), and \texttt{norms} (the list of norms of the vectors in \texttt{remainders}). \texttt{irreducibles} together with \texttt{remainders} form a basis of the \texttt{Z}-lattice spanned by \texttt{characters}.

Note that the vectors in the \texttt{remainders} list are in general not orthogonal (see 70.10.1) to the irreducible characters in \texttt{irreducibles}.

Optional arguments of \texttt{LLL} are

- \texttt{y} controls the sensitivity of the algorithm, see 25.3.1,
Chapter 70. Class Functions

"sort"
LLL sorts characters and the remainders component of the result according to the degrees,

"linearcomb"
the returned record contains components irredecomp and reddecomp, which are decomposition matrices of irreducibles and remainders, with respect to characters.

```
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, 1, 1 ] ),
  Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
  Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, 1 ] ) ],
  remainders := [ ], norms := [ ]
)
```

Extract(tbl, reducibles, grammat[, missing])

Let tbl be an ordinary character table, reducibles a list of characters of tbl, and grammat the matrix of scalar products of reducibles (see 70.8.6). 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 components solution and choice, where the value of solution is a list [M₁,...,Mₙ] of decomposition matrices Mᵢ (up to permutations of rows) with the property that $Mᵢ \cdot X$ 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 Mᵢ corresponds to reducibles[j], and each row in Mᵢ corresponds to an irreducible character. Decreased (see 70.10.7) can be used to examine the solution for computable irreducibles.

```
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(
  solution := [ [ [ 1, 1, 0, 0, 2 ], [ 1, 0, 1, 0, 1 ], [ 0, 1, 0, 1, 0 ],
    [ 0, 0, 1, 0, 1 ] ], [ 0, 0, 0, 0, 1 ] ]),
  choice := [ [ 2, 5, 3, 4, 1 ] ]
)
gap> dec:= Decreased( s4, red, ext.solution[1], ext.choice[1] );
gap> Display( dec );
rec(
  irreducibles :=
    [ Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
    Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
  ]
```

OrthogonalEmbeddingsSpecialDimension is a variant of OrthogonalEmbeddings (see 25.4.1) 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 70.8.6), 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 25.4.1 for information why this may help.)

OrthogonalEmbeddingsSpecialDimension first uses OrthogonalEmbeddings (see 25.4.1) to compute all orthogonal embeddings of \( grammat \) into a standard lattice of dimension up to \( dim \), and then calls Decreased (see 70.10.7) in order to find irreducible characters of \( tbl \).

OrthogonalEmbeddingsSpecialDimension returns a record with components

\textbf{irreducibles}

a list of found irreducibles, the intersection of all lists of irreducibles found by Decreased, for all possible embeddings, and

\textbf{remainders}

a list of remaining reducible virtual characters.

\begin{verbatim}
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( vectors := [ [ -1, 0, 1, 0, 1, 0, 1, 0 ], [ 0, 1, 1, 0, 1, 0, 1, 0 ],
  [ 0, 1, 1, 0, 1, 0, 1, 0 ], [ 0, 1, 0, 0, 1, 0, 1, 0 ],
  [ 0, 0, -1, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 0 ] ],
  norms := [ 1, 1, 1, 1, 1, 1, 1, 1 ] )
\end{verbatim}
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 ] ]

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.

gap> SizeScreen([ 62, ]);;

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.

gap> SizeScreen([ 62, ]);;

gap> emb2:= OrthogonalEmbeddingsSpecialDimension( s6, rem, gram, 8 );;

Decreased

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 (see 70.10.5) or OrthogonalEmbeddings (see 25.4.1, 70.10.6); in the case of Extract, 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 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.

Decreased

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 (see 70.10.5) or OrthogonalEmbeddings (see 25.4.1, 70.10.6); in the case of Extract, 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 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.

Decreased

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 (see 70.10.5) or OrthogonalEmbeddings (see 25.4.1, 70.10.6); in the case of Extract, 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 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.
Section 10. Reducing Virtual Characters

rec( 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, 0, 1 ],
    [ 0, 0, 0, 1 ] ], norms := [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
solutions := [ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ],
gap> dec:= Decreased( s4, red, emb.vectors{ emb.solutions[1] } );;
gap> Display( dec );
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 ] ) ],
    remainders := [ ],
    matrix := [ ]
gap> Decreased( s4, red, emb.vectors{ emb.solutions[2] } );;
fail
gap> Decreased( s4, red, emb.vectors{ emb.solutions[3] } );
fail

DnLattice( tbl, grammat, reducibles )

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 70.8.6) 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.

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 );
gap> dn:= DnLattice( s4, gram, red );
gap> Display( dn );
rec( gram := [ ],
    remainders := [ ],
    irreducibles :=
        [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
          Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ].

8 ▶
Chapter 70. Class Functions

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 ] ) )

DnLatticeIterative( tbl, reducibles )

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 (see 70.10.4).

DnLatticeIterative was designed for iterative use of DnLattice (see 70.10.8). DnLatticeIterative selects the vectors of norm 2 among the given virtual character, calls DnLattice for them, reduces the virtual characters with found irreducibles, calls DnLattice 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 (see 70.10.4).

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 );; Display( dn );
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 ] ) ],
  remainders := [ ],
  norms := [ ] )

70.11 Symmetrizations of Class Functions

Symmetrizations( [ tbl, ]characters, n )
Symmetrizations( [ tbl, ]characters, Sn )

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 table of the symmetric group can be entered as Sn.

The symmetrization $\chi^{[\lambda]}$ of the character $\chi$ of tbl with the character $\lambda$ of the symmetric group $S_n$ of degree $n$ is defined by

$$\chi^{[\lambda]}(g) = \frac{1}{n!} \sum_{\rho \in S_n} \lambda(\rho) \prod_{k=1}^{n} \chi(g^k) a_k(\rho),$$

where $a_k(\rho)$ is the number of cycles of length $k$ in $\rho$.

For special kinds of symmetrizations, see 70.11.2, 70.11.3, 71.4.5 and 70.11.4, 70.11.5.

Note that the returned list may contain zero class functions, and duplicates are not deleted.
Section 11. Symmetrizations of Class Functions

\begin{verbatim}
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" ), [ 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" ), [ 8, 0, -1, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ) ]

2 ▶ SymmetricParts( tbl, characters, n )
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 70.11.1).

\begin{verbatim}
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 ] ) ]

3 ▶ AntiSymmetricParts( tbl, characters, n )
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 70.11.1).

\begin{verbatim}
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 ] ) ]

4 ▶ OrthogonalComponents( tbl, chars, m )
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 70.11.1).

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 70.8.1) 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
\end{verbatim}
\end{verbatim}
indicator 0 or −1, the result might contain virtual characters (see also 70.11.5), therefore the entries of the result do in general not know that they are characters.

\[
\text{gap> } \text{tbl}:= \text{CharacterTable}( "A8" );; \chi:= \text{Irr}( \text{tbl} )[2];
\text{Character( CharacterTable( "A8" ), [ 7, -1, 3, 4, 1, -1, 1, 2, 0, -1, 0, 0, -1, 1 ] )}
\text{gap> OrthogonalComponents( tbl, [ \chi ], 3 );}
[ \text{ClassFunction( CharacterTable( "A8" ), [ 21, -3, 1, 6, 0, 1, -1, 1, -2, 0, 0, 0, 1, 1 ] ), ClassFunction( CharacterTable( "A8" ), [ 27, 3, 7, 9, 0, -1, 1, 2, 1, 0, -1, -1, -1, -1 ] ), ClassFunction( CharacterTable( "A8" ), [ 105, 1, 5, 15, -3, 1, -1, 0, -1, 1, 0, 0, 0, 0 ] ), ClassFunction( CharacterTable( "A8" ), [ 35, 3, -5, 5, 2, -1, -1, 0, 1, 0, 0, 0, 0 ] ), ClassFunction( CharacterTable( "A8" ), [ 77, -3, 13, 17, 2, 1, 1, 2, 1, 0, 0, 0, 2, 2 ] ) ]
\]

5\text{SymplecticComponents( tbl, chars, m )}

If \(\chi\) is a (nonlinear) character with indicator −1, a splitting of the tensor power \(\chi^m\) 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 70.11.1).

\text{SymplecticComponents} returns the symplectic symmetrizations of the nonlinear characters of the character table \(\text{tbl}\) in the list \(\text{chars}\) up to the power \(m\), where \(m\) is an integer between 2 and 5.

\textbf{Note:} If \(\text{chars}\) is a list of character objects (see 70.8.1) then also the result consists of class function objects. It is not checked whether all characters in \(\text{chars}\) do really have indicator −1; if there are characters with indicator 0 or +1, the result might contain virtual characters (see also 70.11.4), therefore the entries of the result do in general not know that they are characters.

\[
\text{gap> } \text{tbl}:= \text{CharacterTable}( "U3(3)" );; \chi:= \text{Irr}( \text{tbl} )[2];
\text{Character( CharacterTable( "U3(3)" ), [ 6, -2, -3, 0, -2, -2, 2, 1, -1, -1, 0, 0, 1, 1 ] )}
\text{gap> SymplecticComponents( tbl, [ \chi ], 3 );}
[ \text{ClassFunction( CharacterTable( "U3(3)" ), [ 14, -2, 5, -1, 2, 2, 1, 0, 0, 0, -1, -1 ] ), ClassFunction( CharacterTable( "U3(3)" ), [ 21, 5, 3, 0, 1, 1, 1, -1, 0, 0, -1, 1, 1 ] ), ClassFunction( CharacterTable( "U3(3)" ), [ 64, 0, -8, -2, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 ] ), ClassFunction( CharacterTable( "U3(3)" ), [ 14, 6, -4, 2, -2, 2, 0, 0, 0, 0, 0, 0, 0, 0 ] ) ]
\]

\section{70.12 Molien Series}

\textbf{MolienSeries( psi )}
\textbf{MolienSeries( psi, chi )}
\textbf{MolienSeries( tbl, psi )}
\textbf{MolienSeries( tbl, psi, chi )}

The \textbf{Molien series} of the character \(\psi\), relative to the character \(\chi\), 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 \(\psi\) with the trivial character of the symmetric group \(S_d\) (see 70.11.2).
MolienSeries returns the Molien series of psi, relative to chi, where psi and chi must be characters of the same character table, which 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 (see 70.12.2). This admits the computation of coefficients of the series with ValueMolienSeries (see 70.12.3). 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 (see 70.12.4) to obtain the series with a prescribed denominator.

For more information about Molien series, see \[NPP84\].

```gap
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 )
```

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 \(\text{NumeratorOfRationalFunction}\)),
- **denom**: denominator of \(ratfun\) (in general a multiple of the denominator one gets by \(\text{NumeratorOfRationalFunction}\)),
- **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 \([[r_1, k_1], ..., [r_n, k_n]]\) such that \(denom = \prod_{i=1}^{n} (1 - z^{r_i})^{k_i}\).
- **summands**: a list of records, each with the components numer, r, and k, describing the summand \(\text{numer} / (1 - z^{r})^{k}\),
- **size**: the order of the underlying matrix group,
- **degree**: the degree of the underlying matrix representation.
Chapter 70. Class Functions

```gap
Chapter 70. Class Functions

gap> HasMolienSeriesInfo( mol );
true
gap> MolienSeriesInfo( mol );
rec( summands := [ rec( numer := [ -24, -12, -24 ], r := 5, k := 1 ),
  rec( numer := [ -20 ], r := 3, k := 1 ),
  rec( numer := [ -45/4, 75/4, -15/4, -15/4 ], r := 2, k := 2 ),
  rec( numer := [ -1 ], r := 1, k := 3 ),
  rec( numer := [ -15/4 ], r := 1, k := 1 ) ], size := 60, degree := 3,
  numer := 1-x_1^2-x_1^3+x_1^6+x_1^7-x_1^9,
  denom := 1-2*x_1^2-x_1^3+x_1^4+x_1^5+x_1^7+x_1^8-x_1^9-2*x_1^10+x_1^12,
  denominfo := [ 5, 1, 3, 1, 2, 2 ], numerstring := "1-z^2-z^3+z^6+z^7-z^9",
  denomstring := "(1-z^5)*(1-z^3)*(1-z^2)^2",
  ratfun := ( 1-z^2-z^3+z^6+z^7-z^9 ) / ( (1-z^5)*(1-z^3)*(1-z^2)^2 ) )
```

3 ▶ ValueMolienSeries( molser, i )

is the $i$-th coefficient of the Molien series $\text{series}$ computed by MolienSeries.

```gap
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 ]
```

4 ▶ MolienSeriesWithGivenDenominator( molser, list )

is a Molien series equal to $\text{molser}$ as rational function, but viewed as quotient with denominator $\prod_{i=1}^{n}(1-z^{r_i})$, where $\text{list} = [r_1, r_2, \ldots, r_n]$. If $\text{molser}$ cannot be represented this way, \text{fail} is returned.

```gap
gap> MolienSeriesWithGivenDenominator( mol, [ 2, 6, 10 ] );
( 1+z^15 ) / ( (1-z^5)*(1-z^3)*(1-z^2)^2 )
```

70.13 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 $\pi$ must have at least the following properties. (For details, see [Isa76], Theorem 5.18.)

(a) $\pi$ 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 $\psi$ as a constituent of $\pi$ 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) $\pi(g) \leq (|G| - \pi(1))/(|G||GalC(g)|)$ for all nonidentity $g \in G$, where $|GalC(g)|$ denotes the number of conjugacy classes of $G$ that contain generators of the group $g$,
(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.
Section 13. Possible Permutation Characters

(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 70.14.1), and also provides functions to check a few more criteria whether a given character can be a transitive permutation character (see 70.14.2).

Some information about the subgroup $U$ can be computed from the permutation character $(1_U)^G$ using PermCharInfo (see 70.13.1).

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 $\psi[i]|U|/\text{SizesCentralizers}(tbl)[i]$, 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 $|U|/\gcd(|U|,\text{SizesCentralizers}(tbl)[i])$, 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 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 $\text{ Irr}(tbl)$. 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 is the string "LaTeX"; if the third argument is the string "HTML" then exponentiation is denoted by $lett^{n}$.

```gap
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( contained := [ [ 1, 9, 0, 8, 6, 0, 0 ] ],
    bound := [ [ 1, 3, 8, 8, 6, 24, 24 ] ],
    display := rec( classes := [ [ 1, 2, 4, 5 ],
        chars := [ [ 15, 3, 0, 3, 1, 0, 0 ], [ 1, 9, 0, 8, 6, 0, 0 ],
            [ 1, 3, 8, 8, 6, 24, 24 ] ], letter := "I" ),
    ATLAS := [ "1a+5b+9a" ] )
gap> Display( t, info.display );
A6
```
Chapter 70. Class Functions

```
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, 0, 0 ] )
gap> PermCharInfo( j1, psi ).ATLAS;
[ "1a+56aabb+76aabb+77aabbbcc+120aabbbccc+133a^4bbcc+209a^5" ]
```

2▸PermCharInfoRelative( tbl, tbl2, permchars ) F

Let `tbl` and `tbl2` be the ordinary character tables of two groups \( H \) and \( G \), respectively, where \( H \) is of index 2 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` (see 70.13.1), 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 \( Na \) of `tbl` are denoted by \( Na^+ \) and \( Na^- \). The irreducible character of `tbl2` of degree 2\( N \), say, whose restriction to `tbl` is the sum of the irreducible characters \( Na \) and \( Nb \) is denoted as \( Nab \). Multiplicities larger than 1 of constituents are denoted by exponents.

(This format is useful mainly for multiplicity free permutation characters.)

```
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^-\{\pm\}a^-" ]
```
Section 14. Computing Possible Permutation Characters

70.14 Computing Possible Permutation Characters

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 record components of the optional argument record arec. 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 arec. 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).

In the first form PermChars returns the list of all possible permutation characters of the group with character table tbl. This list might be rather long for big groups, and its computation might take much time. The algorithm is described in Section 3.2 in [BP98]; 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 70.14.5). So the following commands compute the list of 39 possible permutation characters of the Mathieu group \( M_{11} \).

```
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, use the third form of PermChars and set the 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 if they are needed more than once.

In the second form PermChars returns the list of all possible permutation characters of tbl that have degree degree. 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 70.14.3). The algorithm is described at the end of Section 3.2 in [BP98]; 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.

```
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 ] ) ]
```

In the third form PermChars returns the list of all possible permutation characters that have the properties described by the argument record arec. One such situation has been mentioned above. If arec contains a degree as value of the record component degree then PermChars will behave exactly as in the second form.

```
gap> deg220 = PermChars( m11, rec( degree:= 220 ) );
true
```

For the meaning of additional components of arec besides degree, see 70.14.4.
Instead of degree, arec 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 arec.torso[1] must be an integer. In this case, the algorithm described in Section 3.3 in [BP98] 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 arec.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.)

```gap
gap> PermChars( m11, rec( torso:= [ 220 ] ) );
[ Character( m11, [ 220, 4, 4, 0, 0, 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 arec 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 ψ − π has N in its kernel. If the component chars is bound in arec then the above statements apply. An interpretation of the computed characters is the following. Suppose there exists a subgroup V of G such that π = (1V)G; then N ⊆ V, and if a computed character is of the form (1U)G for a subgroup U of G then V = UN.

```gap
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, TestPerm2, and TestPerm3. So they are possible permutation characters. See 70.14.2 for criteria whether a possible permutation character can in fact be a permutation character.

2▶ TestPerm1( tbl, char ) F
► TestPerm2( tbl, char ) F
► TestPerm3( tbl, chars ) F
► TestPerm4( tbl, chars ) F
► TestPerm5( tbl, chars, modtbl ) F

The first three of these functions implement tests of the properties of possible permutation characters listed in Section 70.13. 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
Section 14. Computing Possible Permutation Characters

TestPerm5 expect the characters in \texttt{chars} to satisfy the conditions checked by \texttt{TestPerm1} and \texttt{TestPerm2} (see below).

The return values of the functions were chosen parallel to the tests listed in [NPP84].

\textbf{TestPerm1} return 1 or 2 if \texttt{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 \texttt{tbl}. If \texttt{char} satisfies the conditions, 0 is returned.

\textbf{TestPerm2} returns 1 if \texttt{char} fails because of the criterion (c), it returns 3, 4, or 5 if \texttt{char} fails because of (T3), (T4), or (T5), respectively; these tests correspond to (g), a weaker form of (h), and (j). If \texttt{char} satisfies the conditions, 0 is returned.

\textbf{TestPerm3} returns the list of all those class functions in the list \texttt{chars} that satisfy criterion (h); this is a stronger version of (T6).

\textbf{TestPerm4} returns the list of all those class functions in the list \texttt{chars} that satisfy (T8) and (T9) for each prime divisor \texttt{p} of the order of \texttt{G}; these tests use modular representation theory but do not require the knowledge of decomposition matrices (cf. \texttt{TestPerm5} below).

(T8) implements the test of the fact that in the case that \texttt{p} divides \(|G|\) and the degree of a transitive permutation character \(\pi\) exactly once, the projective cover of the trivial character is a summand of \(\pi\). (This test is omitted if the projective cover cannot be identified.)

Given a permutation character \(\pi\) of a group \(G\) and a prime integer \(p\), the restriction \(\pi_B\) 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 \(|\pi_B(g^n)| \leq \pi_B(g^n) \leq \pi(g^n)\). (This is Corollary A on p. 113 of [Sco73].)

\textbf{TestPerm5} requires the \(p\)-modular Brauer table \texttt{modtbl} of \(G\), for some prime \(p\) dividing the order of \(G\), and checks whether those characters in the list \texttt{chars} whose degree is divisible by the \(p\)-part of the order of \(G\) can be decomposed into projective indecomposable characters; \textbf{TestPerm5} returns the sublist of all those characters in \texttt{chars} that either satisfy this condition or to which the test does not apply.

```gap
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, 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, 0, 0, 3, 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A7" ), [ 315, 15, 0, 0, 1, 0, 0, 0, 0 ] ) ]
gap> TestPerm4( tbl, perms );
```

Let $tbl$ be the ordinary character table of the group $G$. All $G$-characters $\pi$ satisfying $\pi(g) > 0$ and $\pi(1) = d$, for a given degree $d$, lie in a simplex described by these conditions. \texttt{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 \texttt{PermChars} (see 70.14.1) to construct all possible permutation characters (see 70.13) of a given degree. \texttt{PermChars} either calls \texttt{PermBounds} or takes this information from the \texttt{bounds} component of its argument record.

\texttt{PermComb} computes possible permutation characters of the character table $tbl$ by the improved combinatorial approach described at the end of Section 3.2 in [BP98].

For computing the possible linear combinations without prescribing better bounds (i.e., when the computation of bounds shall be suppressed), enter \texttt{arec := rec( degree := degree, bounds := false )}, where \texttt{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 \texttt{maxmult} component in \texttt{arec}.

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 \texttt{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, tries to keep the number of intermediate inequalities small by eventually changing the order of characters.

```gap
gap> tbl := CharacterTable( "M11" );;
gap> PermComb( tbl, rec( degree:= 110 ) );
[ Character( CharacterTable( "M11" ), [ 110, 6, 2, 0, 0, 0, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "M11" ), [ 110, 6, 2, 6, 0, 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "M11" ), [ 110, 14, 2, 0, 2, 0, 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, 0, 0, 0, 2, 0, 0, 0 ] ) ]
```
70.15 Operations for Brauer Characters

1 ▶ FrobeniusCharacterValue( value, p )

FrobeniusCharacterValue returns the value of the Frobenius character corresponding to the Brauer character value, \( p \) is the characteristic of the field.

The result can be described as the image of \( value \) under the ring homomorphism from \( \mathbb{Z}[\zeta] \) to the field with \( p^n \) elements, where \( \zeta = E(p^n) \) and \( value \) is contained in \( \mathbb{Z}[\zeta] \). This map is defined with the help of Conway polynomials (see 57.5.1), more information can be found in Sections 2–5 of [JLPW95].

If the result of \( \text{FrobeniusCharacterValue} \) cannot be expressed as an element of a finite field in \( \text{GAP} \) (see Chapter 57) then \( \text{FrobeniusCharacterValue} \) returns fail.

If the Conway polynomial of degree \( n \) is required for the computation but not yet stored in the global list of \( \text{GAP} \) then it is computed only if the global function \( \text{TryConwayPolynomialForFrobeniusCharacterValue} \) returns true when it is called with \( p \) and \( n \).

2 ▶ BrauerCharacterValue( mat )

For an invertible matrix \( mat \) over a finite field \( F \), \( \text{BrauerCharacterValue} \) 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.

\[
\text{gap> } g := \text{SL}(2,4) ;; \quad \# \text{2-dim. irreducible representation of A5}
\]
\[
\text{gap> } \text{ccl} := \text{ConjugacyClasses}( g );;
\text{gap> } \text{rep} := \text{List}( \text{ccl}, \text{Representative} );;
\text{gap> } \text{List}( \text{rep}, \text{Order} );
[ 1, 2, 5, 5, 3 ]
\]
\[
\text{gap> } \text{phi} := \text{List}( \text{rep}, \text{BrauerCharacterValue} );
\]
\[
\text{gap> } \text{List}( \text{phi} \{ [ 1, 3, 4, 5 ] \} , x \rightarrow \text{FrobeniusCharacterValue}( x, 2 ) );
[ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2)^0 ]
\]
\[
\text{gap> } \text{List}( \text{rep} \{ [ 1, 3, 4, 5 ] \} , \text{TraceMat} );
[ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2)^0 ]
\]

3 ▶ SizeOfFieldOfDefinition( val, p )

For a cyclotomic or a list of cyclotomics \( val \), and a prime integer \( p \), \( \text{SizeOfFieldOfDefinition} \) 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 \( E(q^d - 1) \) is mapped to the residue class of the indeterminate, modulo the ideal spanned by the Conway polynomial (see 57.5.1) 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.

4 ▶ RealizableBrauerCharacters( matrix, q )

For a list \( matrix \) of absolutely irreducible Brauer characters in characteristic \( p \), and a power \( q \) of \( p \), \( \text{RealizableBrauerCharacters} \) returns a duplicate-free list of sums of Frobenius conjugates of the rows of \( matrix \), each irreducible over the field with \( q \) elements.
Chapter 70. Class Functions

GAP supports groups, vector spaces, and algebras generated by class functions.

70.16 Domains Generated by Class Functions

GAP supports groups, vector spaces, and algebras generated by class functions.
Maps Concerning Character Tables

Besides the characters, power maps (see 71.1) are an important part of a character table. 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 71.2) must be known to get information about the embedding or simply to induce or restrict characters (see 70.9).

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, see 31.) In GAP, maps are represented by lists. Also each character, each list of element orders, centralizer orders, or class lengths are maps, and for a permutation perm of classes, ListPerm(perm) 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 71.3.

The implementation follows [Bre91], a description of the main ideas together with several examples can be found in [Bre99].

71.1 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 $p$ of $n$, so usually only power maps for primes $p$ 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 (see 71.1.1) can be used; if the $n$-th power map cannot be uniquely determined then PowerMap 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 (see 71.1.2). This provides a default strategy, the subroutines are listed in Section 71.4.
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.

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 value of tbl is known, see 69.8.5.)

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 (see 71.4.1).
3. For each character $\chi$ of G and each element $g$ in G, the values $\chi(g^p)$ and 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 $\lambda$ among these characters, the condition $\chi(g^p) = \chi(g)$ is checked. The corresponding function is Congruences (see 71.4.2).
4. For each character $\chi$ 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 $\chi$ specified below, the corresponding function is ConsiderKernels (see 71.4.3).
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 value (see 71.1.1) of tbl then this information is used. This criterion is checked in ConsiderSmallerPowerMaps (see 71.4.4).
Section 1. Power Maps

6. For each character \( \chi \) of \( G \), the symmetrization \( \psi \) defined by \( \psi(g) = (\chi(g)^p - \chi(g^p))/p \) is a character. This condition is checked for the kernels of all characters \( \chi \) specified below; the corresponding function is \text{PowerMapsAllowedBySymmetrizations} \ (\text{see 71.4.6}).

If \( tbl \) is a Brauer table, the possibilities are computed from those for the underlying ordinary table. The optional argument \( options \) must be a record that may have the following components:

- \text{chars}: a list of characters which are used for the check of the criteria 3., 4., and 6.; the default is \( \text{Irr} \ (\ tbl \ ) \).
- \text{powermap}: a parametrized map which is an approximation of the desired map
- \text{decompose}: a boolean; a \text{true} value indicates that all constituents of the symmetrizations of \text{chars} computed for criterion 6. lie in \text{chars}, so the symmetrizations can be decomposed into elements of \text{chars}; the default value of \text{decompose} is \text{true} if \text{chars} is not bound and \( \text{Irr} \ (\ tbl \ ) \) is known, otherwise \text{false}.
- \text{quick}: a boolean; if \text{true} then the subroutines are called with value \text{true} for the argument \text{quick}; especially, as soon as only one possibility remains this possibility is returned immediately; the default value is \text{false}.
- \text{parameters}: a record with components \text{maxamb}, \text{minamb} and \text{maxlen} which control the subroutine \text{PowerMapsAllowedBySymmetrizations}; it only uses characters with current indeterminateness up to \text{maxamb}, tests decomposability only for characters with current indeterminateness at least \text{minamb}, and admits a branch according to a character only if there is one with at most \text{maxlen} possible symmetrizations.

\[
gap \> tbl := \text{CharacterTable} \ (\ "U4(3).4" \ );;
\]
\[
\text{PossiblePowerMaps}( \ tbl, 2 ) ;
\]
\[
\begin{bmatrix}
1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4, 5, 6, 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18, 18, 20, 20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29
\end{bmatrix}
\]

\[
\begin{bmatrix}
1, 2, 3, 3, 4, 4, 5, 6, 6, 6, 6, \text{Unknown}(1), \text{Unknown}(2), 8, 9, 12, 2, 2, 4, 4, 6, 6, 6, 8, 10, 12, 12, 12, \text{Unknown}(3), \text{Unknown}(4), 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 12, 12, 12, 12, 12, 20, 20, 24, 24, \text{Unknown}(5), \text{Unknown}(6), \text{Unknown}(7), \text{Unknown}(8)
\end{bmatrix}
\]

\text{ElementOrdersPowerMap}( \ \text{powermap} \ )

Let \( \text{powermap} \) be a nonempty list containing at position \( p \), if bound, the \( p \)-th power map of a character table or group. \text{ElementOrdersPowerMap} returns a list of the same length as each entry in \( \text{powermap} \), with entry at position \( i \) equal to the order of elements in class \( i \) if this order is uniquely determined by \( \text{powermap} \), and equal to an unknown (see Chapter 19) otherwise.

\[
\gap \> tbl := \text{CharacterTable} \ (\ "U4(3).4" \ );;
\]
\[
\text{known} := \text{ComputedPowerMaps}( \ tbl ) ;;
\]
\[
\text{Length}( \ \text{known} \ );
\]
\[
7
\]
\[
\text{sub} := \text{ShallowCopy}( \ \text{known} \ ); \ \text{Unbind}( \ \text{sub}[7] \ );
\]
\[
\text{ElementOrdersPowerMap}( \ \text{sub} \ );
\]
\[
\begin{bmatrix}
1, 2, 3, 3, 4, 4, 5, 6, 6, 7, 7, 8, 9, 9, 12, 2, 2, 4, 4, 6, 6, 6, 8, 10, 12, 12, 14, 14, 4, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 10, 12, 12, 12, 12, 12, 12
\end{bmatrix}
\]
Chapter 71. Maps Concerning Character Tables

20, 20, 24, 24, 28, 28, 28, 28 ]
gap> ord = OrdersClassRepresentatives( tbl );
true

4 ▷ PowerMapByComposition( tbl, n )

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 list of tbl then PowerMapByComposition returns the n-th power map
of tbl. Otherwise fail is returned.

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 ]
gap> Length( ComputedPowerMaps( tbl ) );
7
gap> PowerMapByComposition( tbl, 11 );
fail
gap> PowerMap( 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 ]
The permutation group of matrix automorphisms (see 69.19.1) acts on the possible power maps returned
by PossiblePowerMaps (see 71.1.2) by permuting a list via Permutted (see 21.20.15) and then mapping the
images via OnPoints (see 39.2.1). Note that by definition, the group of table automorphisms acts trivially.

5 ▷ OrbitPowerMaps( map, permgrp )

returns the orbit of the power map map under the action of the permutation group permgrp via a combination
of Permutted (see 21.20.15) and OnPoints (see 39.2.1).

6 ▷ RepresentativesPowerMaps( listofmaps, permgrp )

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 Permutted (see 21.20.15) and OnPoints (see 39.2.1).

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, 11, 11, 11, 14, 14, 14, 17, 17, 17, 17, 4, 4,
  4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37, 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, 11, 11, 11, 14, 14, 14, 17, 17, 17, 17, 4, 4,
  4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37, 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, 11, 11, 11, 14, 14, 14, 17, 17, 17, 17, 4, 4,
  4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37, 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 ] ]
Section 2. 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 its 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 71.2.2 and 2.2.2 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, \texttt{FusionConjugacyClasses} (see 71.2.1) can be used; if the class fusion cannot be uniquely determined then \texttt{FusionConjugacyClasses} returns \texttt{fail}.

The list of all possible class fusion between two tables in the sense that certain necessary conditions are satisfied can be computed with \texttt{PossibleClassFusions} (see 71.2.6). This provides a default strategy, the subroutines are listed in Section 71.5.

It should be noted that all the following functions except \texttt{FusionConjugacyClasses} (see 71.2.1) deal only with the situation of class fusions from subgroups. The computation of \texttt{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 69.8.8) once the class positions of the kernel of the natural epimorphism have been fixed.
Called with two character tables \( tbl1 \) and \( tbl2 \), \texttt{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 \( G \), \texttt{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 69.6.2).

Called with a group homomorphism \( hom \), \texttt{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, \texttt{fail} is returned; this may happen when \texttt{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 \texttt{FusionConjugacyClassesOp}.

```gap
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 ]
gap> FusionConjugacyClasses( CharacterTable( "A5" ), CharacterTable( "J1" ) );
fail
gap> PossibleClassFusions( CharacterTable( "A5" ), CharacterTable( "J1" ) );
[ [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 5, 4 ] ]
```

The class fusions from the character table \( tbl \) that have been computed already by \texttt{FusionConjugacyClasses} (see 71.2.1) or explicitly stored by \texttt{StoreFusion} (see 71.2.4) are stored in the \texttt{ComputedClassFusions} list of \( tbl1 \). Each entry of this list is a record with the following components.

- **name**
  - the Identifier 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 and the maps entered by
StoreFusion if the table destination has a nonidentity ClassPermutation value. So if one fetches a fusion
map from a table tbl1 to a table tbl2 via access to the data in the ComputedFusionMaps list tbl1 then the
stored value must be composed with the ClassPermutation value of tbl2 in order to obtain the correct
class fusion. (If one handles fusions only via GetFusionMap and StoreFusion (see 71.2.3, 71.2.4) then this
adjustment is made automatically.)

Fusions are identified via the Identifier 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 71.2.3, 71.2.4. For accessing the identifiers of tables
that store a fusion into a given character table, see 71.2.5.

3

\texttt{GetFusionMap( source, destination )}
\texttt{GetFusionMap( source, destination, specification )}

For two ordinary character tables source and destination, GetFusionMap checks whether the Computed-
ClassFusion list of source (see 71.2.2) 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 71.2.1.

4

\texttt{StoreFusion( source, fusion, destination )}

For two character tables source and destination, StoreFusion stores the fusion fusion from source to
destination in the ComputedClassFusions list (see 71.2.2) of source, and adds the Identifier string of
destination to the NamesOfFusionSources list (see NamesOfFusionSources) 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 71.2.2.

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.

\texttt{gap> ComputedClassFusions( CharacterTable( d8 ) );}
\texttt{[ rec( name := "CT1", map := [ 1, 2, 3, 3, 5 ] ) ]}
\texttt{gap> Identifier( tbls4 );}
\texttt{"CT1"}
\texttt{gap> GetFusionMap( CharacterTable( d8 ), tbls4 );}
\texttt{[ 1, 2, 3, 3, 5 ]}
\texttt{gap> GetFusionMap( tbls4, tbls5 );}
\texttt{fail}
\texttt{gap> poss:= PossibleClassFusions( tbls4, tbls5 );}
Chapter 71. Maps Concerning Character Tables

[ [ 1, 5, 2, 3, 6 ] ]
gap> StoreFusion( tbls4, poss[1], tbls5 );
gap> GetFusionMap( tbls4, tbls5 );
[ 1, 5, 2, 3, 6 ]

5> NamesOfFusionSources( tbl )

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 (see 71.2.4).

gap> NamesOfFusionSources( tbls5 );
[ "2.A5.2", "A5", "S3x2", "3^{-1}+4.2^{-1}+4.s5", "s4", "5:4", "CT1" ]

6> PossibleClassFusions( subtbl, tbl[, options] )

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 (see 71.5.1).

2. The class fusion commutes with power maps. This is checked using TestConsistencyMaps (see 71.3.12).

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 (see 71.5.2).

4. The table automorphisms of tbl (see 69.8.8) 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 $\chi$ 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 (see 71.5.4).

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
da list of characters of tbl which are used for the check of 5.; the default is Irr( tbl ),

subchars
da 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 this possibility is returned immediately; the default value is false

parameters

a record with components maxamb, minamb and maxlen which control the subroutine FusionsAllowedByRestrictions; 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.

```gap
subtbl:= CharacterTable( "U3(3)" );;
tbl:= CharacterTable( "J4" );;
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 ] ]
```

The permutation groups of table automorphisms (see 69.8.8) of the subgroup table subtbl and the supergroup table tbl act on the possible class fusions returned by PossibleClassFusions (see 71.2.6), the former by permuting a list via Permuted (see 21.20.15), the latter by mapping the images via OnPoints (see 39.2.1).

If the 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. This may happen for example if a permutation character or if an explicit approximation of the fusion map is prescribed in the call of PossibleClassFusions.

7 OrbitFusions( subtblautomorphisms, fusionmap, tblautomorphisms )  

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.

8 RepresentativesFusions( subtblautomorphisms, listofmaps, tblautomorphisms )
  RepresentativesFusions( subtbl, listofmaps, tbl )  

returns a list of orbit representatives of class fusion maps in the list listofmaps under the action of maximal admissible subgroups of the table automorphisms subtblautomorphisms of the subgroup table and tblautomorphisms of the supergroup table. Both groups of table automorphisms must be permutation groups.

Instead of the groups of table automorphisms, also the character tables subtbl and tbl may be entered. In this case, the AutomorphismsOfTable values of the tables are used.

```gap
fus:= GetFusionMap( subtbl, tbl );
orb:= OrbitFusions( AutomorphismsOfTable( subtbl ), fus,
  AutomorphismsOfTable( tbl ) );
rep:= RepresentativesFusions( AutomorphismsOfTable( subtbl ), orb,
  AutomorphismsOfTable( tbl ) );
```
71.3 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 $\text{paramap}$ with the set of all maps $\text{map}$ with the property that either $\text{map}[i] = \text{paramap}[i]$ or $\text{map}[i]$ is contained in the list $\text{paramap}[i]$; we say then that $\text{map}$ is contained in $\text{paramap}$.

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 26) are not allowed in maps.

The composition of two parametrized maps $\text{paramap1}$, $\text{paramap2}$ is defined as the parametrized map $\text{comp}$ that contains all compositions $f_2 \circ f_1$ of elements $f_1$ of $\text{paramap1}$ and $f_2$ of $\text{paramap2}$. For example, the composition of a character $\chi$ 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 $\chi$ by elements of the parametrized fusion map.

CompositionMaps($\text{paramap2}$, $\text{paramap1}$) is a parametrized map with entry CompositionMaps($\text{paramap2}$, $\text{paramap1}$, $\text{class}$) at position $\text{class}$. If $\text{paramap1}[\text{class}]$ is an integer then CompositionMaps($\text{paramap2}$, $\text{paramap1}$, $\text{class}$) is equal to $\text{paramap2}[\text{paramap1}[\text{class}]]$. Otherwise it is the union of $\text{paramap2}[i]$ for $i$ in $\text{paramap1}[\text{class}]$.

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 ] ]

CompositionMaps($\text{paramap}$, $\text{InverseMap}$($\text{paramap}$)) returns a mutable parametrized map whose $i$-th entry is unbound if $i$ is not in the image of $\text{paramap}$, equal to $j$ if $i$ is (in) the image of $\text{paramap}[j]$ exactly for $j$, and equal to the set of all preimages of $i$ under $\text{paramap}$ otherwise.

We have CompositionMaps($\text{paramap}$, $\text{InverseMap}$($\text{paramap}$)) the identity map.

gap> tbl := CharacterTable( "2.A5" );;
gap> f := CharacterTable( "A5" );;
gap> fus := GetFusionMap( tbl, f );
[ 1, 1, 2, 3, 4, 4, 5, 5, 6, 6, 6, 6 ]

For a parametrized map $\text{paramap}$, InverseMap returns a mutable parametrized map whose $i$-th entry is unbound if $i$ is not in the image of $\text{paramap}$, equal to $j$ if $i$ is (in) the image of $\text{paramap}[j]$ exactly for $j$, and equal to the set of all preimages of $i$ under $\text{paramap}$ otherwise.

We have CompositionMaps($\text{paramap}$, InverseMap($\text{paramap}$)) the identity map.
Section 3. Parametrized Maps

3. ProjectionMap

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.

\[ \text{gap> ProjectionMap( [1, 1, 1, 2, 2, 2, 3, 4, 5, 5, 5, 6, 6, 6] );} \]
\[ [1, 4, 7, 8, 9, 12] \]

4. Indirected

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 19) otherwise.

\[ \text{gap> tbl:= CharacterTable( "M12" );} \]
\[ \text{gap> fus:= [1, 3, 4, [6, 7], 8, 10, [11, 12], [11, 12],} \]
\[ [14, 15];} \]
\[ \text{gap> List( Irr( tbl ){[1..6]}, x -> Indirected( x, fus ) );} \]
\[ [ [1, 1, 1, 1, 1, 1, 1, 1], [11, 3, 2, Unknown(9), 1, 0, Unknown(10), Unknown(11), 0, 0], [11, 3, 2, Unknown(12), 1, 0, Unknown(13), Unknown(14), 0, 0], [16, 0, -2, 0, 1, 0, 0, 0, Unknown(15), Unknown(16)], [16, 0, -2, 0, 1, 0, 0, 0, Unknown(17), Unknown(18)], [45, -3, 0, 1, 0, 0, -1, -1, 1, 1] ] \]

5. Parametrized

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 (see 71.3.6).

\[ \text{gap> Parametrized( [ [1, 2, 3, 4, 5], [1, 3, 2, 4, 5],} \]
\[ [1, 2, 3, 4, 6] ];} \]
\[ [1, [2, 3], [2, 3], 4, [5, 6]] \]

6. ContainedMaps

For a parametrized map paramap, ContainedMaps returns the set of all maps contained in paramap.

ContainedMaps is the inverse function to Parametrized (see 71.3.5) in the sense that Parametrized( ContainedMaps( paramap ) ) is equal to paramap.

\[ \text{gap> ContainedMaps( [1, [2, 3], [2, 3], 4, [5, 6]] );} \]
\[ [1, 2, 2, 4, 5], [1, 2, 2, 4, 6], [1, 2, 3, 4, 5], [1, 2, 3, 4, 6], [1, 3, 2, 4, 5], [1, 3, 3, 4, 5], [1, 3, 3, 4, 6] ] \]

7. UpdateMap

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.

```gap
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+E(7)^5+E(7)^6, 2, E(7)+E(7)^2+2*E(7)^3+E(7)^4+E(7)^5+E(7)^6, 2,
  E(7)+E(7)^2+2*E(7)^3+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)^3+E(7)^5+E(7)^6, 6, 1, 0, 0,
  0, 0, 0, -1, -1, 0, 0, 0, 0, -1, -1, 2, 0, 0,
  0, 0, 0, 0, 0, -1, -1, -10, -2, -2, -2, -1, 0, 1, 1, 0, 1 ] )
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, 0, 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, 0, 0, 0, 0, 0, -1, -1, -10, -2, -2, -2, -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 ] ]
```

8 ▶ MeetMaps( paramap1, paramap2 )

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.

```gap
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 );
true
[ 1, [ 3, 4 ], 5, 6, [ 8, 9 ] ]
```

9 ▶ CommutativeDiagram( paramap1, paramap2, paramap3, paramap4[, improvements] )

Let paramap1, paramap2, paramap3, paramap4 be parametrized maps covering parametrized maps f1, f2, f3, f4 with the property that CompositionMaps(f2, f1) is equal to CompositionMaps(f4, f3).

CommutativeDiagram checks this consistency, and changes the arguments such that all possible images are removed that cannot occur in the parametrized maps fi.

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 \( \text{imp1, imp2, imp3, imp4} \). 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, \texttt{CommutativeDiagram} immediately returns \texttt{fail}. Otherwise a record is returned that contains four lists \( \text{imp1, ..., imp4} \). \( \text{impi} \) is the list of classes where \( \text{paramap}_i \) was changed.

```gap
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 := [ ] )
```

### CheckFixedPoints(*inside1*, *between*, *inside2*)

Let \( \text{inside1}, \text{between}, \text{inside2} \) be parametrized maps, where \( \text{between} \) is assumed to map each fixed point of \( \text{inside1} \) (that is, \( \text{inside1}[i] = i \)) to a fixed point of \( \text{inside2} \) (that is, \( \text{between}[i] \) is either an integer that is fixed by \( \text{inside2} \) or a list that has nonempty intersection with the union of its images under \( \text{inside2} \)). \texttt{CheckFixedPoints} changes \( \text{between} \) and \( \text{inside2} \) by removing all those entries violate this condition.

When an inconsistency is detected, \texttt{CheckFixedPoints} immediately returns \texttt{fail}. Otherwise the list of positions is returned where changes occurred.

```gap
gap> subtbl:= CharacterTable( "L4(3).2_2" );;
gap> tbl:= CharacterTable( "O7(3)" );;
gap> fus:= InitFusion( subtbl, tbl );; fus{ [ 48, 49 ] };
[ [ 54, 55, 56, 57 ] ];
gap> CheckFixedPoints( ComputedPowerMaps( subtbl )[5], fus,
       ComputedPowerMaps( tbl )[5] );
[ 48, 49 ]
gap> fus{ [ 48, 49 ] };
[ [ 56, 57 ] ];
```

### TransferDiagram(*inside1*, *between*, *inside2*, *improvements*)

Let \( \text{inside1}, \text{between}, \text{inside2} \) be parametrized maps covering parametrized maps \( m_1, f, m_2 \) with the property that \( \text{CompositionMaps}(m_2, f) \) is equal to \( \text{CompositionMaps}(f, m_1) \).

\texttt{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 \texttt{TransferDiagram} is similar to \texttt{CommutativeDiagram} (see 71.3.9), but \( \text{between} \) occurs twice in each diagram checked.

If a record \( \text{improvements} \) with fields \( \text{impinside1}, \text{impbetween} \) and \( \text{impinside2} \) is specified, only those diagrams with elements of \( \text{impinside1} \) as preimages of \( \text{inside1} \), elements of \( \text{impbetween} \) as preimages of \( \text{between} \) or elements of \( \text{impinside2} \) as preimages of \( \text{inside2} \) are considered.

When an inconsistency is detected, \texttt{TransferDiagram} immediately returns \texttt{fail}. Otherwise a record is returned that contains three lists \( \text{impinside1, impbetween, impinside2} \) of positions where the arguments were changed.
Chapter 71. Maps Concerning Character Tables

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 );
[ 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( impinside1 := [ ], impbetween := [ 12, 23 ], impinside2 := [ ] )
gap> tr:= TransferDiagram( PowerMap( subtbl, 3 ), fus, PowerMap( tbl, 3 ) );
rec( impinside1 := [ ], impbetween := [ 14, 24, 25 ], impinside2 := [ ] )
gap> tr:= TransferDiagram( PowerMap( subtbl, 3 ), fus, PowerMap( tbl, 3 ), 
  tr );
rec( impinside1 := [ ], impbetween := [ ], 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 ]

12 > TestConsistencyMaps( powermap1, fusionmap, powermap2[, fus_imp] )
F

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 (see 71.3.11). 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 to these arguments for all i until no more changes occur.

If a list fus_imp is specified then only those diagrams with elements of fus_imp as preimages of fusionmap are considered.

When an inconsistency is detected, TestConsistencyMaps immediately returns false. Otherwise true is returned.

gap> subtbl:= CharacterTable( "2F4(2)" );; tbl:= CharacterTable( "Ru" );;
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar );
[ 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

13 > Indeterminateness( paramap )
F

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.
Section 3. Parametrized Maps

\texttt{gap> Indeterminateness([ 1, [ 2, 3 ], [ 4, 5 ], [ 6, 7, 8, 9, 10 ], 11 ]);}
   \texttt{20}

\textbf{PrintAmbiguity( list, paramap )}

For each map in the list \texttt{list}, \texttt{PrintAmbiguity} prints its position in \texttt{list}, the indeterminateness (see 71.3.13) of the composition with the parametrized map \texttt{paramap}, and the list of positions where a list of images occurs in this composition.

\texttt{gap> paramap:= [ 1, [ 2, 3 ], [ 3, 4 ], [ 2, 3, 4 ], 5 ];;}
\texttt{gap> list:= [ [ 1, 1, 1, 1, 1 ], [ 1, 1, 2, 2, 3 ], [ 1, 2, 3, 4, 5 ] ];;}
\texttt{gap> PrintAmbiguity( list, paramap );}

\texttt{1 1 [ ]}
\texttt{2 4 [ 2, 4 ]}
\texttt{3 12 [ 2, 3, 4 ]}

\textbf{ContainedSpecialVectors( tbl, chars, paracharacter, func )}

Let \texttt{tbl} be an ordinary character table, \texttt{chars} a list of class functions (or values lists), \texttt{paracharacter} a parametrized class function of \texttt{tbl}, and \texttt{func} a function that expects the three arguments \texttt{tbl}, \texttt{chars}, and a values list of a class function, and that returns either \texttt{true} or \texttt{false}.

\texttt{ContainedSpecialVectors} returns the list of all those elements \texttt{vec} of \texttt{paracharacter} that have integral norm, have integral scalar product with the principal character of \texttt{tbl}, and that satisfy \texttt{func( tbl, chars, vec )} = \texttt{true}.

Two special cases of \texttt{func} are the check whether the scalar products in \texttt{tbl} between the vector \texttt{vec} and all lists in \texttt{chars} are integers or nonnegative integers, respectively. These functions are accessible as global variables \texttt{IntScalarProducts} and \texttt{NonnegIntScalarProducts}, and \texttt{ContainedPossibleVirtualCharacters} and \texttt{ContainedPossibleCharacters} provide access to these special cases of \texttt{ContainedSpecialVectors}.

\texttt{gap> subtbl:= CharacterTable( "HSM12" );; tbl:= CharacterTable( "HS" );;}
\texttt{gap> fus:= InitFusion( subtbl, tbl );;}
\texttt{gap> rest:= CompositionMaps( Irr( tbl )[8], fus );}
\texttt{[ 231, [ -9, 7 ], [ -9, 7 ], [ -9, 7 ], 6, 15, 15, [ -1, 15 ], [ -1, 15 ], 1,}
\texttt{[ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ -2, 0 ], [ 1, 2 ], [ 1, 2 ],}
\texttt{[ 1, 2 ], 0, 0, 1, 0, 0, 0, 0 ]}
\texttt{gap> irr:= Irr( subtbl );;}
\texttt{gap> cont1:= ContainedSpecialVectors( subtbl, irr, rest,}
\texttt{    function( tbl, chars, vec ) return true; end );;}
\texttt{gap> Length( cont1 );}
\texttt{24}

\texttt{gap> # require scalar products to be integral}
\texttt{gap> cont2:= ContainedSpecialVectors( subtbl, irr, rest,}
\texttt{    IntScalarProducts );}
\texttt{[ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,}
\texttt{ 0, 0, 0, 0 ],}
\texttt{[ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,}
\texttt{ 0, 0, 0 ]},
\texttt{[ 231, 7, -9, -9, 6, 15, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,}
\texttt{ 0, 0, 0 ]],
\texttt{[ 231, 7, -9, -9, 6, 15, 15, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,}
Chapter 71. Maps Concerning Character Tables

\[
\begin{bmatrix}
0, 0, 0, 0 \\
[231, 7, -9, 7, 6, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0, 0, 0, 0]
\end{bmatrix}
\]

\text{gap} > \# \text{ additionally require scalar products to be nonnegative}
\text{gap} > \text{cont3} := \text{ContainedSpecialVectors( subtbl, irr, rest,}
\text{NonnegIntScalarProducts );}
\begin{bmatrix}
[231, 7, -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]
\end{bmatrix}
\text{gap} > \text{cont2} = \text{ContainedPossibleVirtualCharacters( subtbl, irr, rest );}
true
\text{gap} > \text{cont3} = \text{ContainedPossibleCharacters( subtbl, irr, rest );}
true

\textbf{16 ▶ CollapsedMat( mat, maps )}

is a record with components

\begin{itemize}
\item \textbf{fusion}
\begin{itemize}
\item fusion that collapses those columns of \textit{mat} that are equal in \textit{mat} and also for all maps in the list \textit{maps},
\end{itemize}
\item \textbf{mat}
\begin{itemize}
\item the image of \textit{mat} under that fusion.
\end{itemize}
\end{itemize}

\text{gap} > \text{mat} := \begin{bmatrix}
[1, 1, 1, 1], [2, -1, 0, 0], [4, 4, 1, 1]
\end{bmatrix};
\text{gap} > \text{coll} := \text{CollapsedMat( mat, [ ] );}
\text{rec( mat := \begin{bmatrix}
[1, 1, 1], [2, -1, 0], [4, 4, 1]
\end{bmatrix},}
\text{fusion := \begin{bmatrix}
1, 2, 3, 3
\end{bmatrix})}
\text{gap} > \text{List( last.mat, x \to x( last.fusion ) ) = mat;}
true
\text{gap} > \text{coll} := \text{CollapsedMat( mat, [ [1, 1, 1, 2] ] );}
\text{rec( mat := \begin{bmatrix}
[1, 1, 1, 1], [2, -1, 0, 0], [4, 4, 1, 1]
\end{bmatrix},}
\text{fusion := \begin{bmatrix}
1, 2, 3, 4
\end{bmatrix})}

\textbf{17 ▶ ContainedDecomposables( constituents, moduls, parachar, func )}

\textbf{F}

\textbf{F} \textbf{F}

\textbf{For these functions, let} \textit{constituents} \textbf{be a list of} \textit{rational} \textbf{class functions,} \textit{moduls} \textbf{a list of positive integers,} \textit{parachar} \textbf{a parametrized rational class function,} \textit{func} \textbf{a function that returns either} \textit{true} \textbf{or} \textit{false} \textbf{when called with (a values list of) a class function, and} \textit{tbl} \textbf{a character table.}

\textbf{ContainedDecomposables} \textbf{return}s the set of all elements \( \chi \) of \textit{parachar} \textbf{that satisfy} \( \text{func}(\chi) = \text{true} \) \textbf{and that lie in the} \( \mathbb{Z} \)-lattice spanned by \textit{constituents}, modulo \textit{moduls}. \textbf{The latter means they lie in the} \( \mathbb{Z} \)-lattice spanned by \textit{constituents} \textbf{and the set}

\[ \{ \text{moduls}[i] \cdot e_i; 1 \leq i \leq n \} , \]

\textbf{where} \( n \) \textbf{is the length of} \textit{parachar} \textbf{and} \( e_i \) \textbf{is the} \( i \)-th standard basis vector.

\textbf{One application of} \textbf{ContainedDecomposables} \textbf{is the following.} \textit{constituents} \textbf{is a list of (values lists of) rational characters of an ordinary character table} \textit{tbl}, \textit{moduls} \textbf{is the list of centralizer orders of} \textit{tbl} (see 69.8.6), \textbf{and} \textit{func} \textbf{checks whether a vector in the lattice mentioned above has nonnegative integral scalar product in} \textit{tbl} \textbf{with all entries of} \textit{constituents}. \textbf{This situation is handled by} \textbf{ContainedCharacters}. \textbf{Note that the entries of}
Section 4. Subroutines for the Construction of Power Maps

the result list are not necessary linear combinations of constituents, and they are not necessarily characters of tbl.

```gap
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 ], 6, 15, 15, [ -1, 15 ], [ -1, 15 ], 1,
[ 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, 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, 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, 0 ] ]
```

71.4 Subroutines for the Construction of Power Maps

\[\text{InitPowerMap}( tbl, \text{prime} )\]

For an ordinary character table \( tbl \) and a prime \( \text{prime} \), \text{InitPowerMap} returns a parametrized map that is a first approximation of the \( \text{prime} \)-th powermap of \( tbl \), using the conditions 1. and 2. listed in the description of \text{PossiblePowerMaps} (see 71.1.2).

If there are classes for which no images are possible, according to these criteria, then \text{fail} is returned.

```gap
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 ], 20, 20,
[ 9, 20 ], [ 9, 20 ], [ 9, 10, 20, 21, 22 ], [ 9, 10, 20, 21, 22 ],
[ 15, 25, 26, 40, 41, 42, 43 ], [ 15, 25, 26, 40, 41, 42, 43 ], [ 28, 29 ],
[ 28, 29 ], [ 28, 29 ], [ 28, 29 ] ]
```

In the argument lists of the functions \text{Congruences}, \text{ConsiderKernels}, and \text{ConsiderSmallerPowerMaps}, \( tbl \) is an ordinary character table, \( chars \) a list of (values lists of) characters of \( tbl \), \( \text{prime} \) 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, see 71.4.1), and quick a boolean.

The quick value true means that only those classes are considered for which approxmap lists more than one possible image.

\textbf{Congruences( tbl, chars, approxmap, prime, quick )}

Congruences replaces the entries of approxmap by improved values, according to condition 3. listed in the description of PossiblePowerMaps (see 71.1.2).

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.

\texttt{gap> Congruences( t, Irr( t ), pow, 2, false );}
\texttt{pow;}
\texttt{true}
\begin{verbatim}
[ 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, 17, 18, 18, [ 18, 19 ],
  [ 18, 19 ], 20, 20, 20, 20, 22, 22, 24, 24, [ 25, 26 ], [ 25, 26 ], 28, 28,
  29, 29 ]
\end{verbatim}

\textbf{ConsiderKernels( tbl, chars, approxmap, prime, quick )}

ConsiderKernels replaces the entries of approxmap by improved values, according to condition 4. listed in the description of PossiblePowerMaps (see 71.1.2).

Congruences returns true if the orders of the kernels of all characters in chars divide the order of the group of tbl, and false otherwise.

\texttt{gap> ConsiderKernels( t, Irr( t ), init, 2, false );}
\texttt{true}
\begin{verbatim}
[ 1, 1, 3, 4, 2, 6, 3, 8, 1, 1, 2, 3, [ 3, 4 ], 6, 7 ]
\end{verbatim}

\textbf{ConsiderSmallerPowerMaps( tbl, approxmap, prime, quick )}

ConsiderSmallerPowerMaps replaces the entries of approxmap by improved values, according to condition 5. listed in the description of PossiblePowerMaps (see 71.1.2).

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 69.8.5) then true is returned without any tests.

\texttt{gap> ConsiderSmallerPowerMaps( t, approx, 5, false );}
\texttt{true}
\begin{verbatim}
Indeterminateness( init );
4096
\end{verbatim}

\textbf{MinusCharacter( character, prime, powermap, prime )}

Let character be (the list of values of) a class function \( \chi \), prime a prime integer \( p \), and prime powermap a parametrized map that is an approximation of the \( p \)-th power map for the character table of \( \chi \). MinusCharacter returns the parametrized map of values of \( \chi^{p-} \), which is defined by \( \chi^{p-}(g) = (\chi(g)^p - \chi(g^p))/p \).
Section 4. Subroutines for the Construction of Power Maps

gap> tbl:= CharacterTable( "S7" );; pow:= InitPowerMap( tbl, 2 );;
gap> pow;
[ 1, 1, 3, 4, [ 2, 9, 10 ], 6, 3, 8, 1, [ 2, 9, 10 ], 3, [ 3, 4 ], 6, [ 7, 12 ] ]
gap> chars:= Irr( tbl ){ [ 2 .. 5 ] };;
gap> List( chars, x -> MinusCharacter( x, pow, 2 ) );;

6 ▶ PowerMapsAllowedBySymmetrizations( tbl, subchars, chars, approxmap, prime, parameters )

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, see 71.4.1), 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 or ContainedPossibleCharacters.

PowerMapsAllowedBySymmetrizations replaces the entries of approxmap by improved values, according to condition 6. listed in the description of PossiblePowerMaps (see 71.1.2).

More precisely, the strategy used is as follows.

First, for each \( \chi \in \text{chars} \), let \( \text{minus} := \text{MinusCharacter}(\chi, \text{approxmap}, \text{prime}) \).

- If \( \text{Indeterminateness}(\text{minus}) = 1 \) and \( \text{parameters}.\text{quick} = \text{false} \) then the scalar products of \( \text{minus} \) with \( \text{subchars} \) are checked; if not all scalar products are nonnegative integers then an empty list is returned, otherwise \( \chi \) is deleted from the list of characters to inspect.

- Otherwise if \( \text{Indeterminateness}(\text{minus}) \) is smaller than \( \text{parameters}.\text{minamb} \) then \( \chi \) is deleted from the list of characters.

- If \( \text{parameters}.\text{minamb} \leq \text{Indeterminateness}(\text{minus}) \leq \text{parameters}.\text{maxamb} \) then construct the list of contained class functions \( \text{poss} := \text{parameters}.\text{contained}(\text{tbl}, \text{subchars}, \text{minus}) \) and \( \text{Parametrized}(\text{poss}) \), and improve the approximation of the power map using \( \text{UpdateMap} \).

If this yields no further immediate improvements then we branch. If there is a character from \( \text{chars} \) left with less or equal \( \text{parameters}.\text{maxlen} \) 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 \( \text{chars} \) 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.

gap> tbl:= CharacterTable( "U4(3).4" );;
gap> pow:= InitPowerMap( tbl, 2 );;
gap> Congruences( tbl, Irr( tbl ), pow, 2 );;
gap> Congruences( tbl, Irr( tbl ), pow, 2 );;
gap> PowerMapsAllowedBySymmetrizations( tbl, \text{Irr}(\text{tbl})\), \text{Irr}(\text{tbl})\),
\text{> pow, 2, rec( maxlen:= 10, contained:= ContainedPossibleCharacters,}
\text{> minamb:= 2, maxamb:= infinity, quick:= false )});
\text{[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4, 5,}
\text{ 6, 7 ], 8, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18, 18, 18, 18, 18,}
\text{ 18, 20, 20, 22, 22, 24, 24, [ 25, 26 ], 28, 28, 29, 29 ]}
71.5 Subroutines for the Construction of Class Fusions

1 ▶ InitFusion( subtbl, tbl )

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 (see 71.2.6).

If there are classes for which no images are possible, according to this criterion, then fail is returned.

```
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 0, 5, 6, 6, 8, [ 13, 14, 15 ], [ 13, 14, 15 ], [ 18, 19 ], [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], [ 27, 28, 29 ], [ 27, 28, 29 ]]
```

2 ▶ CheckPermChar( subtbl, tbl, approxmap, permchar )

CheckPermChar replaces the entries of the parametrized map approxmap by improved values, according to condition 3. listed in the description of PossibleClassFusions (see 71.2.6).

CheckPermChar returns true if no inconsistency occurred, and false otherwise.

```
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, 28, 29 , [ 27, 28, 29 ]]
```

3 ▶ ConsiderTableAutomorphisms( approxmap, grp )

ConsiderTableAutomorphisms replaces the entries of the parametrized map approxmap by improved values, according to condition 4. listed in the description of PossibleClassFusions (see 71.2.6).

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.

```
gap> ConsiderTableAutomorphisms( fus, AutomorphismsOfTable( tbl ) );
gap> fus;
[ 16 ]
```

4 ▶ FusionsAllowedByRestrictions( subtbl, tbl, subchars, chars, approxmap, parameters )

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 or ContainedPossibleCharacters.

FusionsAllowedByRestrictions replaces the entries of approxmap by improved values, according to condition 5. listed in the description of PossibleClassFusions (see 71.2.6).

More precisely, the strategy used is as follows.
First, for each $\chi \in \text{chars}$, let \texttt{restricted} := $\text{CompositionMaps}(\chi, \text{approxmap})$.

- If \texttt{Indeterminateness(restricted)} = 1 and \texttt{parameters.quick} = false then the scalar products of \texttt{restricted} with \texttt{subchars} are checked; if not all scalar products are nonnegative integers then an empty list is returned, otherwise $\chi$ is deleted from the list of characters to inspect.
- Otherwise if \texttt{Indeterminateness} (\texttt{minus}) is smaller than \texttt{parameters.minamb} then $\chi$ is deleted from the list of characters.
- If \texttt{parameters.minamb} $\leq$ \texttt{Indeterminateness} (\texttt{restricted}) $\leq$ \texttt{parameters.maxamb} then construct \texttt{poss} := \texttt{parameters.contained} (\texttt{subtbl}, \texttt{subchars}, \texttt{restricted}) and \texttt{Parametrized} (\texttt{poss}) and improve the approximation of the fusion map using \texttt{UpdateMap}.

If this yields no further immediate improvements then we branch. If there is a character from \texttt{chars} left with less or equal \texttt{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 \texttt{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.

```gap
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 ], 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 ], 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, 6, 10, 12, 13, 14, 14, 21, 21 ],>
  [ 1, 2, 4, 4, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],>
  [ 1, 2, 4, 4, 6, 6, 10, 12, 13, 16, 16, 22, 22 ] ]
```
This chapter describes functions dealing with the monomiality of finite (solvable) groups and their characters. All these functions assume \texttt{characters} to be class function objects as described in Chapter 70, lists of character values are not allowed.

The usual \texttt{property tests} of \texttt{GAP} that return either \texttt{true} or \texttt{false} are not sufficient for us. When we ask whether a group character \( \chi \) has a certain property, such as quasiprimitivity, we usually want more information than just yes or no. Often we are interested in the reason \texttt{why} a group character \( \chi \) was proved to have a certain property, e.g., whether monomiality of \( \chi \) 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 \( \chi \) 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 \texttt{TestQuasiPrimitive} (see 72.2.3) contains the component \texttt{isQuasiPrimitive} (which is the known boolean property flag), and additionally the component \texttt{comment}, a string telling the reason for the value of the \texttt{isQuasiPrimitive} component, and in the case that the argument \( \chi \) was \texttt{not} quasiprimitive also the component \texttt{character}, which is an irreducible constituent of a nonhomogeneous restriction of \( \chi \) to a normal subgroup. Besides these test functions there are also the known properties, e.g., the property \texttt{IsQuasiPrimitive} which will call the attribute \texttt{TestQuasiPrimitive}, and return the value of the \texttt{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 \texttt{NormalSubgroups} or \texttt{MaximalNormalSubgroups}) 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 69.22.

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.

\begin{verbatim}
gap> S4:= SymmetricGroup( 4 );; SetName( S4, "S4" );
gap> Sl23:= SL( 2, 3 );;
\end{verbatim}

Most of the functions described in this chapter print some (hopefully useful) \texttt{information} if the info level of the info class \texttt{InfoMonomial} is at least 1 (see 7.4 for details).
## 72.1 Character Degrees and Derived Length

1 ▶ \texttt{Alpha( G )}

For a group \( G \), \texttt{Alpha} returns a list whose \( i \)-th entry is the maximal derived length of groups \( G/\ker(\chi) \) for \( \chi \in \text{Irr}(G) \) with \( \chi(1) \) at most the \( i \)-th irreducible degree of \( G \).

2 ▶ \texttt{Delta( G )}

For a group \( G \), \texttt{Delta} returns the list \( [ 1, \alpha[2] - \alpha[1], \ldots, \alpha[n] - \alpha[n-1] ] \), where \( \alpha = \text{Alpha}( G ) \) (see 72.1.1).

3 ▶ \texttt{IsBergerCondition( G )}

\( \texttt{IsBergerCondition( chi )} \)

Called with an irreducible character \( \chi \) of the group \( G \), \texttt{IsBergerCondition} returns \texttt{true} if \( \chi \) satisfies \( M' \leq \ker(\chi) \) for every normal subgroup \( M \) of \( G \) with the property that \( M \leq \ker(\psi) \) for all \( \psi \in \text{Irr}(G) \) with \( \psi(1) < \chi(1) \), and \texttt{false} otherwise.

Called with a group \( G \), \texttt{IsBergerCondition} returns \texttt{true} if all irreducible characters of \( G \) satisfy the inequality above, and \texttt{false} otherwise.

For groups of odd order the result is always \texttt{true} by a theorem of T. R. Berger (see [Ber76], Thm. 2.2). In the case that \texttt{false} is returned \texttt{InfoMonomial} tells about a degree for which the inequality is violated.

\begin{verbatim}
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, 3, 3 ]
\end{verbatim}

## 72.2 Primitivity of Characters

1 ▶ \texttt{TestHomogeneous( chi, N )}

For a group character \( \chi \) of the group \( G \), say, and a normal subgroup \( N \) of \( G \), \texttt{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

- \texttt{isHomogeneous} \( \text{true} \) or \texttt{false},

- \texttt{comment} \( \text{a string telling a reason for the value of the isHomogeneous component} \)
Chapter 72. Monomiality Questions

A character is irreducible constituent of the restriction, only bound if the restriction had to be checked.

The multiplicity of the character component in the restriction of $\chi_i$.

```gap
gap> n:= DerivedSubgroup( Sl23 );;
gap> chi:= Irr( Sl23 )[7];
Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] )
gap> TestHomogeneous( chi, n );
rec( isHomogeneous := false, comment := "restriction checked",
    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), Z(3)^0 ], [ Z(3)^0, Z(3)^0 ] ] ) ), [ 1, -1, -1, 1, 1 ] ),
    multiplicity := 1 )
gap> chi:= Irr( Sl23 )[4];
Character( CharacterTable( SL(2,3) ), [ 2, -2, 1, 1, -1, -1, 0 ] )
gap> cln:= ClassPositionsOfNormalSubgroup( CharacterTable( Sl23 ), n );
[ 1, 2, 7 ]
gap> TestHomogeneous( chi, cln );
rec( isHomogeneous := true, comment := "restricts irreducibly" )
```

For a character $\chi_i$ of the group $G$, say, IsPrimitiveCharacter returns true if $\chi_i$ is not induced from any proper subgroup, and false otherwise.

```gap
gap> IsPrimitive( Irr( Sl23 )[4] );
true
gap> IsPrimitive( Irr( Sl23 )[7] );
false
```

TestQuasiPrimitive returns a record with information about quasiprimitivity of the group character $\chi_i$, i.e., whether $\chi_i$ 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 $\chi_i$ is not quasiprimitive then there is additionally a component character, with value an irreducible constituent of a nonhomogeneous restriction of $\chi_i$.

IsQuasiPrimitive returns true or false, depending on whether the character $\chi_i$ is quasiprimitive.

Note that for solvable groups, quasiprimitivity is the same as primitivity (see 72.2.2).

```gap
gap> chi:= Irr( Sl23 )[4];
Character( CharacterTable( SL(2,3) ), [ 2, -2, 1, 1, -1, -1, 0 ] )
gap> TestQuasiPrimitive( chi );
rec( isQuasiPrimitive := true, comment := "all restrictions checked"
    )
gap> chi:= Irr( Sl23 )[7];
Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] )
gap> TestQuasiPrimitive( chi );
rec( isQuasiPrimitive := false, comment := "restriction checked",
    character := Character( CharacterTable( Group( 
```

P 3 ▶ IsQuasiPrimitive( chi )

A 2 ▶ IsPrimitiveCharacter( chi )

P 3 ▶ TestQuasiPrimitive( chi )

A 2 ▶ IsPrimitiveCharacter( chi )

P
Section 3. Testing Monomiality

\[
\begin{array}{ll}
[ [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ], \\
[ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ], \\
[ [ Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0 ] ] \}
\end{array}
\]

\text{TestInducedFromNormalSubgroup}( \text{chi}[, N] ) \quad \text{F}
\text{IsInducedFromNormalSubgroup}( \text{chi} ) \quad \text{P}

\text{TestInducedFromNormalSubgroup} returns a record with information whether the irreducible character \text{chi} of the group \text{G}, say, is induced from a proper normal subgroup of \text{G}. If the second argument \text{N} is present, which must be a normal subgroup of \text{G} or the list of class positions of a normal subgroup of \text{G}, it is checked whether \text{chi} is induced from \text{N}.

The result contains always the components \text{isInduced} (either \text{true} or \text{false}) and \text{comment} (a string telling a reason for the value of the component \text{isInduced}). In the \text{true} case there is a component \text{character} which contains a character of a maximal normal subgroup from which \text{chi} is induced.

\text{IsInducedFromNormalSubgroup} returns \text{true} if \text{chi} is induced from a proper normal subgroup of \text{G}, and \text{false} otherwise.

\text{gap} \text{> List( Irr( Sl23 ), IsInducedFromNormalSubgroup );}
\text{[ false, false, false, false, false, false, true ]}
\text{gap} \text{> List( Irr( S4 ){ [ 1, 3, 4 ] },}
\text{TestInducedFromNormalSubgroup );}
\text{[ rec( isInduced := false, comment := "linear character" ),}
\text{  rec( isInduced := true, comment := "induced from component '.character'",}
\text{  character := Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),}
\text{  [ 1, 1, E(3), E(3)^2 ] ) ),}
\text{rec( isInduced := false, comment := "all maximal normal subgroups checked" ) ]}

72.3 Testing Monomiality

A character \text{chi} of a finite group \text{G} is called \text{monomial} if \text{chi} is induced from a linear character of a subgroup of \text{G}. A finite group \text{G} is called \text{monomial} (or \text{M-group}) if each ordinary irreducible character of \text{G} is monomial.

There are \text{GAP} properties \text{IsMonomialGroup} (see 37.14.9) and \text{IsMonomialCharacter}, but one can use \text{IsMonomial} instead.

\text{gap} \text{> TestMonomial( \text{chi} )} \quad \text{A}
\text{gap} \text{> TestMonomial( \text{G} )} \quad \text{A}
\text{gap} \text{> TestMonomial( \text{chi}, uselattice )} \quad \text{O}
\text{gap} \text{> TestMonomial( \text{G}, uselattice )} \quad \text{O}

\text{Called with a group character} \text{chi} \text{of a group} \text{G}, \text{TestMonomial} \text{returns a record containing information about monomiality of the group} \text{G} \text{or the group character} \text{chi}, \text{respectively.}

\text{If TestMonomial} \text{proves the character} \text{chi} \text{to be monomial then the result contains components} \text{isMonomial} (\text{with value} \text{true}), \text{comment} (\text{a string telling a reason for monomiality}), \text{and if it was necessary to compute a linear character from which} \text{chi} \text{is induced, also a component} \text{character}.

\text{If TestMonomial} \text{proves} \text{chi} \text{or} \text{G} \text{to be nonmonomial then the value of the component} \text{isMonomial} \text{is} \text{false}, \text{and in the case of} \text{G} \text{a nonmonomial character is the value of the component} \text{character} \text{if it had been necessary to compute it.}

\text{A Boolean can be entered as the second argument} \text{uselattice}; \text{if the value is} \text{true} \text{then the subgroup lattice of the underlying group is used if necessary, if the value is} \text{false} \text{then the subgroup lattice is used only for groups of order at most} \text{TestMonomialUseLattice} (\text{see 72.3.2}). \text{The default value of} \text{uselattice} \text{is} \text{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.

```
gap> TestMonomial( S4 );
rec( isMonomial := true, comment := "abelian by supersolvable group" )
gap> TestMonomial( Sl23 );
rec( isMonomial := false, comment := "list Delta( G ) contains entry > 1" )
```

2► TestMonomialUseLattice

This global variable controls for which groups the operation TestMonomial (see 72.3.1) may compute the subgroup lattice. The value can be set to a positive integer or infinity, the default is 1000.

3► IsMonomialNumber( n )

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 \( \text{ord}(p, q) \) the multiplicative order of \( p \) (mod \( q \)).

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 \( \text{ord}(p, q) \) is even and \( \text{ord}(p, q) < \nu_p(n) \) (especially \( \nu_p(n) \geq 3 \)),
4. there is a prime divisor \( q \) of \( n \) such that \( \nu_2(n) \geq 2 \text{ord}(2, q) + 2 \) (especially \( \nu_2(n) \geq 4 \)),
5. \( \nu_2(n) \geq 2 \) and there is a \( p \) such that \( p \equiv 1 \pmod{4} \), \( \text{ord}(p, q) \) is odd, and \( 2 \text{ord}(p, q) < \nu_p(n) \) (especially \( \nu_p(n) \geq 3 \)).

These five possibilities correspond to the five types of solvable minimal nonmonomial groups (see 72.4.2) that can occur as subgroups and factor groups of groups of order \( n \).

```
gap> Filtered( [ 1 .. 111 ], x -> not IsMonomial( x ) );
[ 24, 48, 72, 96, 108 ]
```

4► TestMonomialQuick( chi )

TestMonomialQuick does some cheap tests whether the irreducible character chi resp. the group \( G \) 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`.

```gap
gap> TestMonomialQuick( Irr( S4 )[3] );
rec( isMonomial := true, comment := "whole group is monomial" )

gap> TestMonomialQuick( S4 );
rec( isMonomial := true, comment := "abelian by supersolvable group" )

gap> TestMonomialQuick( S123 );
rec( isMonomial := "?", comment := "no decision by cheap tests" )
```

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 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.

```gap
gap> TestSubnormallyMonomial( S4 );
rec( isSubnormallyMonomial := false,
    character := Character( CharacterTable( S4 ), [ 3, -1, -1, 0, 1 ] ),
    comment := "found non-SM character" )

gap> TestSubnormallyMonomial( Irr( S4 )[4] );
rec( isSubnormallyMonomial := false,
    comment := "all subnormal subgroups checked" )

gap> TestSubnormallyMonomial( DerivedSubgroup( S4 ) );
rec( isSubnormallyMonobials := true, comment := "all irreducibles checked" )
```

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 $\psi$ of $H$ such that the restriction of $\psi$ 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.

```gap
gap> IsSubnormallyMonomial( DerivedSubgroup( S4 ) );
true
gap> TestRelativelySM( DerivedSubgroup( S4 ) );
rec( isRelativelySM := true,
    comment := "normal subgroups are abelian or have nilpotent factor group" )
```

72.4 Minimal Nonmonomial Groups

A group $G$ is called **minimal nonmonomial** if it is nonmonomial, and all proper subgroups and factor groups are monomial.

```gap
gap> IsMinimalNonmonomial( S123 ); IsMinimalNonmonomial( S4 );
true
false
```

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].

```gap
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
```
GAP runs on a large number of different operating systems. It behaves slightly different on each of those. This chapter describes the behaviour of GAP, the installation, and the options on some of those operating systems.

Currently it contains sections for UNIX (see 73.3), which runs on an ever increasing number of machines, for Windows (see 73.8), which is one operating system for IBM PC compatibles and MacOS (see 73.12), which is the operating system on Apple Macintosh computers.

For other systems the section 73.16 gives hints how to approach such a port.
Section 73.17 contains information about the manual, where to find and how to print it.

73.1 Installation for the Impatient

Because GAP supports three different platforms, Windows, Macintosh and UNIX, the installation process might differ a bit from what you are used to. This document explains in detail which files to get and how to install the system and if you are inexperienced we strongly recommend you follow these instructions.

If you already have experience in installing GAP or would not read the installation instructions anyhow this section provides a quick run through of the installation on a UNIX system. (If it is incomprehensible or causes problems, please follow the explicit installation instructions given below!)

Get and unzip binary and installation archive(s) and the most recent bugfix file (including the most recent binary if you use Windows or Mac). The following instructions assume you have unzip and it is in your path. Call

```
unzoo -x gap4r3.zoo
cd gap4r3
```

If there is a bugfix (starting with release 4.3, bugfixes for a release will be comprehensive), extract it:

```
unzoo -x ../fix4r3nX.zoo
```

Then, to compile the GAP binary call

```
./configure
make
```

and you will get a shell script `bin/gap.sh` to start GAP which you can copy in a standard path, e.g. copy the script to say `/usr/local/bin` and rename it to how you expect users to call GAP (either `gap` or `gap4` is usual).

Send an email to gap@dcs.st-and.ac.uk, telling us about the installation. (This is just a courtesy; we like to know how many people are using GAP and get feedback regarding difficulties (hopefully none) that users may have had with installation.)

You might also want to unpack and install GAP packages. Generally, this involves first getting `accpkg4r3.zoo` and `deppkg4r3.zoo` (these have all the accepted/refereed and all the deposited but not yet refereed GAP packages, respectively) or visiting the GAP website again and first looking for the GAP Packages link and...
following through to the individual GAP package .zoo files you want. The package .zoo files should all be unzoo-ed in the pkg directory, i.e. do: cd pkg first. Some of the GAP packages are written entirely in GAP; these are essentially “ready-to-go” after unpacking, except they should be listed in the ALLPKG file (see 74.3). Other GAP packages have a C code component; these generally require one to do: ./configure; make in their respective directories.

Each GAP package has a README; it’s recommended that despite your impatience that you at least give these a quick glance; for one thing they will tell you whether the ./configure; make steps are needed.

Also note that a few details might have changed since the last release.

### 73.2 Getting GAP

GAP is distributed free of charge. You can obtain it via ftp and give it away to your colleagues. GAP is not in the public domain, however. In particular you are not allowed to incorporate GAP or parts thereof into a commercial product.

If you get GAP, we would appreciate it if you could notify us, e.g., by sending a short e-mail message to gap@dcs.st-and.ac.uk, containing your full name and address, so that we have a rough idea of the number of users. We also hope that this number will be large enough to convince various agencies that GAP is a project worthy of (financial) support. If you publish some result that was partly obtained using GAP, we would appreciate it if you would cite GAP, just as you would cite another paper that you used. (The copyright page of the manual gives a sample citation.) Again we would appreciate if you could inform us about such a paper.

The current release of GAP is version 4.3.

We distribute the full source for everything, the C code for the kernel, the GAP code for the library, and the TeX code for the manual. So it should be no problem to get GAP, even if you have a rather uncommon system. Of course, ports to non UNIX systems may require some work. Note that about 16 MByte of main memory (though at least 32 MByte is desirable) and about 30MB of disk space are required to run GAP. A full GAP installation, including all GAP packages and data libraries can use 200MB and more of disk space. GAP will compile on pentium (586) processors, though a faster machine is recommended.

We list and describe the extraction from .zoo archives below. As of GAP 4.3, however, we also supply the equivalent .zip and .tar.gz archives (simply download the archive type you prefer and use the corresponding extraction command).

(If you already downloaded an installation archive, you may ignore the rest of this section.)

The easiest way to get GAP for most users is probably via the World Wide Web. The main GAP Web site is found at

http://www.gap-system.org

There are two mirror sites updated automatically each night, at:

http://www.math.rwth-aachen.de/~GAP

http://www.ccs.neu.edu/mirrors/GAP

At these sites you can browse this manual, download the system and contributed extensions, read past postings to the GAP forum, and find out about authors of and contributors to GAP, publications that cited GAP and GAP related events.

GAP can also be obtained by anonymous ftp from (at least) the following servers.

ftp-gap.dcs.st-and.ac.uk:
School of Mathematical and Computational Sciences, University of St Andrews, Scotland, UK. Directory /pub/gap/gap4/.
ftp.math.rwth-aachen.de:
Lehrstuhl D für Mathematik, RWTH Aachen, Germany. Directory /pub/gap4/.

ftp.ccs.neu.edu:

ftp to the server closest to you, login as user ftp and give your full e-mail address as password. Remember when you transmit the files to set the file transfer type to binary image, otherwise you will only receive unusable garbage. Those servers will always have the latest version of GAP available.

The WWW page for the GAP distribution and the ftp directory contain the following files. Please check first which files you need, to avoid transferring those that you do not need. Except for the package archives all archives should be extracted at the “top” level level (i.e. they contain a gap4r3 directory).

INSTALL, INSTALL.WIN, INSTALL-MAC.TXT:
Installation files, one of which you are currently reading.

FILES:
More detailed description of the available files.

gap4r3.zoo:
This file contains the complete standard distribution of GAP version 4.3 (it does not include the packages contained in accpkg4r3.zoo and deppkg4r3.zoo).

basic4r3.zoo:
This file contains a minimal distribution of GAP version 4.3. (There are further archives with which you can “upgrade” it to the full version.)

gappc4r3.zoo:
This file contains a complete distribution for 32-bit Windows. It includes a precompiled binary.

gapmac4r3.zoo:
This file contains a complete distribution for Macintosh under MacOS. You will also need to obtain the binaries separately from the bin subdirectory via ftp or from the links on the Distribution web page.

accpkg4r3.zoo:
This file contains all accepted GAP packages as of the release date. It should be extracted in the pkg directory of a previously extracted GAP distribution or a user’s private pkg directory.

deppkg4r3.zoo:
This file contains all deposited, but not accepted, GAP packages whose authors wish to make them available, as of the release date. They should be viewed as analogous to preprints. It should be extracted in the pkg directory of a previously extracted GAP distribution or a user’s private pkg directory.

tools4r3.zoo:
This file contains additional files not provided in the standard distribution, mainly needed by package developers for the purpose of generating documentation.

util/unzoo.c:
A simple zoo archive extractor, which should be used to unpack the distribution. The bin subdirectory contains precompiled executables for common systems.

More files are available from the Distribution web page links or the following ftp subdirectories:

bin:
This directory contains executables for systems that do not come with a C compiler or where another C compiler produces a faster executable.
Chapter 73. Installing GAP

split:
This directory contains the complete distribution of GAP 4.3 in several smaller archives.

accpkg:
This directory contains all accepted GAP packages in case you don’t want to load the comprehensive accpkg archive.

deposit/pkg:
This directory contains all deposited but not yet accepted GAP packages whose authors wish to make them available, in case you don’t want to load the comprehensive deppkg archive.

73.3 GAP for UNIX

GAP runs very well under UNIX. In fact it is being developed on UNIX workstations (and PCs under Linux). GAP running on any UNIX machine should behave exactly as described in the manual.

GAP has successfully been compiled and installed on the following UNIX machines:

- DECalpha under OSF 3.2 with GNU cc 2 or cc,
- HP9000 under HP-UX with GNU cc 2,
- IBM PC under Linux with GNU cc 2,
- IBM RS/6000 & AIX with GNU cc 2, SGI Mips under IRIX 5/6 with gcc2 or cc, Sun Sparc under Solaris with GCC 2 or cc.

We hope that compiling and installing GAP on another UNIX machine does not pose any problem. If it does, please inform us of your problems (and, if possible, of your solution).

The section 73.4 describes how you install GAP on a UNIX machine. See Chapter 3 for information about command line options that GAP accepts under UNIX.

73.4 Installation of GAP for UNIX

Installing GAP on a UNIX machine is fairly easy. Get the files described in 73.2 and decide into which directory you want to install GAP 4.3. If you will be the only user using GAP, you probably should install it in your home directory. If other users will also use GAP, you should install it in a public place, such as /usr/local/lib/. GAP will be installed in a subdirectory gap4r3 of this directory. You can later move GAP to a different location. For example you can first install it in your home directory and when it works move it to /usr/local/lib/.

The following installation example assumes that you (as user you on the machine unix) are installing GAP into the directory /usr/local/lib on a Pentium Pro running Linux.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are not to be taken literally.

If you encounter problems please also see section 73.19 of this document.

Get the distribution gap4r3.zoo and the source for the zoo archive extractor unzoo.c. How you can get those files is described in the section 73.2. Remember that the distribution consists of binary files and that you must transmit them in binary mode.

Compile the zoo archive extractor unzoo with the command

```bash
you@unix> cc -o unzoo -DSYS_IS_UNIX -O unzoo.c
you@unix> ls -l unzoo
-rwxr-xr-x you 45056 Nov 3 10:11 unzoo
you@unix>
```

Now unpack the distribution with the command
Section 4. Installation of GAP for UNIX

you@unix> ./unzoo -x gap4r3.zoo
gap4r3/doc/aboutgap.tex -- extracted as text
gap4r3/doc/aggroup.tex -- extracted as text
[many more lines]
you@unix>

(Messages of the form: /bin/mkdir: cannot create directory are harmless and can be ignored. They stem from issuing mkdir calls for top level directories every time a new subdirectory is created.)

If you got not the full distribution file but several small files, extract all of them (**except the bug fixes and GAP packages**) in this way.

(Afterwards you will not need the file gap4r3.zoo any longer. If you are short of disk space you can remove it now.)

you@unix> rm gap4r3.zoo

Now go into the directory gap4r3. If there is a bug fix (there are none in the initial release but invariably some follow later and are deposited in the bugfixes directory on our ftp server(s). Starting with release 4.3, bugfixes will be comprehensive and you need only the most recent bugfix), extract it while in the gap4r3 directory. (The bug fixes extract only on this level of the directory hierarchy to make them applicable even if you chose later to rename the root directory of your GAP distribution.)

The program may ask you for confirmation to overwrite files, you should give this permission.

you@unix> cd gap4r3
you@unix> ../unzoo -x ../fix4r3n3.zoo

If you got any GAP packages, extract them in the pkg subdirectory in the same way. For example if you got the accpkg4r3.zoo and deppkg4r3.zoo archives, you would issue:

you@unix> cd pkg
you@unix> ../../unzoo -x ../../accpkg4r3.zoo
you@unix> ../../unzoo -x ../../deppkg4r3.zoo
you@unix> cd ..

There is a little more to do to install the GAP packages but (particularly for those packages with a C code component) it’s only appropriate to complete each GAP package’s installation after GAP has been compiled.

The directories trans, small and prim contain data libraries. If you are short of disk space you can erase some of them, and any of the GAP package directories in pkg that you have decided you don’t need, but then of course you will not be able to access these data or packages.

Under UNIX, we now use the “autoconfig” method to take care of system dependencies. (See 73.16 if this is not working on your machine.)

If you are installing GAP on various systems that share a file system then all the above steps have essentially been done for all but the first system and on succeeding systems these steps are replaced by (fairly obviously) changing to the gap4r3 directory. However, before proceeding with the following steps, on each other machine you **must** first remove the files config.cache and config.status created on the previous machine!

Now run the shell script configure:

you@unix> ./configure

checking host system type... i686-unknown-linux2.0.27
[many more lines]

This will automatically detect the machine and compiler target. The process will automatically create an appropriate binary subdirectory. It also creates a Makefile that will be used to compile GAP on this machine. Simply call make:
you@unix> make
[many lines of further configuration and compilation]

The compilation process creates the object files and the executable in the directory

/usr/local/lib/gap4r3/bin/target/

where target is the name printed by the first configure (and /usr/local/lib the directory in which you installed GAP). In our example the executable will be created as

/usr/local/lib/gap4r3/bin/i686-unknown-linux2.0.27/gap

If configure or make fails, see section 73.5 for a description of some remedies. If all of these fail, please inform us at gap-trouble@dcs.st-and.ac.uk.

If you have UNIX experience you might want to change the compilation options to obtain a better performance. Section 73.7 explains how to do this. In particular, if you have little system memory, you might want to remove debugging information from the binary to create a smaller file. This is done by the UNIX strip command which gets invoked by:

you@unix> make strip

In order to test your compilation now run the newly created executable. You should get the GAP banner and then the GAP prompt. (The process of starting GAP may take a little while.)

you@unix> cd ..
you@unix> bin/i686-unknown-linux2.0.27/gap -l /usr/local/lib/gap4r3
[... lines deleted]
gap>

(Having to give the library path with the -l option each time would be a bit tedious. Below we will therefore install a shell script to start GAP.)

Try a few things to see if the compilation succeeded.

```
gap> 2 * 3 + 4;
gap> Factorial( 30 );
2652528598121910586363084800000000
gap> Size( SymmetricGroup( 10 ) );
3628800
gap> Factors( SymmetricGroup( 10 ) );
[ 2, 2, 2, 2, 2, 2, 3, 3, 5, 5, 7 ]
gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));
7920
gap> Size( m11 );
7920
gap> Length( ConjugacyClasses( m11 ) );
10
```

Especially try the command line editing and history facilities, because they are probably the most machine dependent feature of GAP. Enter a few commands and then make sure that $\texttt{ctr-P}$ redisplay the last command, that $\texttt{ctr-E}$ moves the cursor to the end of the line, that $\texttt{ctr-B}$ moves the cursor back one character, and that $\texttt{ctr-D}$ deletes single characters. So, after entering the above commands, typing

```
$\texttt{ctr-P} \texttt{ctr-P} \texttt{ctr-E} \texttt{ctr-B} \texttt{ctr-B} \texttt{ctr-B} \texttt{ctr-D} \texttt{1} \texttt{return}
```

should give the following lines:
If you can compile but command line editing does not work you can always start GAP with option -n to disable command line editing. In such a case however we would like to hear about such problems.

If your operating system has job control, make sure that you can still stop GAP, which is usually done by pressing \texttt{ctr-Z}.

The \texttt{make} process should also have created a shell script \texttt{gap.sh} in the \texttt{bin} subdirectory. This file already contains proper directory paths and it should be possible to start GAP by simply calling this script. Still, you might want to edit it to give further default command line options or modify the default memory.

Execute the script to start GAP again

\begin{verbatim}
you@unix> bin/gap.sh
\end{verbatim}

If the shell complains that it cannot execute or find \texttt{gap.sh} check the permissions of the file by doing a \texttt{ls -l bin/gap.sh}. The permissions should include execute permissions, if they don’t you can set them by

\begin{verbatim}
you@unix> chmod +x bin/gap.sh
\end{verbatim}

Then copy this script into a directory in your search path, for example \texttt{/usr/local/bin/} as \texttt{gap} (or \texttt{gap4} if you also have GAP 3 running on the same machine). (If you are using the C-shell, you will also have to \texttt{rehash}, so that the C-shell adds \texttt{gap} to its internal tables). When you later move GAP to another location you only need to edit this script.

There also is a shell script \texttt{gac} in the same place that can be used to invoke the compiler. Depending on your installation you also might want to copy it in a directory in your search path.

Now you should be able to start GAP by its name:

\begin{verbatim}
you@unix> cd ~
you@unix> gap
\end{verbatim}

(If you get an error message “hmm, I cannot find lib/init.g” you are starting the pure binary. It needs to find the library which is given via the command line parameter \texttt{-l}.)

A set of test files is provided, running them all takes some 40 minutes on a Pentium Pro 200 MHz. As a quick test we start \texttt{combinat.tst} first.

\begin{verbatim}
gap> Filename( DirectoriesLibrary("tst"), "combinat.tst" );
"/usr/local/lib/gap4r3/tst/combinat.tst"
gap> ReadTest(last);
+ GAP4stones: 27801
+ true
\end{verbatim}

The number given as \texttt{GAP4stones} will vary, depending on the speed of your machine.

Now use \texttt{Read} \texttt{(not ReadTest)} to read the file \texttt{testall.g} to run all available test files. This is not a necessary part of the installation, it only serves as a confirmation that everything went OK. The full test suite takes some time (almost 2 hours on a Pentium III/500) and uses quite a bit of memory (around 100MB), so you may wish to skip this step or run only part of the tests. This does no harm.
Chapter 73. Installing GAP

```gap
gap> Filename( DirectoriesLibrary("tst"), "testall.g" );
"/usr/local/lib/gap4r3/tst/testall.g"
gap> Read(last);
[lines omitted]
test file      GAP4stones time(msec)
-------------------------------------------
testing: ./tst/boolean.tst
boolean.tst  0  0
testing: ./tst/unknown.tst
unknown.tst  0  10
testing: ./tst/gaussian.tst
gaussian.tst  0  250
[further lines omitted]
```

You can ignore warnings from `weakptr.tst`, which stem from garbage collections occurring at different times, and those from `grpconst.tst` which stem from differences in available GAP packages.

The information about the manual is system independent, you can find it in section 73.17.

Now you should finish the installation for the GAP packages you intend to install. First do:

```shell
you@unix> cd pkg
```

Each GAP package has a README, which details the additional steps necessary for installation of the package. Some of the GAP packages are written entirely in GAP; for these once you have installed them on one system there is no further work required on other systems sharing the same file system. For the packages with a C code component, generally one must do `./configure` and `make` after changing to the package’s directory, e.g. for the ACE package you usually only need to do:

```shell
you@unix> cd ace
you@unix> ./configure
you@unix> make
you@unix> cd..
```

For other systems sharing the same file system you need to repeat these steps for each system; the appropriate time is immediately after compiling GAP on each system.

Some GAP packages are set up to load (or provide the documentation) automatically. To enable this, after installing all the packages you think you will need, you should list all the GAP packages you have installed in a file `ALLPKG` in the `pkg` directory (see Section 74.3 in the Reference Manual for details); an easy way to do this under UNIX is by issuing the command

```shell
find * -type d -maxdepth 0 -print > ALLPKG
```

while still in the `pkg` directory.

Please see Chapter 74 for more details on the installation of GAP packages.

If you want to install GAP also on different architectures, which share the same file system with the machine on which you just installed GAP log into these machines and go to the GAP home directory. You do not need to extract any new files, simply execute the same “configure/make” process which has been described above again (but remember to remove the files `config.cache` and `config.status`, each time). Make sure also, however, that you copy the `bin/gap.sh` and `bin/gac` shell scripts created by the `make` process, each time, because they are system dependent and will be overwritten when compiling on each new system.

A few final reminders:

- Make sure that you got and installed the most recent bugfix.
Section 6. Problems on Particular Systems

We would appreciate after installation your sending us a short note at gap@dcs.st-and.ac.uk (even if you have installed GAP 3 before). Generally, we do not reply to such emails; we only use them to gain some idea of how many people use GAP and of the machines/operating systems on which GAP has been successfully installed.

We also suggest that you subscribe to our gap-forum mailing list; see the GAP web pages for details. Whenever there is a bug fix or new release of GAP this is where it is announced. The gap-forum also deals with user questions of a general nature; bug reports and other problems you have while installing and/or using GAP should be sent to gap-trouble@dcs.st-and.ac.uk.

That’s all, your installation should be complete. We hope that you will enjoy using GAP. Remember, if you have problems, do not hesitate to contact us at gap-trouble@dcs.st-and.ac.uk. See Section 73.19 for what to include in a bug report.

73.5 Known Problems of the Configure Process

If make complains “Do not know how to make xyz” but xyz is an existing file, it is likely that the dates of the files were not extracted properly (Alpha-OSF machines are prone to this). Call
touch * cnf/* src/*
from the main GAP directory (this ought to reset the date of all relevant files to “now”) and try again.

Sometimes the configure process does not properly figure out the “inline” compiler command. If you get error messages that complain that “inline” is unknown, edit the file config.h in the bin/target subdirectory and replace the line

/* #undef inline */

by

#define inline

and then try to compile again.

The configure script respects compiler settings given in environment variables. However such settings may conflict with the automatic configuration process. If configure produces strange error messages about not being able to run the compiler, check whether environment variables that might affect the compilation (in particular CC, LD, CFLAGS, LDFLAGS and C_INCLUDE_PATH) are set and reset them using unsetenv.

Some users reported problems with make, while the GNU version gmake worked. Thus if problems occur you should try gmake instead if it is installed on your machine.

73.6 Problems on Particular Systems

The highest levels of optimization of the OSF/4 C compiler cc on the Compaq alpha chip make assumptions about the use of pointers which are not valid for GAP, and produce executables that can crash; -O3 seems to be safe, but -O4 and -fast are not.

On Sun and IRIX systems which are capable of running in 32 or 64 bit modes, it is possible to build a 64 bit version of GAP, but special procedures are needed (and, on Suns, a compiler bug must be circumvented). If you wish to compile on such a system, please send an email to gap-trouble@dcs.st-and.ac.uk.
73.7 Optimization and Compiler Options

Because of the large variety of different versions of UNIX and different compilers it is possible that the `configure` process will not choose best possible optimization level, but you might need to tell `make` about it. If you want to compile GAP with further compiler options (for example specific processor optimizations) you will have to assign them to the variable `COPTS` as in the following example when calling `make`:

```
make COPTS=-option
```

If there are several compiler options or if they contain spaces you might have to enclose them by quotes to avoid depending on the shell you are using.

The `configure` process also introduces some default compiler options. (See the `Makefile` in the `bin` directory for details.) You can eliminate these by assigning the variable `CFLAGS` (which contains the default options and `COPTS`) to the desired list of compiler options in the same way as you would assign `COPTS`.

The recommended C compiler for GAP is the GNU C compiler `gcc`, or a related compiler such as `egcs`. There are two reasons for this recommendation: firstly we use `gcc` in GAP development and so this combination has been far more heavily tested than any other and secondly, we have found that it generally produces code which is faster than that produced by other compilers.

If you do wish to use another compiler, you should remove `config.cache` and `config.status` in the GAP root directory, set the environment variable `CC` to the name of your preferred compiler and then rerun `configure` and `make`. You may have to experiment to determine the best values for `CFLAGS` and/or `COPTS` as described above. Please let us (gap-trouble@dcs.st-and.ac.uk) know the results of your experiments.

73.8 GAP for Windows

It is possible to compile GAP for 32-bit Windows using the Cygnus `cygwin32` implementation of GCC. We provide such a binary, and have tested that runs under the English version of Windows 98. It is likely that it will also work under later versions such as Windows NT and Windows XP.

It is likely that features of GAP like `Process`, `Exec` or the compiler, that rely on a UNIX environment will not work under Windows.

The following sections contain information about GAP that is specific to this port of GAP (simply called GAP for Windows below).

To run GAP under Windows you need an IBM PC compatible with an Intel Intel 80486, or Intel Pentium processor or better. The system must have at least 16 MByte of main memory and a harddisk. The operating system must be Windows 95 or better.

The section 73.9 describes the copyright as it applies to the executable version that we distribute. The section 73.10 describes how you install GAP for Windows.

The compiler we are using to create the Windows binary (Cygnus GNUWin32) provides a C-library that emulates all standard UNIX commands. There are however a few low-level routines that have to be called system-specific and for which we have not yet found out how to do it under Windows. We will try to fix these in a future version. If you have experience in Windows programming and know how to do such things we would very much welcome help.
73.9 Copyright of GAP for Windows

In addition to the general copyright for GAP set forth in the Copyright the following terms apply to GAP for Windows.

The executable of GAP for Windows that we distribute was compiled with The gnuwin32 compiler of the cygwin package. This compiler can be obtained by anonymous ftp from a variety of general public FTP archives. Many thanks to the Free Software Foundation and Cygnus Support for this amazing piece of work.

The GNU C compiler is

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

under the terms of the GNU General Public License (GPL). Note that the GNU GPL states that the mere act of compiling does not affect the copyright status of GAP.

The compiler is copyright under the terms of the GNU General Public License (GPL). You can find details under


The Cygwin32 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 Cygwin32 glue code). This means that the executable falls under the GPL too, i.e., is distributed freely, which it basically does anyhow.

73.10 Installation of GAP for Windows

Installing GAP under 32-bit Windows should be fairly easy.

In the following, we assume that you install GAP in the directory

C:\GAP4R3

on a Windows98 machine. (For experts (or users without write permission in the C: directory) it is possible to install GAP in another directory, but this will require specialist knowledge (editing a batch file) which many Windows users do not possess. See section 73.11 for details on how to do this.)

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are not to be taken literally.

If you encounter problems please also see section 73.19 of this document.

Get the Windows-specific files gappc4r3.zoo and the Windows version of the zoo archive extractor unzip.exe, and copy them in the C: directory.

How you can get those files is described in the section 73.2. Remember that the distribution consists of binary files and that you must transmit them in binary mode. See

http://www.gap-system.org/Info/mime.html if you have problems with loading the files.

(The zoo archives we provide for GAP contain comments which indicate whether files are text or binary files. The unzip we provide uses these comments to translate the “LF” line endings we use to the Windows style “CRLF”. If you use another zoo extractor you might lose this information and end up with files that might not conform to your operation system standards.)

Open a window with the MS-DOS Command Prompt and go to the C: directory.

C:\WINDOWS> cd c:\

Now unpack the distribution with the command
Chapter 73. Installing GAP

C:\> unzoo -x gappc4r3.zoo
gap4r3\bin\gap.bat -- extracted
gap4r3\doc\aboutgap.tex -- extracted
gap4r3\doc\aggroup.tex -- extracted
[many more lines]

(Afterwards you will not need the file gappc4r3.zoo any longer. If you are short of disk space you can remove it now.)

C:\> del gappc4r3.zoo

Now go into the directory GAP4R3. If there is a bug fixes (there are none in the initial release but invariably some follow later and are deposited in the bugfixes directory on our ftp server(s). Starting with release 4.3, bugfixes will be comprehensive and you need only the most recent bugfix), extract it while in the gap4r3 directory. (The bug fixes extract only on this level of the directory hierarchy to make them applicable even if you chose later to rename the root directory of your GAP distribution.)

The program may ask you for confirmation to overwrite files, you should give this permission.

C:\> cd GAP4R3
C:\GAP4R3> ../unzoo -x ../fix4r3n3.zoo
[extraction information]

If the bugfixes provide a new binary, unpack the corresponding file wbinXXX.zoo in the same way. It overwrites the old binary.

If you got any GAP packages, extract them in the pkg subdirectory in the same way. Some GAP packages are set up to load (or provide the documentation) automatically. To enable this, after installing all the packages you think you will need, you should list all the GAP packages you have installed in a file ALLPKG in the pkg directory (see Section 74.3 in the Reference Manual for details). (Any GAP package that has a C code component is essentially UNIX-dependent and you may as well delete those; such packages typically describe in their README files that they require configure and make to complete their installation or have a src directory.)

The directories trans, small and prim contain data libraries. If you are short of disk space you can erase some of them and any GAP package directories in the pkg directory that you don’t need, but then of course you will not be able to access these data and packages.

We provide a precompiled binary in the bin subdirectory, so there is no need to compile GAP yourself. The provided binary should work under Windows9x and Windows NT.

Now start GAP either by calling

C:\\GAP4R3\ bin\gapw95.exe

or by starting the program gapw95.exe in the directory

C:\GAP4R3\BIN

You should get the GAP banner and then the GAP prompt. (The process of starting GAP may take a little while.)

Try a few things to see if the binary works.
Section 10. Installation of GAP for Windows

```
gap> 2 * 3 + 4;
10
gap> Factorial( 30 );
265252859812191058636308480000000
gap> Size( SymmetricGroup( 10 ) );
3628800
gap> Factors( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 445819223320340849 ]
gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));;
gap> Size( m11 );
7920
gap> Factors( 7920 );
[ 2, 2, 2, 2, 3, 3, 5, 11 ]
gap> Length( ConjugacyClasses( m11 ) );
10
```

Especially try the command line editing and history facilities, because they are probably the most machine dependent feature of GAP.

Note that GAP is developed under UNIX and therefore the key commands are rather UNIX-type then Windows type. We try also to recognize some common Windows key commands such as the arrow keys, but it is likely that not all Windows-special key commands will be recognized; in general GAP will not conform to standard Windows “look-and feel”. (See below on how to enable cut-and-paste.)

Enter a few commands and then make sure that \texttt{ctr-P} redisplay the last command, that \texttt{ctr-E} moves the cursor to the end of the line, that \texttt{ctr-B} moves the cursor back one character, and that \texttt{ctr-D} deletes single characters. So, after entering the above commands, typing

\texttt{ctr-P ctr-P ctr-E ctr-B ctr-B ctr-B ctr-D} 1 return

should give the following lines:

```
gap> Factors( 7921 );
[ 89, 89 ]
```

You also might want to enable cut-and-paste in the \texttt{GAP} window and select a more pleasant icon. You should be able to do this with the PIF file \texttt{gap.pif} in the \texttt{bin} directory (it might show up only as \texttt{GAP} with an MSDOS icon), if necessary following the instructions on the web page

\texttt{http://www.gap-system.org/Info4/windows.html}. (If this PIF file does not work, you will have to create a new PIF file. For details on how to do this, see the Documentation provided by Microsoft.)

The information about the manual is system independent, you can find it in section 73.17.

A few final reminders:

- Make sure that you got and installed the most recent (and probably the corresponding binary).
- We would appreciate after installation your sending us a short note at \texttt{gap@dcs.st-and.ac.uk} (even if you have installed \texttt{GAP 3} before). Generally, we do not reply to such emails; we only use them to gain some idea of how many people use \texttt{GAP} and of the machines/operating systems on which \texttt{GAP} has been successfully installed.
- We also suggest that you subscribe to our \texttt{gap-forum} mailing list; see the \texttt{GAP} web pages for details. Whenever there is a bug fix or new release of \texttt{GAP} this is where it is announced. The \texttt{gap-forum} also deals with user questions of a general nature: bug reports and other problems you have while installing and/or using \texttt{GAP} should be sent to \texttt{gap-trouble@dcs.st-and.ac.uk}.

That’s all, your installation should be complete. We hope that you will enjoy using \texttt{GAP}. Remember, if you have problems, do not hesitate to contact us at \texttt{gap-trouble@dcs.st-and.ac.uk}. See Section 73.19 for what to include in a bug report.
73.11 Expert Windows installation

This section describes how to get a better shell for GAP, how to install GAP in another directory and how to test the installation.

Some users report that the rxvt shell gives a better windows environment for cut/paste etc. You can find information on this at

http://www.gap-system.org/Forum/Pasechni.1/Dima.1/Re._Wind.1/1.html

If you decide to install GAP in another directory you can do so, but you will have to edit a batch file and use this file to start GAP.

First unpack the GAP distribution in the directory you want.

Let's suppose you want GAP to reside in the directory

C:\SOFTWARE\GAP

Extract GAP (as described in the previous section for C:) in this directory. (Alternatively, you can also first unpack it in

C:\GAP4R3 test it there first, and afterwards move it in the desired location.)

You now will have to edit a batch file that will be used to start GAP. This batch file is needed, since GAP otherwise will not find its library directories. The most natural name for this file would be

C:\SOFTWARE\GAP\GAPGAP4R3\BIN\GAP.BAT

This file should contain the following single line (which might be broken over in several lines in this instructions since the page width is limited):

C:\SOFTWARE\GAP\GAPGAP4R3\bin\gapw95.exe -m 14m
-l C:\SOFTWARE\GAP\GAPGAP4R3 %1 %2 %3 %4 %5 %6 %7 %8

If the path includes blanks (such as Program Files), you will have to enclose the -l argument in apostrophes:

C:\Program Files\GAP\GAPGAP4R3\bin\gapw95.exe -m 14m
-l 'C:\Program Files\GAP\GAPGAP4R3' %1 %2 %3 %4 %5 %6 %7 %8

You now should be able to start GAP by clicking this gap.bat file. You might want to copy this file gap.bat into a directory in your search path (for example the DOS directory) as gap.bat to be able to start GAP by its name from any location:

C:\WINDOWS> gap

If you want to check your installation, a set of test files is provided. This test is not a necessary part of the installation, it only serves as a confirmation that everything went OK. (Since running these tests might take several hours and requires quite a bit of free memory, you may wish to skip this step or run only part of the tests. This does no harm.)

As a quick test we start combinat.tst first.

    gap> Filename( DirectoriesLibrary("tst"), "combinat.tst" );
    ".\tst/combinat.tst"
    gap> ReadTest(last);
    + GAP4stones: 26801
    true

Note that GAP internally uses / to separate directory names. This will be translated to backslashes for the operating system.
Now you can use `Read` (not `ReadTest`) to read the file `testall.g` to run all available test files. (You should not pay too much attention to the GAPstone ratings of the different files. The time measurements are not calibrated and sometimes vary substantially because further tests have been added to a file.)

```gap
gap> Filename( DirectoriesLibrary("tst"), "testall.g" );
"./tst/testall.g"
gap> Read(last);
```

You can ignore warnings from `weakptr.tst`, which stem from garbage collections occurring at different times, and those from `grpconst.tst` which stem from differences in available GAP packages.

### 73.12 GAP for MacOS

This section contains information about GAP that is specific to the port of GAP for Apple Macintosh systems under MacOS (simply called GAP for MacOS below).

To run GAP for MacOS you need an Apple Macintosh with a Motorola M68020, M68030, or M68040 processor, or a Power Macintosh. The computer must have at least 16MByte of (physical) memory and a harddisk. For serious calculations, much more may be needed. The operating system must be System 7 or higher. GAP for MacOS runs under System X, however only in Classic Mode.

Note that, alternatively, you can run GAP for UNIX version of GAP in the Terminal window. However, you may need additional development software, available from Apple, to compile GAP yourself. See 73.3 for installation instructions.

The section 73.13 describes the copyright as it applies to the executable version that we distribute. The section 73.14 describes how you install GAP for MacOS.

Please refer to the relevant sections of Chapter 3 in the GAP reference manual (included with the GAP distribution for an overview over the features of GAP for MacOS.

### 73.13 Copyright of GAP for MacOS

In addition to the general copyright for GAP set forth in the Copyright the following terms apply to GAP for the Mac.

The system dependent part of GAP for the Mac was written by Burkhard Höfling (his email address is `b.hoefling@tu-bs.de`). He assigns the copyright to the GAP group. Many thanks to Burkhard for his help! Burkhard Höfling’s port was partly based on an earlier port of GAP for the Mac, which was done by Dave Bayer ( `dab@math.columbia.edu`) and used the Mac Programmers Workshop (MPW) compiler. Many thanks to Dave for his work. Moreover, the built-in editor is based upon the freeware text editor PlainText by Mel Park which, in turn, uses TE32K, a TextEdit replacement by Roy Wood. It also uses Internet Config.

For technical reasons we do not distribute the Macintosh specific source and project files as part of the standard archives. If you are interested in compiling GAP yourself, we are happy to provide you with the appropriate files (contact us at `gap-trouble@dcs.st-and.ac.uk`). The source can be compiled with CodeWarrior Pro 5 with Apple’s Universal Headers 3.3 installed.

Please contact the author `b.hoefling@tu-bs.de` or `gap-trouble@dcs.st-and.ac.uk` if you need further information.
73.14 Installation of GAP for MacOS

Installing GAP under MacOS is fairly easy. First, decide into which folder you want to install GAP 4.3. GAP will be installed in a subfolder \texttt{gap4r3} of this folder. You can later move GAP to a different location.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are not to be taken literally.

If you encounter problems please also see section 73.19 of this document.

Get the Mac-specific files described in 73.2, that is, the distribution \texttt{gapmac4r3.zoo}, the binary archive \texttt{unzoo4r3-PPC.sit}, the most recent bugfix, and the GAP application program (also called binary or executable). The GAP application program is distributed as a binary archive. It is called \texttt{bin4r3-PPC.sit} if no bugfixes requiring a new GAP application have been issued so far. Otherwise, it is called \texttt{bin4r3nX-PPC.sit}, where \texttt{X} is the number of the last bugfix. The latest GAP application archive is available from the GAP distribution web page, or from the \texttt{mac} subdirectory of the \texttt{bin} subdirectory on the ftp servers.

How you can get those files is described in the section 73.2. Remember that the distribution consists of binary files and that you must transmit them in binary mode.

If the \texttt{sit} files did not extract automatically click on them to extract them. If even this fails use one of the standard decompression utilities, such as Stuffit Expander.

After this process you should end up with two applications, GAP 4 PPC and unzoo 4.3 PPC. The latter is used to uncompress the .zoo archives which contain most of GAP. The zoo archives we provide for GAP contain comments which indicate whether files are text or binary files. The \texttt{unzoo} we provide uses these comments. If you use another zoo extractor you might lose this information and end up with files that contain text but cannot be opened as text files.

The following installation example assumes that you are installing GAP in the folder \texttt{Applications} on a PowerPC Macintosh. (For a 68k Macintosh you should replace all references to PPC to ones referring to 68K)

Move the file \texttt{gapmac4r3.zoo} into the folder \texttt{Applications} and drag it onto the icon of \texttt{unzoo 4.3 PPC}. You will get many lines of output in this window.

This should have created a folder \texttt{gap4r3} in the current folder.

(You will not need the file \texttt{gapmac4r3.zoo} any longer. If you are short of disk space you can remove it now.)

If you got not the full distribution file but several small files, extract all of them (except the bug fixes and GAP packages!) in this way.

Move GAP 4 PPC and the bugfix file (if there is one) in the folder \texttt{gap4r3}. Drag the bugfix file onto the icon of unzoo 4.3 PPC to decompress it.

If you got any GAP packages, move them into the \texttt{pkg} folder in the folder folder \texttt{gap4r3} extract them there, in the same way as the bugfix.

Some GAP packages are set up to load (or provide the documentation) automatically. To enable this, after installing all the packages you think you will need, you should list all the GAP packages you have installed in a file \texttt{ALLPKG} in the \texttt{pkg} directory (see Section 74.3 in the Reference Manual for details).

After extraction you may discard all .zoo files if you are short of disk space.

The folders \texttt{trans, small} and \texttt{prim} contain data libraries. If you are short of disk space you can erase some of them and any GAP package directories in the \texttt{pkg} directory that you don't need, but then of course you will not be able to access these data and packages. (Any GAP package that has a C code component is essentially UNIX-dependent and you may as well delete those; such packages typically describe in their README files that they require configure and make to complete their installation or have a src directory.)

Before you use GAP, you should set up GAP's memory allocation, by setting appropriate values by selecting the GAP application and Get Info... in the Finder's File menu (in order to be able to modify the values there, you have to do this before you launch GAP).
The maximum amount of workspace GAP can use depends on the amount of memory the Finder allocates to GAP when it is launched. The maximum amount of GAP workspace is this value, minus a certain amount used internally by the GAP application (for the PPC version, currently around 1.7 Megabytes, plus the size of the GAP application if you do not use virtual memory, and 2.9 Megabytes for the 68K version), minus any additional amount set with the \texttt{-a}, \texttt{-P} or \texttt{-W} command line options (see below).

You can find information about the amount of free GAP workspace, the total amount of available workspace, and the remaining free memory, by choosing About GAP in the Apple menu.

To ensure efficient operation, you should not allocate more memory to GAP than the amount of physical memory in your computer. If you are not using virtual memory, the amount may have to be considerably less (depending on your system and the number of other applications which you may want to run at the same time).

If you notice heavy disk use during garbage collections, this is a clear indication that you have allocated too much memory to GAP.

In order to test your installation now run the GAP application by clicking on GAP 4 PPC. You should get the GAP banner and then the GAP prompt in a window titled GAP log. (The process of starting GAP may take a while.)

Try a few things to see if the installation succeeded.

```gap
gap> 2 * 3 + 4;
10

gap> Factorial( 30 );
265252859812191058636308480000000

gap> Size( SymmetricGroup( 10 ) );
3628800

gap> Factors( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 445819222320340849 ]

gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));;

gap> Size( m11 );
7920

gap> Length( ConjugacyClasses( m11 ) );
10
```

A set of test files is provided, running them all probably takes some 40 minutes on a 200 MHz PPC machine. As a quick test we start combinat.tst first. Initially we must ensure that the print width of GAP is 80 characters per line which we achieve with the \texttt{SizeScreen} command (otherwise we will be swamped with error messages).

```gap
gap> SizeScreen([80,]);

gap> Filename( DirectoriesLibrary("tst"), "combinat.tst" );
"./tst/combinat.tst"

gap> ReadTest(last);
+ GAP4stones: 27801
true
```

Now use \texttt{Read} (not \texttt{ReadTest}) to read the file testall.g to run all available test files. This is not a necessary part of the installation; it only serves as a confirmation that everything went OK. The full test suite takes some time (almost 2 hours on a Pentium III/500) and uses quite a bit of memory (around 100MB), so you may wish to skip this step or run only part of the tests. This does no harm.
You should not pay too much attention to the GAPstone ratings of the different files. The time measurements are not calibrated and sometimes vary substantially because further tests have been added to a file.

```gap
gap> Filename( DirectoriesLibrary("tst"), "testall.g" );
./tst/testall.g
gap> Read(last);
```

<table>
<thead>
<tr>
<th>test file</th>
<th>GAP4stones</th>
<th>time(msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown.tst</td>
<td>15238</td>
<td>21 (next ~ 0 sec)</td>
</tr>
<tr>
<td>listgen.tst</td>
<td>118000</td>
<td>5 (next ~ 0 sec)</td>
</tr>
<tr>
<td>gaussian.tst</td>
<td>2738</td>
<td>325 (next ~ 0 sec)</td>
</tr>
</tbody>
</table>

You can ignore warnings from `weakptr.tst`, which stem from garbage collections occurring at different times, and those from `grpconst.tst` which stem from differences in available GAP packages.

The information about the manual is system independent; you can find it in section 73.17.

A few final reminders:

- Make sure that you got and installed the most recent bugfix.
- We would appreciate after installation your sending us a short note at gap@dcs.st-and.ac.uk (even if you have installed GAP 3 before). Generally, we do not reply to such emails; we only use them to gain some idea of how many people use GAP and of the machines/operating systems on which GAP has been successfully installed.
- We also suggest that you subscribe to our gap-forum mailing list; see the GAP web pages for details. Whenever there is a bug fix or new release of GAP this is where it is announced. The gap-forum also deals with user questions of a general nature; bug reports and other problems you have while installing and/or using GAP should be sent to gap-trouble@dcs.st-and.ac.uk.

That's all, your installation should be complete. Please refer to Chapter 3 in the GAP reference manual for a description of some special features and options of GAP for MacOS.

We hope that you will enjoy using GAP. Remember, if you have problems, do not hesitate to contact us at gap-trouble@dcs.st-and.ac.uk. See Section 73.19 for what to include in a bug report.

### 73.15 Testing for the System Architecture

1. `ARCH_IS_UNIX( )` F tests whether GAP is running on a UNIX system.
2. `ARCH_IS_MAC( )` F tests whether GAP is running on a Macintosh under MacOS
3. `ARCH_IS_WINDOWS( )` F tests whether GAP is running on a Windows system.
Porting GAP to a new operating system should not be very difficult. However, GAP expects some features from the operating system and the compiler and porting GAP to a system or with a compiler that do not have those features may prove very difficult.

The design of GAP makes it quite portable. GAP consists of a small kernel written in the programming language C and a large library written in the programming language provided by the GAP kernel, which is also called GAP.

Once the kernel has been ported, the library poses no additional problem, because all those functions only need the kernel to work, they need no additional support from the environment.

The kernel itself is separated into a large part that is largely operating system and compiler independent, and one file that contains all the operating system and compiler dependent functions. Usually only this file must be modified to port GAP to a new operating system.

Now let us take a look at the minimal support that GAP needs from the operating system and the machine:

You need enough main memory in your computer. The size of the GAP kernel varies between 1.5 and 2.5 MByte (depending on the machine). The GAP library additionally takes a minimum of 10MByte and the library of functions that GAP loads takes up another 1.5 MByte. So it is clear that at least 16 MByte of main memory are required to do any serious work with GAP.

Additionally, the GAP kernel needs a flat address space, that is all the memory is available in one contiguous chunk.

Note that this implies that there is no point in trying to port GAP to plain MS-DOS running on IBM PCs and compatibles. The version of GAP for IBM PC compatibles that we provide runs on machines with the Intel 80486, Pentium or beyond processor under 32-bit Windows. (This is also necessary, because, as just mentioned, GAP wants to view its memory as a large flat address space.)

Next let us turn to the requirements for the C compiler and its library.

As was already mentioned, the GAP kernel is written in the C language. We have tried to use as few features of the C language as possible. GAP has been compiled without problems with compilers that adhere to the old definition from Kernighan and Ritchie, and with compilers that adhere to the new definition from the ANSI-C standard.

Porting GAP to another UNIX should not be hard. You need some very basic understanding of C and UNIX. If you plan to port GAP to a non-UNIX system please contact gap-trouble@dcs.st-and.ac.uk.

The configuration script runs various tests to determine the configuration of your system. It produces a file bin/architecture/config.h which contains definitions according to the test results. It might be, however, that the tests used don’t produce on your machine the results they are expected to or that further tests are necessary. If this is the case the easiest way is to edit the config.h script, remove all object files and call make in the bin/architecture subdirectory. If you have to resort to changing or amending this file, please tell us what had to be changed (mail to gap-trouble@dcs.st-and.ac.uk). If you had to add further definitions please also tell what properties of your system these defines represent.

If GAP compiles but crashes while reading the library or during a garbage collection with a bus error it is possible that the configuration script did not guess the permitted pointer alignment correctly. This value is stored in the line

```
#define C_STACK_ALIGN 2
```

of config.h. Increase the value to the next power of 2 (≤ 8) and compile GAP anew.

There is still a Makefile in the src directory, but it is not used by the configuration process any longer. As a last resort you might want to try this file, but please still report your problems to gap-trouble.
73.17 The Documentation

The GAP manual is distributed in various “books”. The standard distribution contains four of them (as well as a comprehensive index). GAP packages (see Chapter 74 and, in particular, Section 74.3) provide their own documentation in their own doc directories.

All documentation will be available automatically for the online help (see Section 2.7 in the Tutorial and Chapter 2 in this manual for more details).

There also is (if installed) an HTML version of some books that can be viewed with an HTML browser, see 2.3. Some of these use symbol fonts for mathematical formulae. To get them correctly displayed (only in a browser which uses a graphical display), you may want to adjust the font setup for your browser. See 73.18 below for a hint.

The manual is also available in dvi and pdf format. In the full UNIX distribution (gap4r3.zoo) these files are included in the directory gap4r3/doc in the subdirectories tut (a beginner’s tutorial), ref (the reference manual), prg (programmer’s tutorial), ext (programmer’s reference) and new (new material that might still change in future versions).

If you downloaded another distribution file, you can get the missing dvi (resp. pdf) files by downloading the archive docdvi4r3.zoo (resp. docpdf4r3.zoo) from the same place you got GAP.

If you want to use these manual files with the online help you may check (or make sure) that your system provides some additional software like

\texttt{xpdf} \\
\texttt{http://www.foolabs.com/xpdf/}

\texttt{xdvi} \\
\texttt{http://www.ctan.org/}

\texttt{http://www.ctan.org/tex-archive/CTAN.sites} \\
\texttt{and the mirrors are listed at:}

\texttt{http://www.ctan.org/tex-archive/dviware/xdvi/xdvi.tar.gz}

\texttt{acroread} \\
\texttt{see} \\
\texttt{http://www.adobe.com/products/acrobat/readstep.html}

As a complete beginner, we suggest you read the tutorial first for an introduction to GAP 4. Then start to use the system with extensive use of the online help system (see Section 2.7 in the Tutorial and Chapter 2 in this manual).

If you have experience with GAP 3, it might be still worthwhile to at least glance over the first chapters of the tutorial. You however should read the last chapter of the tutorial, “Migrating to GAP4”. This chapter gives a summary of changes between GAP 3 and GAP 4 that will affect the user. It also explains a “compatibility mode” you may turn on to make GAP 4 behave a bit more like GAP 3.

As some of the manuals are quite large, you should not immediately print them. If you start using GAP it will be helpful to print the tutorial (and probably the first chapters of the reference manual). There is no compelling reason to print the whole of the reference manual, better use the online help which provides useful search features.
73.18 HTML Font Setup

The HTML pages of the manual use the symbol font to display non-latin symbols in mathematical formulae. This font might not be enabled by default on your browser. The documentation of the tth converter we use describes how to set up this font for use in Netscape or Konqueror. For Netscape on UNIX, essentially you want:

\[ \text{Netscape*documentFonts.charset*adobe-fontspecific: iso-8859-1} \]

in your .Xresources or .Xdefaults file.

A good test to see if you need this remedy is to check the display of: \( \{1, \ldots, n\} \); if you see an ellipsis (three dots) between the commas, you are ok, and if you see \( 1/4 \) you need the remedy!

If you have problems viewing symbol fonts on the Mac, you may have to choose Mac font encoding, sometimes called MacRoman, or Western (Mac). More information about the font setup on Macintoshes can be found

\[ \text{http://hutchinson.belmont.ma.us/tth/manual/sec12.html} \]

Section 12.3.

73.19 If Things Go Wrong

This section lists a few common problems when installing or running GAP and their remedies. Also see the FAQ list on the GAP web pages at

\[ \text{http://www.gap-system.org/Info/faq.html} \]

GAP starts with a warning “hmm, I cannot find ’lib/init.g’”.

You either started only the binary or did not edit the shell script/batch file to give the correct library path. You must start the binary with the command line option \( -l \) path where \( \text{path} \) is the path to the GAP home directory. See section 3.1 in the reference manual.

When starting, GAP produces error messages about undefined variables.

You might have a .gaprc file that was intended for GAP 3 but is not compatible with GAP 4. See section 3.4 in chapter 3 of the reference manual.

GAP stops with an error message: “cannot extend the workspace any more”. Your calculation exceeded the available memory. Most likely you asked GAP to do something which required more memory than you have (as listing all elements of \( S_{15} \) for example). You can use the command line option \( -g \) (see section 3.1 in the reference manual) to display how much memory GAP uses. If this is below what your machine has available (this happens for example under Windows) extending the workspace is impossible. Start GAP with more memory or use the \( -a \) option to pre-allocate initially a large piece of workspace.

GAP complains: “corrupted completion file”.

Some library files got changed without rebuilding the completion files. This is often a sign that earlier a bugfix was not installed properly or that you changed the library yourself. In the latter case, start GAP with command line option \( -N \) and see section 3.5.

GAP stops with an error message “exceeded the permitted memory”. Your job got bigger than what is permitted by default (128MB). (This is a safety feature to avoid single jobs wrecking a multi-user system.) You can type \texttt{return}; to continue, if the error message happens repeatedly you better start the job anew and use the command line option \( -o \) to set a higher memory limit.

\texttt{make} complains about not being able to find files in \texttt{cnf} or \texttt{src} which exist.

The dates of the new files were not extracted properly (Alpha-OSF machines are prone to this). Call
touch * cnf/* src/*
from the main GAP directory (this ought to reset the date of all relevant files to “now”) and try again.
Recompilation does not actually compile changed files.
The dates of the new files were not extracted properly. Go in the source directory and touch (UNIX command to change date) the new files.
Recompilation fails or the new binary crashes.
Call make clean and restart the configure / make process completely from scratch. (It is possible that the operating system and/or compiler got upgraded in the meantime and so the existing .o files cannot be used any longer.
A calculation runs into an error “no method found”.
GAP is not able to execute a certain operation with the given arguments. Besides the possibility of bugs in the library this means two things: Either GAP truly is incapable of coping with this task (the objects might be too complicated for the existing algorithms or there might be no algorithm that can cope with the input). Another possibility is that GAP does not know that the objects have certain nice properties (like being finite) which are required for the available algorithms. See sections 7.2.1 and 13.7.1.

Problems specific to Windows
The gap.bat file does not start GAP.
Make sure you ran instwin.bat. If your version of Windows uses a language other than English you must still edit the resulting file gap.bat in the bin subdirectory, due to a misdesign of Windows. If the path name contains long file names or blanks, you might have to edit the file to rectify the -l argument or to enclose it by apostrophes.
Windows complains Out of environment space.
Click the batch file instwin.bat or gap.bat which caused the problem with the right mouse button and select Properties, Memory and increase the initial environment space to at least 1024. This will create a pif shortcut which should be used to start GAP.
Command line editing does not work under Windows.
The default key commands are UNIX-like. GAP also tries to emulate some of the special keys under Windows, however if the key repeat is set too high, Windows loses parts of the codes for these keys and thus GAP cannot recognize them. Windows98 produces the same scan code for all cursor keys. As GAP does not interface directly with the Windows machinery, there is no way around this problem so far.
The -key cannot be entered.
This is a problem if you are running a keyboard driver for some non-english languages. These drivers catch the ^ character to produce the French circumflex accent and do not pass it properly to GAP.
No fix is known.
Cut and Paste does not work
See

If all these remedies fail or you encountered a bug please send a mail to gap-trouble@dcs.st-and.ac.uk.
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 machine, operating system, (compiler used, if UNIX/Linux) and version of GAP (for example “gap4r3, fix1”) you are using (the line after the GAP 4 banner starting:
GAP4, Version: 4...
when your GAP 4 starts up, supplies the information required).
A GAP package is the way to make software written by (groups of) GAP users available to the computational community together with GAP.

A GAP package extends the functionality of GAP as defined by the GAP kernel, the GAP library and the various data libraries. Some GAP packages are written entirely in the GAP language, while others include one or more standalone programs written in C or some other language which either perform tasks that are not available in GAP or work particularly fast.

The responsibility and copyright of a GAP package remains with the original author while the responsibility of the rest of GAP lies with the GAP developer team.

A GAP package undergoes a formal refereeing process before it becomes part of the GAP distribution. This process is in many ways similar to the refereeing process of a paper submitted to a journal. It assesses the quality and usefulness of the submitted package, it makes sure that it can be started and runs smoothly together with GAP and it ensures that binary parts are portable to other architectures. GAP packages should be submitted to the chairman of the GAP Council, Prof. Charles Wright. See the GAP Web site (see 73.2) for more details and addresses.

While much effort has been spent on making GAP available on as many platforms as possible, the same is not always true for GAP packages. GAP packages that consist entirely of GAP code can be used on any platform where GAP runs. However, GAP packages that contain standalone programs often run only in a UNIX environment.

### 74.1 Installing GAP Packages

Typically, a GAP package is installed by copying its directory (or unpacking its distribution file using unzoo) into the pkg directory of the GAP distribution. The files acceptedpkg4r3.zoo and deppkg4r3.zoo each contain several GAP packages, but each package can also be obtained individually from each of the GAP web sites; just follow the links from the GAP Packages link. The easiest procedure is to change directory to the pkg directory; then download or move the package.zoo file to that directory and do:

```
unzoo package.zoo
```

This typically creates a subdirectory package (or several subdirectories in the case of the omnibus archives: acceptedpkg4r3.zoo and deppkg4r3.zoo) in the directory pkg. Such subdirectories of pkg are called the **home directories** for these GAP packages. It is possible to have several GAP root directories (see 9.2); therefore it is easy to install a package locally even if a user has no permission to add files to the main GAP installation. Thus, GAP packages also provide an easy way to add optional components to the functionality of GAP.

If the GAP package uses external binaries, additional compilation is necessary. The documentation of each GAP package will describe how to do this, the following paragraph describes a suggested procedure for compiling binaries on a UNIX system:

Go to the directory of the GAP package, i.e. for package **package** do:
cd package

Then call:

./configure path

where path is the path to the root directory of the GAP installation. So for example if the GAP package is in the main GAP path, simply call

./configure ../..

This will fetch the name of the system architecture on which GAP has been compiled. Finally call:

make

to invoke the actual compilation. You should check the package’s README in case there are any departures from this procedure.

The standard GAP distribution contains a GAP package example which makes use of a default path mechanism. The configure script in this package permits one to omit the path, using ../../ as a default.

Note: If GAP is installed on different architectures on a common file system, this configuration process will only work for the last architecture for which GAP was compiled. Therefore compile the GAP package binaries always immediately after compiling the system architecture. If you want to add GAP packages later, you should call configure again in the main GAP directory for each architecture before configuring the GAP package for this architecture.

### 74.2 Installing a GAP Package in your home directory

If you are using GAP on a multi-user system installing a GAP package globally in the way described in the last section might require the cooperation of your system administrator.

If your administrator is absent/unwilling/in capable you can still install the GAP package in your local home directory: Create a directory that will serve as “your” personal GAP home directory. We will assume now that you are using a UNIX system and that this directory is called

```
/home/user/mygap
```

Go to this directory and create a subdirectory pkg

```
you@unix> cd /home/user/mygap
```

```
you@unix> mkdir pkg
```

Now extract the GAP package in the pkg subdirectory, creating a directory with the package name. Configure the package as usual. However you now have to give the correct path to the main GAP home directory (start up GAP and look at the variable GAP_ROOT_PATHS to find out where this is, e.g.

```
gap> GAP_ROOT_PATHS;
[ "/usr/local/lib/gap4r3/" ]
```

indicates that the main GAP directory is /usr/local/lib/gap4r3/), so your commands would look something like:

```
you@unix> cd /home/user/mygap/pkg/thepackage
```

```
you@unix> ./configure /usr/local/lib/gap4r3
```

```
you@unix> make
```

Now the GAP package is installed locally. Finally, to use the package you only need to tell GAP to look in the right place. You can achieve this by giving two paths to home directories, the old one and your new
Section 3. Loading a GAP Package

821

one, separated by a semicolon as arguments to the -l command line option to your usual gap4 command. (See 3.1 for details.)

you@unix> gap4 -l "/usr/local/lib/gap4r3;/home/user/mygap"

Now GAP starts as usual, and the new package ought to be available.

Of course, it may be a little tedious to type this each time, and you may be tempted to create an alias and put this in your .cshrc (or .bashrc or whatever) file. In general, this is not the best solution. It is far better to create a shell script (let’s call it mygap), put this script into your private bin directory and add that bin directory to your path. The easy way to create mygap is to copy the one that runs your gap4 (it better be a script! ... if not tell your administrator to read Section 73.4), e.g.

you@unix> mkdir /home/user/bin
you@unix> cd /home/user/bin
you@unix> which gap4
/usr/local/bin/gap4
you@unix> cp /usr/local/bin/gap4 mygap

(If you already have a bin directory you can, of course, skip the first step).

Now edit the -l part of the last line of mygap which should initially look something like:

exec $GAP_DIR/bin/$GAP_PRG -m $GAP_MEM -o 970m -l $GAP_DIR $*

so that it becomes:

exec $GAP_DIR/bin/$GAP_PRG -m $GAP_MEM -o 970m -l "$GAP_DIR;/home/user/mygap" $*

Then to start gap4 with your locally installed package you only need to type mygap.

For the GAP package ANUPQ the above strategy is actually necessary if one wants to run the binary as a stand-alone; when computing stabilisers it needs to call a GAP that knows where the packages ANUPQ and AutPGrp are installed. So, take the hint, always use an appropriate script when running GAP; don’t use aliases. (Most of the time aliases work ok; but when that approach doesn’t work you will be left puzzling!!)

74.3 Loading a GAP Package

Some GAP packages will be 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 RequirePackage.

1 ▶ RequirePackage( name, [version] )

loads the GAP package 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 (see 4.11 in “Extending GAP”). The argument name is case insensitive.

RequirePackage 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 available version is too small. If the package name has already been loaded in a version number at least version, RequirePackage returns true without doing anything.

After a package has been loaded its code and documentation should be available as other parts of the GAP library are. Each package has a doc directory containing the package’s documentation; generally this contains .tex files, and manual.dvi and manual.six files. Some packages have an htm directory containing an HTML version of the manual. If the documentation was produced using GAPDoc there will be .txt files (plain text files that are read by GAP when one types ?topic commands) and .html files (which contains the HTML version of the manual) as well as manual.dvi (the \TeX .dvi version of the manual), all in the package’s doc directory. In any case, the manual.six file must be present to enable GAP’s on-line help to
display a GAP package’s documentation by reading from either the package’s .txt or .tex files; if it is not check the package’s README for instructions on how to obtain it.

The documentation of each GAP package will tell you if the package loads automatically or not. Also, GAP prints the list of names of all GAP packages which have been loaded (either by automatic loading or via RequirePackage commands in one’s .gaprc file or the like) at the end of the initialization process.

A GAP package may also install only its documentation automatically but still need loading by RequirePackage. In this situation the online help displays (not loaded) in the header lines of the manual pages belonging to this GAP package.

(At the moment automatic loading is only possible for the packages listed in the file pkg/ALLPKG. (This is due to the fact that there is no standard C-Function that will list the contents of a subdirectory.) This file must list each package name on a line of its own without any heading or trailing extra characters. Under UNIX you can create such a file easily by issuing the command

```
find * -type d -maxdepth 0 -print > ALLPKG
```

in the pkg directory.)

If the GAP installation you are using loads GAP packages automatically which (for example for reasons of memory usage) you do not want to load automatically, you can disable the automatic loading of GAP packages by putting a line

```
RemoveSet(AUTOLOAD_PACKAGES, "name");
```

where name is the name of the GAP package, in your .gaprc file (see 3.4).

You can disable automatic loading globally, by listing the name of the GAP package in a file NOAUTO in the pkg directory.

### 74.4 Functions for GAP Packages

1. DeclarePackage( name, version, tester )
2. DeclareAutoPackage( name, version, tester )

This function may only occur within the init.g file of the GAP package name. It prepares the installation of the package name, which will be installed in version version. The third argument tester is a function which tests for necessary conditions for the package to be loadable, like the availability of other packages (using TestPackageAvailability, see 74.4.4) or – if necessary – the existence of compiled binaries. It should return true only if all conditions are fulfilled (and false or fail otherwise). If it does not return true, the package will not be loaded, and the documentation will not be available. The second version DeclareAutoPackage declares the package and enables automatic loading when GAP is started. (Because potentially all installed packages are automatically loaded, the tester function should take little time.)

The argument name is case sensitive. But note that the package name is internally converted to a lower case string; so, there cannot be different packages with names which are only distinguished by case.

The difference between the Auto and the other version is that in case of IS_IN_AUTOLOAD the component PACKAGES_VERSIONS.(lname) is bound only with the Auto version. This leads to different behaviour in RequirePackage when it asks for package availability.

1. DeclarePackageDocumentation( name, doc[, short[, long ] ] )
2. DeclarePackageAutoDocumentation( name, doc[, short[, long ] ] )

This function indicates that the documentation of the GAP package name can be found in its doc subdirectory. The second version will enable that the documentation is loaded automatically when GAP starts, even if the package itself will not be loaded.

Both functions may only occur within the init.g file of a GAP package.
The string \texttt{name} is case insensitive (but internally converted to lower case, e.g., for the directory name of the package).

There are two optional arguments: \texttt{short} is a short string which is used as identifying name of the book in GAP's online help. And \texttt{long} is a short description of the book. This is shown with the \texttt{?books} query, so \texttt{short} and \texttt{long} together should easily fit on a line.

If \texttt{short} and \texttt{long} are not given, the default values \texttt{name} for \texttt{short} and \textbf{GAP Package} \texttt{‘name’} for \texttt{long} are used.

\begin{verbatim}
3\textgreater \texttt{ReadPkg( name, file )}
\end{verbatim}

reads the file \texttt{file} of the GAP package \texttt{name}, where \texttt{file} is given as a relative path to the directory of \texttt{name}. The \texttt{name} argument is case insensitive.

\begin{verbatim}
4\textgreater \texttt{TestPackageAvailability( name, version )}
\end{verbatim}

tests, whether the GAP package \texttt{name} is available for loading in a version that is at least \texttt{version}. It returns \texttt{true} if the package is already loaded, \texttt{fail} if it is not available, and the directory path to the package if it is available, but not yet loaded. A test function (the third parameter to \texttt{DeclarePackage}) should therefore test for the result of \texttt{TestPackageAvailability} being not equal to \texttt{fail}. The argument \texttt{name} is case insensitive.

\begin{verbatim}
5\textgreater \texttt{InstalledPackageVersion( name )}
\end{verbatim}

returns the version number string of the package \texttt{name} if it is installed and \texttt{fail} otherwise. The \texttt{name} argument is case insensitive.

\begin{verbatim}
6\textgreater \texttt{DirectoriesPackageLibrary( name [, path] )}
\end{verbatim}

takes the string \texttt{name}, a name of a GAP package and returns a list of directory objects for the sub-directory/ies containing the library functions of the GAP package, up to one for each \texttt{pkg} sub-directory of a path in \texttt{GAP\_ROOT\_PATHS}. The default is that the library functions are in the subdirectory \texttt{lib} of the GAP package’s home directory. If this is not the case, then the second argument \texttt{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 \texttt{name}.

As an example, the following returns a directory object for the library functions of the GAP package \texttt{example}:

\begin{verbatim}
gap> DirectoriesPackageLibrary( "example", "gap" );
[ dir("/home/werner/Gap/4.0/pkg/example/gap") ]
\end{verbatim}

Observe that we needed the second argument \texttt{"gap"} here, since \texttt{example}'s library functions are in the sub-directory \texttt{gap} rather than \texttt{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 \texttt{lib}. The directories in \texttt{path} should be separated by \texttt{/} (even on systems, like Windows, which use \texttt{\} as the directory separator). For example, suppose there is a package \texttt{somepackage} with a subdirectory \texttt{m11} in the directory \texttt{data}, then we might expect the following:

\begin{verbatim}
gap> DirectoriesPackageLibrary( "somepackage", "data/m11" )
[ dir("/home/werner/Gap/4.0/pkg/somepackage/data/m11") ]
\end{verbatim}

\begin{verbatim}
7\textgreater \texttt{DirectoriesPackagePrograms( name )}
\end{verbatim}

returns a list of the \texttt{bin/architecture} subdirectories of all packages \texttt{name} where \texttt{architecture} is the architecture on which GAP has been compiled. The directories returned by \texttt{DirectoriesPackagePrograms} is the place where external binaries for the GAP package \texttt{name} and the current architecture should be located.
gap> DirectoriesPackagePrograms( "nq" );
[ dir("/home/werner/Gap/4.0/pkg/nq/bin/i686-unknown-linux2.0.30-gcc") ]

CompareVersionNumbers( supplied, required )

compares two version numbers, given as strings. They are split at non-number 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. See Section 4.11 of “Extending GAP” for details about version numbers.
Bibliography


Bibliography


[Ple90] Wilhelm Plesken. Additive decompositions of positive integral quadratic forms. The paper is available at Lehrstuhl B für Mathematik, Rheinisch Westfälische Technische Hochschule Aachen, may be it will be published in the near future, 1990.


[PP80] Wilhelm Plesken and Michael 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.


This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., “PermutationCharacter” comes before “permutation group”.

(Near-)Additive Magma Categories, 539
(Near-)Additive Magma Generation, 540
+, 48
-, 48
-A, 30
-B, 31
-C, 31
-D, 31
-K, 29
-L, 30
-M, 31
-N, 31
-P, 31
  on Macintosh, 32
-R, 30
-T, 31
-U, 31
-W, 31
  on Macintosh, 32
-X, 31
-Y, 31
-a, 30
  on Macintosh, 32
-b, 28
-e, 28
  on Macintosh, 32
-f, 28
  on Macintosh, 32
-g, 29
-h, 28
-i, 31
-l, 29
-m, 29
-n, 28
  on Macintosh, 32
-o, 29

on Macintosh, 32
-q, 28
-r, 30
-x, 28
-y, 29
-z, 31
  on Macintosh, 32
.gaprc, 34
/, 48
  for character tables, 698
*, 48
  for character tables, 698
\', 242
\\, 242
\n, 242
\r, 242
\n, 242
\in, operation for testing membership, 264
\in, 242
\ns, 242
\b, 242
\c, 242
\n, 242
\r, 242
\", 48
  for class functions, 740
1-Cohomology, 363
2-Cohomology and Extensions, 441
4.2BSD, 800
4.3BSD, 800

A
AbelianGroup, 493
AbelianInvariants, for character tables, 701
  for groups, 352
Abelian Invariants for Subgroups, 461
AbelianInvariantsMultiplier, 366
AbelianInvariantsNormalClosureFpGroup, 461
AbelianInvariantsNormalClosureFpGroupRrs, 461
AbelianInvariantsSubgroupFpGroup, 461
AbelianInvariantsSubgroupFpGroupMtc, 461
AbelianInvariantsSubgroupFpGroupRrs, 461
AbelianNumberField, 570
abelian number field, 570
abelian number fields, Galois group, 571
AbelianSubfactorAction, 388
About Group Actions, 380
AbsInt, 126
AbsoluteValue, 155
absolute value of an integer, 126
abstract word, 314
AbstractWordTietzeWord, 470
accessing, list elements, 170
record elements, 251
Accessing a Module, 665
Accessing Record Elements, 251
Accessing Subgroups via Tables of Marks, 685
AClosestVectorCombinationsMatFFEVecFFE, 212
ActingAlgebra, 608
ActingDomain, 392
Acting OnRight and OnLeft, 418
Action, 386
action, by conjugation, 381
on blocks, 381
on sets, 381
ActionHomomorphism, 386
Action of a group on itself, 387
Action on Subfactors Defined by a Pcgs, 432
actions, 381
Actions of Matrix Groups, 415
ActorOfExternalSet, 394
Add, 173
add, an element to a set, 191
AddCoeffs, 211
AddGenerator, 472
AddGenerators, 333
AddGeneratorsExtendSchreierTree, 411
addition, 48
list and non-list, 181
matrices, 216
matrix and scalar, 216
operation, 285
rational functions, 644
scalar and matrix, 216
scalar and matrix list, 217
scalar and vector, 209
vector and scalar, 209
vectors, 208
AdditiveArithmetic for Lists, 180
AdditiveInverse, 283
AdditiveInverseAttr, 283
AdditiveInverseImmutable, 283
AdditiveInverseMutable, 283
AdditiveInverseOp, 283
AdditiveInverseSameMutability, 283
AdditiveInverseSM, 283
AdditiveNeutralElement, 541
AddRelator, 472
AddRowVector, 211
AddRule, 333
AddRuleReduced, 333
AddSet, 191
AdjointAssociativeAlgebra, 625
AdjointBasis, 598
AdjointMatrix, 625
AdjointModule, 610
administrator, 820
Advanced Features of GAP, 30
Advanced List Manipulations, 199
Advanced Methods for Dixon-Schneider Calculations, 718
AffineAction, 433
AffineActionLayer, 433
AffineOperation, 433
AffineOperationLayer, 433
Agemo, 349
Algebra, 589
AlgebraByStructureConstants, 592
AlgebraGeneralMappingByImages, 602
AlgebraHomomorphismByImages, 603
AlgebraHomomorphismByImagesNC, 603
AlgebraicExtension, 660
AlgebraWithOne, 589
AlgebraWithOneGeneralMappingByImages, 603
AlgebraWithOneHomomorphismByImages, 604
AlgebraWithOneHomomorphismByImagesNC, 604
AllBlocks, 391
AllIrreducibleSolvableGroups, 510
AllLibraryGroups, 499
ALLPKG, 822
AllPrimitiveGroups, 499
AllSmallGroups, 502
AllTransitiveGroups, 499
Alpha, 790
AlternatingGroup, 493
and, 166
for filters, 116, 167
Index

An Example of Advanced Dixon-Schneider Calculations, 719
ANFAutomorphism, 571
AntiSymmetricParts, 757
antisymmetric relation, 304
Append, 173
AppendTo, 94
for streams, 100
Apple, 811
ApplicableMethod, 78
ApplicableMethod, 78
ApplicableMethodTypes, 78
Apply, 194
ApplySimpleReflection, 622
ApproximateSuborbitsStabilizerPermGroup, 409
ARCH_IS_MAC, 814
ARCH_IS_UNIX, 814
ARCH_IS_WINDOWS, 814
arg, special function argument, 46, 56
Arithmetic for External Representations of Polynomials, 658
Arithmetic for Lists, 178
Arithmetic Operations for Class Functions, 738
Arithmetic Operations for Elements, 284
Arithmetic Operations for General Mappings, 297
Arithmetic Operators, 48
Arrangements, 145
arrow notation for functions, 57
AsAlgebra, 598
AsAlgebraWithOne, 599
AsBinaryRelationOnPoints, 305
AsBlockMatrix, 230
AscendingChain, 355
AsDivisionRing, 560
AsDuplicateFreeList, 193
AsField, 560
AsFreeLeftModule, 557
AsGroup, 336
AsGroupGeneralMappingByImages, 369
AsLeftIdeal, 547
AsLeftModule, 555
AsList, 259
AsMagma, 309
AsMonoid, 527
AsPolynomial, 646
AsRightIdeal, 547
AsRing, 544
AsSemigroup, 520

Assert, 81
AssertionLevel, 81
Assertions, 81
AsSet, 260
AssignGeneratorVariables, 321
assignment, to a list, 171
to a record, 252
variable, 49
Assignments, 49
AssignNiceMonomorphismAutomorphismGroup, 376
AssociatedPartition, 150
AssociatedReesMatrixSemigroupOfDClass, 526
Associates, 550
associativity, 48
AssocWordByLetterRep, 326
AsSortedList, 260
AsSSortedList, 260
AsStruct, 277
AsSubalgebra, 599
AsSubalgebraWithOne, 599
AsSubgroup, 337
AsSubgroupOfWholeGroupByQuotient, 459
AsSubmagma, 309
AsSubmonoid, 527
AsSubsemigroup, 520
AsSubspace, 574
AsSubstruct, 280
AsTransformation, 537
AsTransformationNC, 537
AsTwoSidedIdeal, 547
AsVectorSpace, 573
at exit functions, 72
ATLAS Irrationalities, 157
AtlasIrrationality, 159
atomic irrationalities, 157
Attributes, 120
Attributes and Operations for Algebras, 597
Attributes and Properties for (Near-)Additive Magmas, 541
Attributes and Properties for Collections, 261
Attributes and Properties for Magmas, 311
Attributes and Properties for Matrix Groups, 414
Attributes and Properties of Character Tables, 699
Attributes and Properties of Elements, 281
Attributes of and Operations on Equivalence Relations, 306
Attributes of Tables of Marks, 675
AttributeValueNotSet, 121
AugmentationIdeal, 637
AugmentedCosetTableInWholeGroup, 454
AugmentedCosetTableMtc, 454
AugmentedCosetTableRrs, 454
Augmented Coset Tables and Rewriting, 454
AUTLOAD_PACKAGES, 822
automatic loading of gap packages, 821
AutomorphismDomain, 375
AutomorphismGroup, 375
for groups with pcgs, 434
automorphism group, of number fields, 571
Automorphisms and Equivalence of Character Tables, 726
AutomorphismsOfTable, 702

B
by, 157
backslash character, 242
backspace character, 242
Backtrace, GAP3 name for Where, 70
Backtrack, 411
BaseFixedSpace, 223
BaseMat, 225
BaseMatDestructive, 225
BaseOfGroup, 408
BaseOrthogonalSpaceMat, 225
Bases of Vector Spaces, 575
BaseStabChain, 408
BaseSteinitzVectors, 226
Basic Actions, 381
Basic Groups, 492
Basic Operations for Class Functions, 737
Basic Operations for Lists, 169
BasicWreathProductOrdering, 273
Basis, 576
BasisNC, 576
BasisVectors, 577
Bell, 143
Bernoulli, 144
BestQuoInt, 128
BestSplittingMatrix, 718
BiAlgebraModule, 607
BiAlgebraModuleByGenerators, 606
BilinearFormMat, 620
binary relation, 303
BinaryRelationByElements, 303
BinaryRelationOnPoints, 304
BinaryRelationOnPointsNC, 304
Binary Relations on Points, 304
BinaryRelationTransformation, 538
BindGlobal, 45
Binomial, 143
blank, 40
BlistList, 204
Block Matrices, 230
BlockMatrix, 230
Blocks, 390
BlocksInfo, 707
Block Systems, 390
BlowUpMat, 227
BlowUpVector, 227
BombieriNorm, 651
Boolean Lists Representing Subsets, 204
bound, 42
Brauer character, 744
BrauerCharacterValue, 767
BrauerTable, 693
BrauerTableOp, 693
BravaisGroup, 418
BravaisSubgroups, 418
BravaisSupergroups, 418
Break, 54
break loop message, 69
Break Loops, 67
break statement, 54
browsing backwards, 23
browsing backwards one chapter, 23
browsing forward, 23
browsing forward one chapter, 23
browsing the next section browsed, 23
browsing the previous section browsed, 23
Browsing through the Sections, 23
BSD, 800
bug reports, see If Things Go Wrong, 818
Building new orderings, 269

C
cy, 157
Calculating with Group Automorphisms, 376
Calendar Arithmetic, 248
CallFuncList, 61
Calling a function with a list argument that is interpreted as several arguments, 61
Index

Cancellation Tests for Rational Functions, 658
CanComputeIndex, 367
CanComputeIsSubset, 367
CanComputeSize, 366
CanComputeSizeAnySubgroup, 367 candidates, for permutation characters, 760, 762
CanEasilyCompareElements, 284
CanEasilyCompareElementsFamily, 284
CanEasilyComputePcgs, 421
CanEasilySortElements, 284
CanEasilySortElementsFamily, 284
CanEasilyTestMembership, 366
CanonicalBasis, 576 canonical basis, for matrix spaces, 583 for row spaces, 583
CanonicalGenerators, 620
CanonicalPcElement, 423
CanonicalPcgs, 426
CanonicalPcgsByGeneratorsWithImages, 428
CanonicalRepresentativeDeterminatorOfExternalSet, 393
CanonicalRepresentativeOfExternalSet, 393
CartanMatrix, 620
CartanSubalgebra, 617
Cartesian, 195
CAS, 728
CAS format, 728
CASString, 728
CAS tables, 728
Categories, 117
Categories and Properties of Algebras, 596
Categories for Streams and the StreamsFamily, 96
Categories of Associative Words, 319
Categories of Matrices, 215
CategoriesOfObject, 119
Categories of Words and Nonassociative Words, 314
CategoryCollections, 257
Center, 312
center, 311
CentralCharacter, 747 central character, 747
CentralIdempotentsOfAlgebra, 601 centraliser, 311
Centralizer, 311 for groups with pegs, 434
CentralizerInGLnZ, 418
CentralizerModulo, 356
CentralizerSizeLimitConsiderFunction, 435 Centre, 312 for groups with pegs, 434 centre, of a character, 746 CentreOfCharacter, 746 ChangeStabChain, 410 Changing Presentations, 471 Changing the Help Viewer, 24 Changing the Representation, 278 Changing the Structure, 277 Character, 742 CharacterDegrees, 699 Character Degrees and Derived Length, 790 Characteristic, 281 characteristic, for class functions, 741 CharacteristicPolynomial, 226 characteristic polynomial, for field elements, 562 CharacterNames, 703 characters, 735 permutation, 760, 762 symmetrizations of, 756 CharacterTable, 693 Character Table Categories, 694 CharacterTableDirectProduct, 721 CharacterTableFactorGroup, 722 CharacterTableIsoclinic, 723 character tables, 692 access to, 692 calculate, 692 infix operators, 698 of groups, 692 CharacterTableWithSortedCharacters, 724 CharacterTableWithSortedClasses, 724 CharacterTableWreathSymmetric, 723 character value, of group element using powering operator, 740 CharsFamily, 243 CheckFixedPoints, 781 CheckForHandlingByNiceBasis, 588 CheckPermChar, 788 ChevalleyBasis, 619 ChiefSeries, 352 ChiefSeriesThrough, 352 ChiefSeriesUnderAction, 353 ChineseRem, 129 Chinese remainder, 129
Chomp, 247
CIUnivPols, 643
ClassElementLattice, 358
classes, real, 704
ClassesSolvableGroup, 434
ClassFunction, 742
class function, 735
class function objects, 735
class functions, 777
    as ring elements, 739
ClassFunctionSameType, 743
Class Fusions between Character Tables, 773
Classical Groups, 494
ClassMultiplicationCoefficient, for character tables, 711
class multiplication coefficient, 711, 712
ClassNames, 703
ClassNamesTom, 676
ClassOrbit, 704
ClassPositionsOfAgemo, 705
ClassPositionsOfCentre, for characters, 746
    for character tables, 705
ClassPositionsOfDerivedSubgroup, 705
ClassPositionsOfDirectProduct-Decompositions, 705
ClassPositionsOfElementaryAbelianSeries, 705
ClassPositionsOfFittingSubgroup, 705
ClassPositionsOfNormalClosure, 705
ClassStructureCharTable, 711
ClassTypesTom, 676
CleanedTailPcElement, 423
ClearCacheStats, 84
ClearProfile, 82
clone, an object, 113
CloseMutableBasis, 581
CloseStream, 97
ClosureGroup, 338
ClosureGroupAddElm, 338
ClosureGroupCompare, 338
ClosureGroupDefault, 338
ClosureGroupIntest, 338
ClosureLeftModule, 556
ClosureNearAdditiveGroup, 541
Closure Operations and Other Constructors, 305
ClosureRing, 544
Closures of (Sub)groups, 338
ClosureStruct, 277
ClosureSubgroup, 338
ClosureSubgroupNC, 338
Coboundaries, 630
Cochain, 629
CochainSpace, 629
Cocycles, 630
cocycles, 363
CodePcGroup, 444
CodePcgs, 444
Coding a Pc Presentation, 444
coefficient, binomial, 143
Coefficient List Arithmetic, 211
Coefficients, 577
coefficients, for cyclotomics, 155
CoefficientsAndMagmaElements, 638
CoefficientsFamily, 655
CoefficientsMultiadic, 129
CoefficientsOfLaurentPolynomial, 652
CoefficientsOfUnivariatePolynomial, 646
CoefficientsOfUnivariateRationalFunction, 646
CoefficientsQadic, 129
CoefficientsRing, 653
CoeffsCyc, 155
CoeffsMod, 211
cohomology, 363
COHORTS_PRIMITIVE_GROUPS, 510
CoKernelOfAdditiveGeneralMapping, 300
CoKernelOfMultiplicativeGeneralMapping, 299
CollapsedMat, 784
Collected, 193
Collection Families, 257
CollectionsFamily, 257
ColumnIndexOfReesMatrixSemigroupElement, 526
ColumnIndexOfReesZeroMatrixSemigroupElement, 526
Combinations, 145
Combinatoric Arrangements and Tuples, 145
CombinatorialCollector, 439
Combinatorial Numbers, 143
Index

Comm, 285
   for words, 322
Command Line Options, 28
   comments, 40
CommutativeDiagram, 780
CommutatorFactorGroup, 356
CommutatorLength, 346
   for character tables, 701
CommutatorSubgroup, 345
Compacted, 193
CompanionMat, 228
CompareVersionNumbers, 823
   comparison, fp semigroup elements, 532
   operation, 284
   rational functions, 644
Comparison of Associative Words, 322
Comparison of Class Functions, 738
Comparison of Elements of Finitely Presented Groups, 447
Comparison of Elements of Finitely Presented Semigroups, 532
Comparison of Permutations, 396
Comparison of Rational Functions, 644
Comparison of Words, 315
Comparison Operations for Elements, 284
Comparisons, 47
   comparisons, of booleans, 165
   of lists, 177
Comparisons of Booleans, 165
Comparisons of Cyclotomics, 157
Comparisons of Lists, 177
Comparisons of Records, 254
Comparisons of Strings, 244
CompatibleConjugacyClasses, 697
CompatiblePairs, 443
Compiling Library Code, 36
Complementclasses, 345
ComplementclassesEA, 365
ComplementSystem, 348
Completion Files, 34
ComplexConjugate, 571
   for class functions, 741
ComplexificationQuat, 593
Components of a Dixon Record, 719
CompositionMapping, 293
   for Frobenius automorphisms, 568
CompositionMapping2, 293
CompositionMaps, 778
CompositionOfStraightLinePrograms, 330
CompositionSeries, 353
   for groups with gcgs, 434
ComputedBrauerTables, 693
ComputedClassFusions, 774
ComputedIndicators, 710
ComputedIsPSolvableCharacterTables, 710
ComputedPowerMaps, 769
ComputedPrimeBlocks, 706
Computing a GCgs, 420
Computing a Permutation Representation, 400
Computing Pc Groups, 440
Computing Possible Permutation Characters, 762
Computing the Irreducible Characters of a Group, 715
Concatenation, 193
   concatenation, of lists, 193
Conductor, 155
ConfluentRws, 333
Congruences, for character tables, 786
Congruences for semigroups, 522
ConjugacyClass, 343
Conjugacy Classes, 343
ConjugacyClasses, attribute, 343
   for character tables, 696
   for groups with gcgs, 434
   for linear groups, 497
ConjugacyClassesByOrbits, 344
ConjugacyClassesByRandomSearch, 343
Conjugacy Classes in Classical Groups, 497
Conjugacy Classes in Solvable Groups, 434
ConjugacyClassesMaximalSubgroups, 357
ConjugacyClassesPerfectSubgroups, 360
ConjugacyClassesSubgroups, 357
ConjugacyClassesSubgroups, 356
conjugate, matrix, 217
   of a word, 322
ConjugateDominantWeight, 622
ConjugateDominantWeightWithWord, 622
ConjugateGroup, 336
Conjugates, 563
ConjugateSubgroup, 338
ConjugateSubgroups, 338
conjugation, 381
ConjugatorAutomorphism, 373
ConjugatorAutomorphismNC, 373
ConjugatorIsomorphism, 373
ConjugatorOfConjugatorIsomorphism, 374
ConnectGroupAndCharacterTable, 697
ConsiderKernels, 786
ConsiderSmallerPowerMaps, 786
ConsiderTableAutomorphisms, 788
ConstantTimeAccessList, 189
constituent, of a group character, 745
ConstituentsCompositionMapping, 293
ConstituentsOfCharacter, 746
Constructing Algebras as Free Algebras, 590
Constructing Algebras by Generators, 589
Constructing Algebras by Structure Constants, 591
Constructing Character Tables from Others, 721
Constructing Domains, 276
Constructing Lie algebras, 614
Constructing Pc Groups, 438
Constructing Subdomains, 280
Constructing Tables of Marks, 671
Constructing Vector Spaces, 573
Construction of Stabilizer Chains, 405
Constructors for Basic Groups, 498
ContainedCharacters, 784
ContainedDecomposables, 784
ContainedMaps, 779
ContainedPossibleCharacters, 783
ContainedPossibleVirtualCharacters, 783
ContainedSpecialVectors, 783
ContainedTom, 680
ContainingTom, 680
continue statement, 55
Conventions for Character Tables, 696
convert, to a string, 243
Converting Groups to Finitely Presented Groups, 455
ConvertToCharacterTable, 694
ConvertToCharacterTableNC, 694
ConvertToMatrixRep, 229
ConvertToRangeRep, 202
ConvertToStringRep, 243
ConvertToTableOfMarks, 675
ConvertToVectorRep, 210
ConwayPolynomial, 569
Conway Polynomials, 569
coprime, 48
copy, 113
an object, 113
CopyOptionsDefaults, 410
Copyright of GAP for MacOS, 811
Copyright of GAP for Windows, 806
CopyStabChain, 410
Core, 345
CorrespondingGeneratorsByModuloPcgs, 428
coset, 340
CosetLeadersMatFFE, 212
Cosets, 340
CosetTable, 449
CosetTableBySubgroup, 450
CosetTableDefaultLimit, 452
CosetTableDefaultMaxLimit, 451
CosetTableFromGensAndRelns, 451
CosetTableInWholeGroup, 453
CosetTableOfFpSemigroup, 535
Coset Tables and Coset Enumeration, 449
Coset tables for subgroups in the whole group, 453
CosetTableStandard, 453
CRC, 37
CrcFile, 94
example, 37
CRC Numbers, 37
CreateCompletionFiles, 35
Creating Character Tables, 692
Creating Class Functions from Values Lists, 742
Creating Class Functions using Groups, 743
Creating Finite Fields, 567
Creating Finely Presented Groups, 447
Creating Finely Presented Semigroups, 531
Creating Group Homomorphisms, 368
Creating Groups, 335
Creating Mappings, 292
Creating Permutations, 398
Creating Presentations, 464
Creation of Algebraic Extensions, 660
Creation of Rational Functions, 657
Credit, 22
CrystGroupDefaultAction, 419
Cycle, 389
CycleLength, 389
CycleLengths, 389
Cycles, 389
CycleStructureClass, 747
CycleStructurePerm, 398
CyclicExtensionsTom, 680
CyclicGroup, 492
CyclotomicField, 570
cyclotomic field elements, 153
CyclotomicPolynomial, 650
Cyclotomic Polynomials, 650
Index

Cyclotomics, 153

D

dv, 157
Darstellungsgruppe, see EpimorphismSchurCover, 365
DataType, 124
data type, unknown, 163
DayDMY, 249
DaysInMonth, 248
DaysInYear, 248
Debugging Recursion, 84
DEC, 235
DeclareAutoPackage, 822
DeclareHandlingByNiceBasis, 587
DeclareInfoClass, 80
DeclarePackage, 822
DeclarePackageAutoDocumentation, 822
DeclarePackageDocumentation, 822
DecodeTree, 481
DecodeTree, 481
decompose, a group character, 745
DecomposedFixedPointVector, 681
DecomposeTensorProduct, 630
Decomposition, 236
DecompositionInt, 236
DecompositionMatrix, 708
decomposition matrix, 235
Decompositions, 235
Decreased, 754
DefaultField, 559
    for cyclotomics, 156
    for finite field elements, 567
DefaultFieldByGenerators, 560
DefaultFieldOfMatrix, 218
DefaultFieldOfMatrixGroup, 414
DefaultRing, 543
    for finite field elements, 567
DefaultRingByGenerators, 544
DefaultStabChainOptions, 407
Defining a Pcgsg Yourself, 421
DefiningPolynomial, 561
DefiningQuotientHomomorphism, 459
DegreeFF, 566
DegreeIndeterminate, 648
DegreeOfBinaryRelation, 304
DegreeOfCharacter, 745
DegreeOfLaurentPolynomial, 647
DegreeOfTransformation, 536
DegreeOfTransformationSemigroup, 522
DegreeOverPrimeField, 561
Delta, 791
denominator, of a rational, 142
DenominatorCyc, 155
DenominatorOfModuloPcgs, 427
DenominatorOfRationalFunction, 645
DenominatorOfRat, 142
DepthOfPcElement, 423
DepthOfUpperTriangularMatrix, 226
Derangements, 147
Derivations, 615
Derivative, 649
DerivedLength, 353
DerivedSeriesOfGroup, 353
DerivedSubgroup, 346
DerivedSubgroupsTom, 679
DerivedSubgroupsTomPossible, 679
DerivedSubgroupsTomUnique, 679
DerivedSubgroupTom, 679
DescriptionOfRootOfUnity, 156
Determinant, 218
determinant character, 747
DeterminantMat, 218
DeterminantMatDestructive, 219
DeterminantMatDivFree, 219
DeterminantOfCharacter, 747
Developing rewriting systems, 334
DiagonalizeIntMatNormDriven, 232
DiagonalizeMat, 224
DiagonalMat, 220
DiagonalOfMat, 226
Difference, 264
DifferenceBlist, 205
DihedralGroup, 493
Dimension, 557
DimensionOfHighestWeightModule, 631
DimensionOfMatrixGroup, 414
DimensionOfVectors, 582
DimensionsLoewyFactors, 354
DimensionsMat, 218
Directories, 90
DirectoriesLibrary, 90
DirectoriesPackageLibrary, 823
DirectoriesPackagePrograms, 823
DirectoriesSystemPrograms, 91
Directory, 90
Index

DirectoryCurrent, 90
DirectoryTemporary, 90
DirectProduct, 487
DirectProductOp, 487
Direct Products, 487
DirectSumDecomposition, 601
   for Lie algebras, 619
Direct Sum Decompositions, 618
DirectSumOfAlgebraModules, 612
   for Lie algebras, 634
DirectSumOfAlgebras, 600
DisableAttributeValueStoring, 122
disable automatic loading, 821
Discriminant, 649
   for character tables, 712, 741
   for tables of marks, 673
DisplayCacheStats, 83
DisplayCompositionSeries, 353
DisplayEggBoxOfDClass, 524
DisplayInfInvariants, 513
DisplayInformationPerfectGroups, 506
DisplayOptions, 713
DisplayOptionsStack, 87
DisplayProfile, 82
DisplayRevision, 84
DistancesDistributionMatFFEVecFFE, 212
DistancesDistributionVecFFEVecFFE, 212
DistanceVecFFE, 212
Distinguished Subalgebras, 616
division, 48
   operation, 285
DivisionRingByGenerators, 560
division rings, 559
divisors, of an integer, 132
DivisorsInt, 132
Dixon-Schneider algorithm, 717
DixonInit, 718
DixonRecord, 718
DixonSplit, 718
Dixonini, 718
DMYDay, 249
DMYhmsSeconds, 250
DnLattice, 755
DnLatticeIterative, 756
do, 52
document formats (text, dvi, ps, pdf, html), 24
Domain, 281
   DomainByGenerators, 281
   Domain Categories, 278
   Domains, 110
   Domains Generated by Class Functions, 768
   Domains of Subspaces of Vector Spaces, 574
   DominantCharacter, 630
   DominantWeights, 630
   DoubleCoset, 342
   DoubleCosetRepsAndSizes, 342
   Double Cosets, 342
   DoubleCosets, operation, 342
   DoubleCosetsNC, operation, 342
doublequote character, 242
doublequotes, 240
DownEnv, 71
Dummy Streams, 105
duplicate free, 188
DuplicateFreeList, 193
Duplication of Lists, 175
Duplication of Objects, 113
DxIncludeIrreducibles, 719

E
E, 153
cN, 157
Earns, 390
EB, 157
EC, 157
Echelonized Matrices, 224
ED, 157
Edit, 74
Editing Files, 74
Editor Support, 75
EE, 157
EF, 157
Efficiency of Homomorphisms, 371
EG, 157
EggBoxOfDClass, 524
EH, 157
EI, 158
Eigenspaces, 223
Eigenvalues, 223
EigenvaluesChar, 748
Eigenvectors, 223
Eigenvectors and eigenvalues, 223
EJ, 158
EK, 158
EL, 158
Index

ElementaryAbelianGroup, 493
ElementaryAbelianSeries, 353
ElementaryAbelianSeriesLargeSteps, 353
Elementary Divisors, 223
ElementaryDivisorsMat, 223
Elementary Operations for a Pcg, 422
Elementary Operations for a Pcg and an Element, 422
Elementary Operations for Integers, 125
Elementary Operations for Rationals, 141
Elementary Tietze Transformations, 475
ElementOfFpGroup, 449
ElementOfFpSemigroup, 533
ElementOfMagmaRing, 638
ElementOrdersPowerMap, 771
ElementProperty, 411
Elements, 261
  of a list or collection, 261
Elements as equivalence classes, 109
ElementsFamily, 257
Elements in Algebraic Extensions, 660
Elements of Free Magma Rings, 637
Elements of pc groups, 437
ElementsStabChain, 409
Elements with Prescribed Images, 385
  element test, for lists, 177
elif, 51
EliminatedWord, 324
else, 51
Em, 158
emacs, 75
Embedding, 294
  example for direct products, 487
  example for semidirect products, 489
  example for wreath products, 490
  for group products, 491
  for Lie algebras, 614
  for magma rings, 638
embeddings, find all, 377
Embeddings and Projections for Group Products, 491
EmptyBinaryRelation, 303
EmptyMatrix, 219
EmptySTable, 591
EmptyStabChain, 410
EnableAttributeValueStoring, 122
End, 585
end, 55
Enlarging Internally Represented Lists, 177
Enumerator, 258
EnumeratorByBasis, 578
Enumerators, 203
EnumeratorSorted, 258
environment, 55
EpimorphismNilpotentQuotient, 461
EpimorphismPGroup, 460
EpimorphismQuotientSystem, 460
epimorphisms, find all, 377
EpimorphismSchurCover, 365
equality, associative words, 322
  elements of finitely presented groups, 447
  nonassociative words, 315
  of records, 254
  operation, 284
  pcwords, 437
Equivalence and Comparison of Domains, 276
equality test, 47
  for permutations, 396
equivalence class, 306
Equivalence Classes, 306
EquivalenceClasses, attribute, 307
EquivalenceClassOfElement, 307
EquivalenceClassOfElementNC, 307
EquivalenceClassRelation, 307
equivalence relation, 304, 305
equivalence relation, 304, 305
EquivalenceRelationByPairs, 306
EquivalenceRelationByPairsNC, 306
EquivalenceRelationByPartition, 306
EquivalenceRelationByPartitionNC, 306
EquivalenceRelationByProperty, 306
EquivalenceRelationByRelation, 306
EquivalenceRelationPartition, 306
Equivalence Relations, 305
ER, 158
Error, 72
ErrorCount, 72
ErrorNoTraceBack, 68
ers, syntax, 64
ES, 158
escaped characters, 242
escaping non-special characters, 242
ET, 158
EU, 158
EuclideanDegree, 551
EuclideanQuotient, 551
EuclideanRemainder, 551
Euclidean Rings, 551
Euler’s totient function, 134
EulerianFunction, 352
EulerianFunctionByTom, 681
EV, 158
EvalStraightLineProgElm, 331
EvalString, 248
evaluation, 42
strings, 247
EW, 158
EX, 158
ExactSizeConsiderFunction, 362
Exec, 108
Execution, 49
exit, 72
expanded form of monomials, 656
Expert Windows installation, 809
Exponent, 352
for character tables, 701
exponent, of the prime residue group, 134
exponentiation, operation, 285
ExponentOfPcElement, 422
ExponentsConjugateLayer, 424
ExponentsOfCommutator, 424
ExponentsOfConjugate, 424
ExponentsOfPcElement, 422
ExponentsOfRelativePower, 424
Exponents of Special Products, 424
ExponentSumWord, 323
ExponentSyllable, 324
Expressing Group Elements as Words in Generators, 339
Expressions, 42
ExtendedPcgs, 426
ExtendStabChain, 410
Extension, 442
ExtensionNC, 442
ExtensionRepresentatives, 443
Expansions, 442
Extensions of the p-adic Numbers, 663
ExteriorPowerOfAlgebraModule, 634
ExternalOrbit, 393
ExternalOrbits, 393
ExternalOrbitsStabilizers, 393
External Representation for Nonassociative Words, 318
external representation of polynomials, 656
ExternalSet, 392
External Sets, 391
ExternalSubset, 392
Extract, 752
ExtraspecialGroup, 493
ExtRepDenominatorRatFun, 657
ExtRepNumeratorRatFun, 657
ExtRepOfObj, external representation, for
cyclotomics, 156
ExtRepPolynomialRatFun, 657
EY, 158
F
fN, 157
FactorCosetAction, 387
for fp groups, 450
FactorCosetOperation, 450
FactorFreeSemigroupByRelations, 531
FactorGroup, 355
FactorGroupFpGroupByRels, 447
FactorGroupNC, 355
FactorGroupNormalSubgroupClasses, 733
Factor Groups, 355
Factor Groups of Polycyclic Groups - Modulo Pcgs, 426
Factor Groups of Polycyclic Groups in their Own
Representation, 428
FactorGroupTom, 682
Factorial, 143
Factorization, 340
factorization, 339
Factors, 551
of univariate polynomial, 650
FactorsInt, 131
FactorsSquarefree, 651
Fail, 165
fail, 165
FaithfulModule, 611
for Lie algebras, 628
Families, 115
FamiliesOfGeneralMappingsAndRanges, 302
FamiliesOfRows, 728
FamilyForOrdering, 270
FamilyObj, 115
FamilyPcgs, 436
Index

FamilyRange, 302
FamilySource, 302
FAQ, 817
features, under UNIX, 28
fi, 51
Fibonacci, 151
Fibonacci and Lucas Sequences, 151
Field, 559
FieldExtension, 561
field homomorphisms, Frobenius, 568
FieldOfMatrixGroup, 414
FieldOverItselfByGenerators, 560
fields, 559
File Access, 92
FileDescriptorOfStream, 97
Filename, 91
Filename, 91
File Operations, 93
File Streams, 102
Filtered, 196
Filters, 116
Filters Controlling the Arithmetic Behaviour of
Lists, 179
Finding Positions in Lists, 185
Finding Submodules, 666
FindSL2, 626
Finite Field Elements, 565
Finitely Presented Lie Algebras, 626
Finitely presented monoids, 533
finiteness test, for a list or collection, 261
Finite Perfect Groups, 504
First, 197
FittingSubgroup, 346
Flat, 194
flush character, 242
For, 52
ForAll, 198
ForAny, 198
for loop, 52
FpErmComparisonMethod, 448
FpGroupPresentation, 464
FpGrpNonSmgOfFpGrpNonSmgElement, 531
FpLieAlgebraByCartanMatrix, 627
frame, 757
FrattiniSubgroup, 346
for groups with pegs, 434
FreeAlgebra, 590
FreeAlgebraWithOne, 590
FreeAssociativeAlgebra, 590
FreeAssociativeAlgebraWithOne, 590
FreeGeneratorsOfFpGroup, 448
FreeGeneratorsOfFpSemigroup, 532
FreeGeneratorsOfWholeGroup, 448
FreeGroup, 319
FreeGroupOfFpGroup, 448
Free Groups, Monoids and Semigroups, 319
FreeLeftModule, 557
FreeLieAlgebra, 615
FreeMagma, 317
FreeMagmaRing, 636
Free Magma Rings, 636
Free Magmas, 317
FreeMagmaWithOne, 317
Free Modules, 556
FreeMonoid, 527
with example, 320
FreeMonoidOfRewritingSystem, 535
FreeSemigroup, 320
with examples, 521
FreeSemigroupOfFpSemigroup, 532
FreeSemigroupOfRewritingSystem, 535
Frobenius automorphism, 568
FrobeniusAutomorphism, 568
FrobeniusAutomorphism, 568
FrobeniusCharacterValue, 766
ftp, 798
FullMatrixAlgebra, 593
FullMatrixAlgebraCentralizer, 601
FullMatrixLieAlgebra, 615
FullMatrixModule, 558
FullMatrixSpace, 582
FullRowModule, 558
FullRowSpace, 582
Function, 55
function, 55
FunctionAction, 392
function call, 46
with arguments, 46
with options, 46
Function Calls, 46
FunctionOperation, 394
functions, 61, 292
definition by arrow notation, 57
definition of, 55
recursive, 55
with a variable number of arguments, 46, 56
FunctionsFamily, 63
Functions for Coding Theory, 212
Functions for GAP Packages, 822
Functions that do nothing, 62
Function that Modify Boolean Lists, 206
Function Types, 62
FusionCharTableTom, 688
FusionConjugacyClasses, 773
FusionConjugacyClassesOp, 773
fusions, 773
FusionsAllowedByRestrictions, 788
FusionsTom, 677
G
G-sets, 380, 391
gv, 157
gac, 35
Galois Action, 561
Galois Conjugacy of Cyclotomics, 159
GaloisCyc, 159
for class functions, 741
GaloisField, 567
GaloisGroup, of field, 562
of rational class of a group, 344
Galois Groups of Abelian Number Fields, 571
GaloisMat, 161
GaloisStabilizer, 571
gap-trouble, email address, 818
gap.rc, 32, 34
GAP3, 34
Gap3CatalogueIdGroup, 502
GAP_ROOT_PATHS, 30, 90
GAPChars, 732
GAP for MacOS, 811
GAP for UNIX, 800
GAP for Windows, 806
GapInputPcGroup, 441
GapInputSCTable, 591
GAP package, permissions, 820
GAP Root Directory, 89
Gaussian algorithm, 222
GaussianIntegers, 572
GaussianRationals, 570
Gaussians, 572
Gcd, 552
Gcd and Lcm, 552
Gcdex, 128
GcdInt, 128
GcdOp, 552
GcdRepresentation, 552
GcdRepresentationOp, 552
General Binary Relations, 303
GeneralisedEigenspaces, 223
GeneralisedEigenvalues, 223
generalized characters, 735
GeneralisedEigenspaces, 223
GeneralisedEigenvalues, 223
GeneralLinearGroup, 494
GeneralMappingByElements, 292
General Mappings, 301
GeneralMappingsFamily, 302
GeneralOrthogonalGroup, 496
GeneralUnitaryGroup, 495
Generating Fields, 559
Generating modules, 555
Generating Rings, 543
generator, of the prime residue group, 136
GeneratorsOfAdditiveGroup, 541
GeneratorsOfAdditiveMagma, 541
GeneratorsOfAdditiveMagmaWithZero, 541
GeneratorsOfAlgebra, 597
GeneratorsOfAlgebraModule, 607
GeneratorsOfAlgebraWithOne, 597
GeneratorsOfDivisionRing, 560
GeneratorsOfDomain, 280
GeneratorsOfEquivalenceRelationPartition, 306
GeneratorsOfField, 560
GeneratorsOfGroup, 336
GeneratorsOfIdeal, 546
GeneratorsOfLeftIdeal, 547
GeneratorsOfLeftModule, 555
GeneratorsOfLeftOperatorAdditiveGroup, 555
GeneratorsOfLeftVectorSpace, 574
GeneratorsOfMagma, 311
GeneratorsOfMagmaWithInverses, 311
GeneratorsOfMagmaWithOne, 311
GeneratorsOfMonoid, 527
GeneratorsOfNearAdditiveGroup, 541
GeneratorsOfNearAdditiveMagma, 541
GeneratorsOfNearAdditiveMagmaWithZero, 541
GeneratorsOfPresentation, 464
GeneratorsOfRightIdeal, 547
GeneratorsOfRightModule, 556
GeneratorsOfRightOperatorAdditiveGroup, 555
GeneratorsOfRing, 544
Index

GeneratorsOfRingWithOne, 548
GeneratorsOfRws, 333
GeneratorsOfSemigroup, 520
GeneratorsOfStruct, 277
GeneratorsOfTwoSidedIdeal, 546
GeneratorsOfVectorSpace, 574
GeneratorsPrimeResidues, 135
GeneratorsSmallest, 362
GeneratorsSubgroupsTom, 685
GeneratorSyllable, 324
Generic Construction of Tables of Marks, 689
GetFusionMap, 775
Getting GAP, 798
getting help, 23
GL, 494
GL and SL, 415
GModuleByMats, 665
GO, 496
GQuotients, 377
Grading, 602
Green's Relations, 523
GreensDClasses, 524
GreensDClassOfElement, 524
GreensDRelation, 523
GreensHClasses, 524
GreensHClassOfElement, 524
GreensHRelation, 523
GreensJClasses, 524
GreensJClassOfElement, 524
GreensJRelation, 523
GreensLClasses, 524
GreensLClassOfElement, 524
GreensLRelation, 523
GreensRClasses, 524
GreensRClassOfElement, 524
GreensRRelation, 523
Group, 335
group actions, 380, 381
group algebra, 636
Group Automorphisms, 373
GroupByRws, 439
GroupByRwsNC, 439
group characters, 735
Group Elements, 335
GroupGeneralMappingByImages, 369
GroupHClassOfGreensDClass, 525
GroupHomomorphismByFunction, 369
GroupHomomorphismByImages, 368
GroupHomomorphismByImagesNC, 368
GroupOfPcgs, 422
group operations, 381, 394
Group Properties, 349
GroupRing, 636
group ring, 636
Groups of Automorphisms, 375
GroupStabChain, 409
GroupWithGenerators, 336
GU, 495
H
hN, 157
HallSubgroup, 348
HallSystem, 349
for groups with pcgs, 434
Handling of Streams in the Background, 105
HasAbelianFactorGroup, 356
HasElementaryAbelianFactorGroup, 356
HasIndeterminateName, 643
HasParent, 279
HasseDiagramBinaryRelation, 305
HeadPcElementByNumber, 424
HenselBound, 651
HermiteNormalFormIntegerMat, 234
HermiteNormalFormIntegerMatInverses
Transforms, 234
HermiteNormalFormIntegerMatTransforms, 234
HeuristicCancelPolynomials, 659
HexStringInt, 245
HighestWeightModule, 633
History of Character Theory Stuff in GAP, 692
HMSMSec, 249
HNFCollins, 233
HNFDriver, 233
HNFDriver, 233
Hom, 585
HomeEnumerator, 392
Homomorphism for very large groups, 372
HomomorphismQuotientSemigroup, 523
homomorphisms, find all, 376
homomorphisms, Frobenius, field, 568
Homomorphisms of Algebras, 602
HomomorphismTransformationSemigroup, 522
How to Implement New Kinds of Vector Spaces, 587
HP-UX, 800
HTML Font Setup, 816
Index
Index

InertiaSubgroup, 746
Infinity, 156
infinity, 156
inflated class functions, 748
Info, 80
InfoAlgebra, 589
InfoAttributes, 122
InfoBckt, 412
InfoCharacterTable, 695
InfoCoh, 365
InfoComplement, 345
InfoCoset, 343
InfoFpGroup, 447
Info Functions, 80
InfoGroup, 336
InfoLattice, 360
InfoLevel, 80
InfoMatrix, 215
InfoMonomial, 790
InfoNumtheor, 134
InfoOptions, 87
InfoPcSubgroup, 362
Information about a function, 61
Information about the version used, 84
InfoText, 703
InfoTom, 675
InfoWarning, 81
InitFusion, 787
InitPowerMap, 785
InjectionZeroMagma, 310
InnerAutomorphism, 374
InnerAutomorphismNC, 374
InnerAutomorphismsAutomorphismGroup, 375
inner product, of group characters, 745
Input-Output Streams, 103
InputLogTo, 94
for streams, 101
stop logging input, 94
InputOutputLocalProcess, 104
InputTextFile, 102
InputTextNone, 105
InputTextString, 103
InputTextUser, 103
InsertTrivialStabilizer, 410
InstallAtExit, 73
installation, 797
installation, under UNIX, 800
under Windows, 807
Installation for the Impatient, 797
Installation of GAP for MacOS, 811
Installation of GAP for UNIX, 800
Installation of GAP for Windows, 807
InstallCharReadHookFunc, 105
InstalledPackageVersion, 823
InstallFactorMaintenance, 287
InstallHandlingByNiceBasis, 587
Installing a GAP Package in your home directory, 820
Installing GAP Packages, 819
InstallIsomorphismMaintenance, 287
InstallSubsetMaintenance, 287
Int, 125
for cyclotomics, 154
for strings, 247
integer part of a quotient, 128
Integers, 125
Integral Bases for Abelian Number Fields, 570
IntegratedMat, 236
IntegratedStraightLineProgram, 330
Interface to the CAS System, 728
Interface to the MOC System, 729
IntermediateGroup, 355
IntermediateSubgroups, 355
Internally Represented Cyclotomics, 162
Internally Represented Strings, 243
InterpolatedPolynomial, 554
IntersectBlist, 206
Intersection, 263
for groups with pcgs, 434
intersection, of collections, 263
of sets, 192
Intersection2, 263
IntersectionBlist, 205
IntersectionsTom, 681
IntersectSet, 192
IntFFE, 567
IntHexString, 247
IntScalarProducts, 783
IntVecFFE, 567
InvariantBilinearForm, 416
InvariantElementaryAbelianSeries, 354
Invariant Forms, 416
InvariantLattice, 417
InvariantQuadraticForm, 416
InvariantSesquilinearForm, 416
InvariantSubgroupsElementaryAbelianGroup, 360
Inverse, 283
inverse, group homomorphism, 370
matrix, 217
of class function, 740
InverseAttr, 283
InverseClasses, 704
InverseGeneralMapping, 293
InverseImmutable, 283
InverseMap, 778
InverseMatMod, 230
InverseMutable, 283
InverseOp, 283
InverseRepresentative, 409
InverseSameMutability, 283
InverseSM, 283
Invoking the Help, 23
IRIX, 800
Irr, 699
irrationals, 153
IrrBaumClausen, 715
IrrComion, 715
IrrDixonSchneider, 715
Irreducibility Tests, 666
irreducible character, 744
irreducible characters, computation, 718
IrreducibleDifferences, 751
Irreducible Maximal Finite Integral Matrix Groups, 511
IrreducibleModules, for groups with pcgs, 434
IrreducibleRepresentations, 716
IrreducibleRepresentationsDixon, 717
IrreducibleSolvableGroup, 511
IrreducibleSolvableGroupMS, 510
Irreducible Solvable Matrix Groups, 510
Is16BitsFamily, 326
Is32BitsFamily, 326
Is8BitsFamily, 326
IsAbelian, 312
for character tables, 701
IsAbelianNumberField, 570
IsAbelianNumberFieldPolynomialRing, 653
IsAbelianTom, 679
IsAdditiveElement, 288
IsAdditiveElementWithInverse, 288
IsAdditiveElementWithZero, 288
IsAdditiveGroup, 540
IsAdditiveGroupGeneralMapping, 300
IsAdditiveGroupHomomorphism, 300
IsAdditivelyCommutative, 541
IsAdditivelyCommutativeElement, 290
IsAdditivelyCommutativeElementCollColl, 290
IsAdditivelyCommutativeElementCollection, 290
IsAdditivelyCommutativeElementFamily, 290
IsAdditiveMagma, 539
IsAdditiveMagmaWithInverses, 540
IsAdditiveMagmaWithZero, 539
IsAlgebra, 596
IsAlgebraGeneralMapping, 301
IsAlgebraHomomorphism, 301
IsAlgebraicElement, 661
IsAlgebraicExtension, 660
IsAlgebraModuleElement, 608
IsAlgebraModuleElementCollection, 608
IsAlgebraModuleElementFamily, 608
IsAlgebraWithOne, 596
IsAlgebraWithOneGeneralMapping, 301
IsAlgebraWithOneHomomorphism, 301
IsAlphaChar, 244
IsAlternatingGroup, 402
IsAnticommutative, 549
IsAntisymmetricBinaryRelation, 304
IsAssociated, 549
IsAssociative, 312
IsAssociativeElement, 290
IsAssociativeElementCollColl, 290
IsAssociativeElementCollection, 290
IsAssocWord, 319
IsAssocWordWithInverse, 319
IsAssocWordWithOne, 319
IsAutomorphismGroup, 375
IsBasicWreathLessThanOrEqual, 322
IsBasicWreathProductOrdering, 274
IsBasis, 576
IsBasisByNiceBasis, 587
IsBasisOfAlgebraModuleElementSpace, 609
IsBergerCondition, 791
IsBijective, 295
IsBinaryRelation, 303
same as IsEndoGeneralMapping, 303
IsBLetterAssocWordRep, 325
IsBLetterWordsFamily, 325
IsBlist, 204
IsBlockMatrixRep, 230
IsInputOutputStream, 104
IsInputStream, 96
IsInputTextNone, 96
IsInputTextStream, 96
IsInt, 125
IsIntegerMatrixGroup, 417
IsIntegers, 125
IsIntegralBasis, 579
IsIntegralCyclotomic, 154
IsIntegralRing, 548
IsInternallyConsistent, 114
for character tables, 709
for tables of marks, 679
IsIrreducibleCharacter, 744
IsIrreducibleRingElement, 550
IsIterator, 267
IsJacobianElement, 290
IsJacobianElementCollColl, 290
IsJacobianElementCollection, 290
IsJacobianRing, 549
IsLaurentPolynomial, 646
IsLaurentPolynomialDefaultRep, 657
IsLDistributive, 548
IsLeftAlgebraModuleElement, 608
IsLeftAlgebraModuleElementCollection, 608
IsLeftIdeal, 546
IsLeftIdealInParent, 546
IsLeftModule, 555
IsLeftModuleGeneralMapping, 300
IsLeftModuleHomomorphism, 300
IsLeftOperatorAdditiveGroup, 555
IsLeftSemigroupIdeal, 522
IsLeftSemigroupIdealEnumerator, 522
IsLeftVectorSpace, 573
IsLessThanOrEqualUnder, 270
IsLessThanUnder, 270
IsLetterAssocWordRep, 325
IsLetterWordsFamily, 325
IsLexicographicallyLess, 194
IsLieAbelian, 618
IsLieAlgebra, 596
IsLieMatrix, 216
IsLieNilpotent, 618
IsLieObject, 613
IsLieObjectCollection, 613
IsLieSolvable, 618
IsLinearMapping, 300
IsLinearMappingsModule, 586
IsList, 168
IsListDefault, 179
IsListOrCollection, 258
IsLowerAlphaChar, 244
IsLowerTriangularMat, 219
IsMagma, 308
IsMagmaHomomorphism, 298
IsMagmaRingModuloRelations, 639
IsMagmaRingModuloSpanOfZero, 639
IsMagmaWithInverses, 308
IsMagmaWithInversesIfNonzero, 308
IsMagmaWithOne, 308
IsMapping, 294
IsMatchingSublist, 188
IsMatrix, 215
IsMatrixGroup, 414
IsMatrixModule, 558
IsMatrixSpace, 581
IsMinimalNonMonomial, 796
IsModuloPcgs, 427
IsMonoid, 527
IsMonomial, for characters, 793
for character tables, 701
for groups, 793
for positive integers, 794
IsMonomialGroup, 350
IsMonomialMatrix, 219
IsMonomialNumber, 794
IsMultiplicativeElement, 288
IsMultiplicativeElementWithInverse, 289
IsMultiplicativeElementWithOne, 288
IsMultiplicativeElementWithZero, 288
IsMultiplicativeGeneralizedRowVector, 179
IsMultiplicativeZero, 312
IsMutable, 112
IsMutableBasis, 580
IsNaturalAlternatingGroup, 401
IsNaturalGL, 415
IsNaturalGLnZ, 417
IsNaturalSL, 416
IsNaturalSLnZ, 417
IsNaturalSymmetricGroup, 401
IsNearAdditiveElement, 288
IsNearAdditiveElementWithInverse, 288
IsNearAdditiveElementWithZero, 288
IsNearAdditiveGroup, 539
IsNearAdditiveMagma, 539
IsNearAdditiveMagmaWithInverses, 539
Index

IsPrimitivePolynomial, 647
IsPrimitiveRootMod, 136
IsProbablyPrimeInt, 130
IsPSolvable, 351
IsPSolvableCanonicalBasisFullHomModule, 586
IsPSolvableCharacterTable, 710
IsPSolvableCharacterTableOp, 710
IsPurePadicNumber, 662
IsPurePadicNumberFamily, 662
IsQuasiPrimitive, 792
IsQuaternion, 597
IsQuaternionCollColl, 597
IsQuaternionCollection, 597
IsQuickPositionList, 203
IsQuotientSemigroup, 523
IsRange, 202
IsRat, 141
IsRationalFunction, 645
IsRationalFunctionDefaultRep, 656
IsRationalFunctionsFamily, 655
IsRationalMatrixGroup, 417
IsRationals, 141
IsRationalesPolynomialRing, 653
IsRDistributive, 549
IsReadableFile, 92
IsRecord, 251
IsRecordCollColl, 251
IsRecordCollection, 251
IsReduced, 333
IsReductionOrdering, 271
IsReesCongruence, 523
IsReesCongruenceSemigroup, 521
IsReesMatrixSemigroup, 525
IsReesMatrixSemigroupElement, 525
IsReesZeroMatrixSemigroup, 525
IsReesZeroMatrixSemigroupElement, 525
IsReflexiveBinaryRelation, 303
IsRegular, 390
IsRegularDClass, 525
IsRegularSemigroup, 521
IsRegularSemigroupElement, 521
IsRelativelySM, 795
IsRestrictedLieAlgebra, 623
IsRewritingSystem, 332
IsRightAlgebraModuleElement, 608
IsRightAlgebraModuleElementCollection, 608
IsRightCoset, 341
IsRightIdeal, 546
IsRightIdealInParent, 546
IsRightModule, 555
IsRightOperatorAdditiveGroup, 555
IsRightSemigroupIdeal, 522
IsRightSemigroupIdeal Enumerator, 522
IsRing, 543
IsRingElement, 289
IsRingElementWithInverse, 289
IsRingElementWithOne, 289
IsRingGeneralMapping, 301
IsRingHomomorphism, 301
IsRingWithOne, 547
IsRingWithOneGeneralMapping, 301
IsRingWithOneHomomorphism, 301
IsRootSystem, 619
IsRootSystemFromLieAlgebra, 620
IsRovModule, 558
IsRovSpace, 581
IsRovVector, 208
IsScalar, 289
IsSemiEchelonized, 582
IsSemigroup, 520
IsSemigroupCongruence, 523
IsSemigroupIdeal, 522
IsSemigroupIdealEnumerator, 522
IsSemiRegular, 389
IsSet, 189
IsShortLexLessThanOrEqual, 322
IsShortLexOrdering, 272
IsSimple, for character tables, 701
IsSimpleAlgebra, 596
IsSimpleGroup, 350
IsSimpleSemigroup, 521
IsSingleValued, 294
IsSL, 415
IsSolvable, for character tables, 701
IsSolvableGroup, 350
IsSolvableTom, 679
IsSortedList, 188
IsSpecialLinearGroup, 415
IsSpecialPcgs, 431
IsSPGeneralMapping, 302
IsSSortedList, 189
IsStandardGeneratorsOfGroup, 684
IsStraightLineProgElm, 331
IsStraightLineProgram, 327
IsStream, 96
Index

LGLength, 431
LGWeights, 431
library tables, 692
LieAlgebra, 614
LieAlgebraByStructureConstants, 614
LieBracket, 285
LieCenter, 616
LieCentralizer, 616
LieCentre, 616
LieCoboundaryOperator, 629
LieDerivedSeries, 617
LieDerivedSubalgebra, 616
LieFamily, 613
LieLowerCentralSeries, 617
LieNilRadical, 617
LieNormalizer, 616
LieObject, 613
Lie objects, 613
LieSolvableRadical, 617
LieUpperCentralSeries, 617
LiftedInducedPcgs, 428
LiftedPcElement, 428
LinearAction, 432
LinearActionLayer, 432
LinearCharacters, 700
LinearCombination, 578
LinearCombinationPcgs, 423
LinearIndependentColumns, 236
Linear Mappings, 300
LinearOperation, 432
LinearOperationLayer, 432
Line Editing, 73
LinesOfStraightLineProgram, 328
Linux, 800
List, 196
list and non-list, difference, 181
left quotient, 184
mod, 183
product, 182
quotient, 183
List Assignment, 171
list assignment, operation, 170
ListBlist, 204
list boundedness test, operation, 170
List Categories, 168
list element, access, 170
assignment, 171
operation, 170
List Elements, 170
list equal, comparison, 177
ListN, 199
list of available books, 24
ListPerm, 398
Lists and Collections, 258
list smaller, comparison, 178
ListStabChain, 409
list unbind, operation, 170
ListWithIdenticalEntries, 185
ListX, 199
LLL, 751
LLL algorithm, for Gram matrices, 237
for vectors, 236
for virtual characters, 751
LLLReducedBasis, 236
LLLReducedGramMat, 237
LoadDynamicModule, 35
Loading a GAP Package, 821
loading a saved workspace, 37
local, 55
logarithm, discrete, 135
of a root of unity, 156
LogFFE, 567
logical, 165
logical operations, 166
LogInt, 127
LogMod, 135
LogTo, 94
for streams, 101
stop logging, 94
LongestWeylWordPerm, 622
loop, read eval print, 64
loop, for, 52
repeat, 52
while, 51
loop over iterator, 54
loop over object, 54
loop over range, 53
loops, leaving, 54
restarting, 55
LowercaseString, 245
LowerCentralSeriesOfGroup, 354
Low Index Subgroups, 454
LowIndexSubgroupsFpGroup, 454
Low Level Routines to Modify and Create Stabilizer Chains, 410
Lucas, 151
Index

M
mX, 158
Macintosh, 811
MacOS, 811
Magma, 308
MagmaByGenerators, 309
MagmaByMultiplicationTable, 310
Magma Categories, 308
MagmaElement, 310
Magma Generation, 308
MagmaHomomorphismByFunctionNC, 298
Magma Homomorphisms, 298
MagmaRingModuloSpanOfZero, 639
Magma Rings modulo Relations, 639
Magma Rings modulo the Span of a Zero Element, 639
Magma Defined by Multiplication Tables, 310
MagmaWithInverses, 309
MagmaWithInversesByGenerators, 309
MagmaWithInversesByMultiplicationTable, 310
MagmaWithOne, 308
MagmaWithOneByGenerators, 309
MagmaWithOneByMultiplicationTable, 310
Main Loop, 64
MakeConfluent, 333
MakeImmutable, 112
MAKElb11, 729
MakeReadOnlyGlobal, 44
MakeWriteGlobal, 44
Making transformation semigroups, 522
Manual Conventions, 21
map, parametrized, 777
MappedWord, 316
MappingByFunction, 292
MappingPermListList, 308
Mappings that Respect Addition, 299
Mappings that Respect Multiplication, 298
Mappings which are Compatible with Algebraic Structures, 298
maps, 769
MarksTom, 675
MatAlgebra, 593
MatClassMultCoeffsCharTable, 712
MathieuGroup, 494
MatLieAlgebra, 615
matrices, commutator, 217
Matrices as Basis of a Row Space, 225
Matrices as Linear Mappings, 226
Matrices over Finite Fields, 228
Matrices Representing Linear Equations and the Gaussian Algorithm, 222
MatrixAlgebra, 593
MatrixAutomorphisms, 726
matrix automorphisms, 772
MatrixByBlockMatrix, 230
Matrix Constructions, 219
Matrix Groups in Characteristic 0, 417
MatrixLieAlgebra, 615
MatrixOfAction, 609
matrix spaces, 581
MatScalarProducts, 745
MatTom, 678
MaximalAbelianQuotient, 356
MaximalBlocks, 390
MaximalNormalSubgroups, 357
MaximalSubgroupClassReps, 357
MaximalSubgroups, 357
for groups with pcgs, 434
MaximalSubgroupsLattice, 358
MaximalSubgroupsTom, 682
Maximum, 195
MaximumList, 195
MeatAxe Modules, 665
MeetEquivalenceRelations, 306
MeetMaps, 780
Membership Test for Collections, 264
Membership Test for Lists, 177
MinimalElementCosetStabChain, 409
MinimalGeneratingSet, 363
for groups with pcgs, 434
MinimalNonmonomialGroup, 796
Minimal Nonmonomial Groups, 796
MinimalPolynomial, 650
over a field, 562
over a ring, 650
Minimal Polynomials, 650
MinimalStabChain, 407
MinimalSupergroupsLattice, 359
MinimalSupergroupsTom, 682
MinimizedBombieriNorm, 651
Minimum, 195
MinimumList, 195
MinusCharacter, 786
Miscellaneous, 139
MOCChars, 732
MOCString, 731
MOCTable, 730
mod, integers, 132
  laurent polynomials, 644
  lists, 183
  rationals, 48
mod, 48
  arithmetic operators, 48
  for character tables, 698
  residue class rings, 132
modular inverse, 48
modular remainder, 48
modular roots, 138
ModuleByRestriction, 611
Module Homomorphisms, 667
ModuleOfExtension, 442
Modules over Lie Algebras and Their Cohomology, 628
Modules over Semisimple Lie Algebras, 630
modulo, 48
  arithmetic operators, 48
  for pcgs, 427
  residue class rings, 132
ModuloPcgs, 427
MoebiusMu, 138
MoebiusTom, 678
Molien Series, 758
MolienSeries, 758
MolienSeriesInfo, 759
MolienSeriesWithGivenDenominator, 760
Monoid, 527
MonoidByGenerators, 527
MonoidByMultiplicationTable, 528
MonoidOfRewritingSystem, 534
MonomialRevLexicoLess, 655
MonomialTotalDegreeLess, 654
monomorphisms, find all, 377
MorClassLoop, 377
More about Boolean Lists, 206
More About Global Variables, 44
More about Tables of Marks, 670
MostFrequentGeneratorFpGroup, 452
MovedPoints, 397
Moved Points of Permutations, 397
MTX.BasesCompositionSeries, 666
MTX.BasesMaximalSubmodules, 666
MTX.BasesMinimalSubmodules, 666
MTX.BasesMinimalSupermodules, 666
MTX.BasesSubmodules, 666
MTX.BasisRadical, 666
MTX.BasisSocle, 666
MTX.CollectFactors, 667
MTX.CompositionFactors, 666
MTX.DegreeSplittingField, 666
MTX.Dimension, 665
MTX.Distinguish, 668
MTX.Field, 665
MTX.Generators, 665
MTX.Homomorphism, 667
MTX.Homomorphisms, 668
MTX.InducedAction, 667
MTX.InducedActionFactorMatrix, 667
MTX.InducedActionFactorModule, 667
MTX.InducedActionMatrix, 667
MTX.InducedActionMatrixNB, 667
MTX.InducedActionSubmodule, 667
MTX.InducedActionSubmoduleNB, 667
MTX.IsAbsolutelyIrreducible, 666
MTX.IsEquivalent, 667
MTX.IsIrreducible, 666
MTX.Isomorphism, 667
MTX.NormedBasisAndBaseChange, 667
MTX.ProperSubmoduleBasis, 666
MTX.SubmoduleGModule, 666
multiplication, 48
  matrices, 217
  matrix and matrix list, 218
  matrix and scalar, 216
  matrix and vector, 217
  operation, 285
  scalar and matrix, 216
  scalar and matrix list, 217
  scalar and vector, 209
  vector and matrix, 216
  vector and matrix list, 218
  vector and scalar, 209
  vectors, 209
MultiplicationTable, 310
Multiplicative Arithmetic for Lists, 182
Multiplicative Arithmetic Functions, 138
MultiplicativeNeutralElement, 312
multiplicative order of an integer, 135
MultiplicativeZero, 312
MultiplicativeZeroOp, 282
multiplicity, of constituents of a group character, 745
multiplier, 366
multisets, 190
Multivariate Polynomials, 649
MultRowVector, 211
Murnaghan components, 757, 758
Mutability and Copyability, 111
Mutable Bases, 579
MutableBasis, 580
MutableBasisOfClosureUnderAction, 599
MutableBasisOfIdealInNonassociativeAlgebra, 600
MutableBasisOfNonassociativeAlgebra, 600
MutableIdentityMat, 221
MutableNullMat, 221

N
n, 158
Name, 114
NameFunction, 61
NameRNam, 256
NamesFilter, 117
NamesGVars, 45
NamesOfFusionSources, 776
NamesSystemGVars, 45
NamesUserGVars, 45
NaturalEmbeddings related to Magma Rings, 638
NaturalHomomorphismByGenerators, 298
NaturalHomomorphismByIdeal, 604
NaturalHomomorphismByNormalSubgroup, 355
NaturalHomomorphismByNormalSubgroupNC, 355
NaturalHomomorphismBySubAlgebraModule, 611
NaturalHomomorphismBySubspace, 585
NearAdditiveGroup, 540
NearAdditiveGroupByGenerators, 540
NearAdditiveMagma, 540
NearAdditiveMagmaByGenerators, 540
NearAdditiveMagmaWithZero, 540
NearlyCharacterTablesFamily, 696
negative number, 48
NegativeRoots, 620
NegativeRootVectors, 620
NestingDepth, 180
NestingDepthM, 180
NewInfoClass, 80
newline, 40
newline character, 242
NewmanInfinityCriterion, 463
New Presentations and Presentations for Subgroups, 458
NextIterator, 267
NextPrimeInt, 130
NiceBasis, 587
NiceBasisFiltersInfo, 588
NiceFreeLeftModule, 586
NiceFreeLeftModuleInfo, 587
NiceMonomorphism, 373
NiceMonomorphismAutomGroup, 376
Nice Monomorphisms, 372
NiceObject, 373
NiceVector, 586
NilpotencyClassOfGroup, 349
NilpotentQuotientOffpLieAlgebra, 627
NK, 158
NOAUTO, 822
NonnegativeIntegers, 125
NonnegIntScalarProducts, 783
NonNilpotentElement, 625
Norm, 562
of character, 745
NormalBase, 563
NormalClosure, 345
NormalFormIntMat, 234
Normal Forms over the Integers, 232
NormalIntersection, 345
NormalizedElementOfMagmaRingModulo-Relations, 639
NormalizedWhitespace, 246
Normalizer, 345
normalizer, 344
NormalizerInGLnZ, 418
NormalizerInGLnZBravaisGroup, 418
NormalizersTom, 680
NormalizerTom, 680
NormalizeWhitespace, 246
NormalSeriesByWhitespace, 429
Normal Structure, 344
NormalSubgroupClasses, 733
NormalSubgroupClassesInfo, 732
NormalSubgroups, 357
NormedRowVector, 209
NormedRowVectors, 583
not, 167
NrArrangements, 145
NrBasisVectors, 580
NrCombinations, 145
NrConjugacyClasses, 344
for character tables, 701
NrConjugacyClassesGL, 498
NrConjugacyClassesGU, 498
NrConjugacyClassesPGL, 498
NrConjugacyClassesPGU, 498
NrConjugacyClassesPSL, 498
NrConjugacyClassesPSU, 498
NrConjugacyClassesSL, 498
NrConjugacyClassesSU, 498
NrConjugacyClassesSLSIsogeneous, 498
NrConjugacyClassesSUSIsogeneous, 498
NrDerangements, 148
NrInputsOfStraightLineProgram, 328
NrMovedPoints, 397
NrOrderedPartitions, 149
N Partitions, 149
NrPartitionsSet, 148
NrPartitionTuples, 151
NrPermutationsList, 147
NrPolyhedralSubgroups, 711
NrPrimitiveGroups, 509
NrRestrictedPartitions, 150
NrSubsTom, 676
NrTransitiveGroups, 500
NrTuples, 147
NrUnorderedTuples, 146
NullAlgebra, 593
NullMat, 219
NullspaceMat, 222
NullspaceMatDestructive, 222
NullspaceModQ, 230
Number, 197
number, Bell, 144
binomial, 143
Stirling, of the first kind, 144
Stirling, of the second kind, 145
NumberArgumentsFunction, 61
NumberFFVector, 211
number field, 570
number fields, Galois group, 571
NumberIrreducibleSolvableGroups, 510
NumberPerfectGroups, 505
NumberPerfectLibraryGroups, 505
NumberSmallGroups, 502
NumberSyllables, 324
numerator, of a rational, 142
NumeratorOfModuloPcgs, 427
NumeratorOfRationalFunction, 645
NumeratorRat, 142
Numerical Group Attributes, 352
O
$O_n(G)$, see PCore, 345
ObjByExtRep, 632
Objects, 109
OCOneCocycles, 365
octal character codes, 242
OctaveAlgebra, 593
od, 52
OldGeneratorsOfPresentation, 480
Omega, 349
ONanScottType, 402
OnBreak, 68
OnBreakMessage, 69
One, 281
OneAttr, 281
OneCoboundaries, 364
OneCocycles, 363
one cohomology, 363
OneFactorBound, 651
OneImmutable, 281
OneIrreducibleSolvableGroup, 511
OneLibraryGroup, 500
OneMutable, 281
OneOfPcgs, 422
OneOp, 281
OnePrimitiveGroup, 500
OneSameMutability, 281
OneSM, 281
OneSmallGroup, 502
OneTransitiveGroup, 500
OnIndeterminates, 650
as a permutation action, 382
OnIndeterminates, 381
OnLines, 382
example, 495
OnPairs, 381
OnPoints, 381
OnRight, 381
OnSets, 381
OnSetsDisjointSets, 381
OnSetsSets, 381
OnSetsTuples, 381
OnSubspacesByCanonicalBasis, 383
Index

OnTuples, 381
OnTuplesSets, 381
OnTuplesTuples, 382
Operation, 394
OperationAlgebraHomomorphism, 605
Operational Structure of Domains, 275
OperationHomomorphism, 394
operations, for booleans, 166
Operations and Attributes for Vector Spaces, 574
Operations applicable to All Streams, 97
Operations Concerning Blocks, 706
Operations for (Near-)Additive Magmas, 541
Operations for Abelian Number Fields, 570
operations for algebraic elements, 660
Operations for Associative Words, 322
Operations for Associative Words by their Syllables, 324
Operations for Booleans, 166
Operations for Brauer Characters, 766
Operations for Class Functions, 744
Operations for Collections, 262
Operations for Cyclotomics, 153
Operations for Domains, 280
Operations for Finite Field Elements, 566
Operations for Finitely Presented Groups, 449
Operations for Group Homomorphisms, 370
Operations for Input Streams, 97
Operations for Lists, 192
Operations for Output Streams, 100
Operations for Pc Groups, 441
Operations for Rational Functions, 644
Operations for Special Kinds of Bases, 579
Operations for Stabilizer Chains, 408
Operations for Vector Space Bases, 577
Operations for Words, 316
Operations on elements of the algebra, 333
Operations on rewriting systems, 332
Operations to Evaluate Strings, 247
Operations to Produce or Manipulate Strings, 245
Operations which have Special Methods for Groups with Pcs, 434
operators, 42
arithmetic, 48
associativity, 49
for cyclotomics, 157
for lists, 178
precedence, 48
Operators for Character Tables, 698
Operators for Matrices, 216
Operators for Row Vectors, 208
Optimization and Compiler Options, 805
options, 28, 797
command line, filenames, 30
command line, internal, 31
options, under UNIX, 28
or, 166
Orbit, 383
OrbitFusions, 777
OrbitLength, 384
OrbitLengths, 384
OrbitLengthsDomain, 384
OrbitPerms, 400
OrbitPowerMaps, 772
Orbits, operation/attribute, 383
Orbits, 383
OrbitsDomain, 383
OrbitsPerms, 400
OrbitStabChain, 409
OrbitStabilizer, 384
OrbitStabilizerAlgorithm, 385
Orbit Stabilizer Methods for Polycyclic Groups, 433
Order, 284
of a class function, 741
order, of a group, 335
of a list, collection or domain, 262
of the prime residue group, 134
OrderedPartitions, 149
ordering, booleans, 165
of records, 254
OrderingLessThanFunctionNC, 269
OrderingLessThanOrEqualFunctionNC, 269
OrderingOfRewritingSystem, 332
OrderingOnGenerators, 271
OrderingsFamily, 269
Orderings on families of associative words, 270
OrderMod, 135
OrderOfRewritingSystem, 332
OrdersClassRepresentatives, 702
OrdersTom, 676
Ordinal, 248
ordinary character, 744
OrdinaryCharacterTable, 700
OrthogonalComponents, 757
Orthogonal Embeddings, 238
OrthogonalEmbeddings, 238
OrthogonalEmbeddingsSpecialDimension, 753
OSF, 800
Other Filters, 123
Other Operations Applicable to any Object, 114
Other Operations for Character Tables, 709
Other Operations for Tables of Marks, 679
Output, suppressing, 64
OutputLogTo, 94
for streams, 101
stop logging output, 94
OutputLogFile, 102
OutputTextNone, 105
OutputTextString, 103
OutputTextUser, 103

P
p-group, 351
package, 819
PadicCoefficients, 236
PadicExtensionNumberFamily, 663
PadicNumber, 663
for pure padics, 662
Pager, 26
Parametrized, 779
Parametrized Maps, 777
parametrized maps, 769
Parent, 279
ParentPcgs, 425
Parents, 279
partial order, 304
PartialOrderByOrderingFunction, 305
PartialOrderOfHasseDiagram, 304
Partitions, 149
partitions, improper, of an integer, 149
ordered, of an integer, 149
restricted, of an integer, 150
PartitionsGreatestEQ, 150
PartitionsGreatestLE, 150
PartitionsSet, 148
PartitionTuples, 151
PcElementByExponents, 423
PcElementByExponentsNC, 423
PCentralLieAlgebra, 624
PCentralSeries, 354
PcGroupCode, 444
PcGroupCodeRec, 444
PcGroupFpGroup, 438
Pc groups versus fp groups, 437
PcGroupWithPcgs, 440
Pcgs, 421
PcgsOrbitStabilizer, 433
Pcgs and Normal Series, 428
PcsByPcSequence, 421
PcsByPcSequenceNC, 421
PcsCentralSeries, 429
PcsChiefSeries, 429
PcsElementaryAbelianSeries, 429
PcsPCentralSeriesPGroup, 429
PClassPGroup, 351
PCore, 345
PcSeries, 422
PerfectGroup, 504
perfect groups, 504
PerfectIdentification, 505
PerfectResiduum, 347
Permanent, 152
Permanent of a Matrix, 152
PermBounds, 766
PermCharInfo, 761
PermCharInfoRelative, 762
PermChars, 762
PermCharsTom, 688
PermComb, 766
PermLeftQuoTransformation, 538
PermList, 398
PermListList, 194
Permutation, 388
PermutationCharacter, 743
permutation character, 788
permutation characters, possible, 760, 762
PermutationCycle, 388
PermutationMat, 220
PermutationsFamily, 396
Permutations Induced by Elements and Cycles, 388
PermutationsList, 147
PermutationTom, 674
Permutated, 196
as a permutation action, 382
for class functions, 741
PGL, 497
PGU, 497
Phi, 134
point stabilizer, 384
Polycyclic Generating Systems, 420
PolynomialByExtRep, 657
PolynomialCoefficientsOfPolynomial, 648
Polynomial Factorization, 650
Index

Polynomial Reduction, 654
PolynomialReduction, function, 654
PolynomialRing, 652
Polynomial Rings, 652
Polynomials as Univariate Polynomials in one Indeterminate, 648
Polynomials over the Rationals, 651
PopOptions, 87
Portability, 89
Porting GAP, 814
Position, 185
PositionBound, 187
PositionCanonical, 185
PositionNonZero, 187
PositionNot, 187
PositionNthOccurrence, 186
PositionProperty, 187
PositionSet, 186
PositionSorted, 186
PositionStream, 99
PositionSublist, 188
PositionWord, 323
PositiveIntegers, 125
positive number, 48
PositiveRoots, 620
PositiveRootVectors, 620
PossibleClassFusions, 776
PossibleFusionsCharTableTom, 688
Possible Permutation Characters, 760
possible permutation characters, 760, 762
PossiblePowerMaps, 770
power, 48
   matrix, 217
   meaning for class functions, 740
   of words, 322
PowerMap, 769
PowerMapByComposition, 772
PowerMapOp, 769
Power Maps, 769
PowerMapsAllowedBySymmetrizations, 787
PowerMod, 553
PowerModCoeffs, 213
PowerModInt, 129
PowerPartition, 151
powerset, 145
PowerSubalgebraSeries, 598
PQuotient, 460
precedence, 48
precedence test, for permutations, 396
PrefrattiniSubgroup, 346
   for groups with pgs, 434
PreImage, 297
PreImageElm, 296
PreImages, 297
PreImagesElm, 296
Preimages in the Free Group, 448
Preimages in the Free Semigroup, 532
PreimagesOfTransformation, 537
PreImagesRange, 296
PreImagesRepresentative, 297
PreImagesSet, 297
Preimages under Homomorphisms from an FpGroup, 459
Preimages under Mappings, 296
preorder, 304
PresentationFpGroup, 464
PresentationNormalClosure, 469
PresentationNormalClosureRrs, 469
PresentationSubgroup, 466
PresentationSubgroupMtc, 468
PresentationSubgroupRrs, 467
PresentationViaCosetTable, 465
previous result, 64
PrevPrimeInt, 131
PrimaryGeneratorWords, 468
primary subgroup generators, 482
PrimeBlocks, 706
PrimeBlocksOp, 706
PrimeField, 561
Prime Integers and Factorization, 129
PrimePGroup, 351
PrimePowersInt, 132
prime residue group, 134
   exponent, 134
   generator, 136
   order, 134
Prime Residues, 134
PrimeResidues, function, 134
Primes, 129
primitive, 390
PrimitiveElement, 561
PrimitiveGroup, 509
Primitive Groups, 402
PrimitiveGroupsIterator, 509
PrimitiveIdentification, 509
PrimitiveIndexIrreducibleSolvableGroup, 511
Primitive Permutation Groups, 508
PrimitivePolynomial, 651
PrimitiveRoot, 568
PrimitiveRootMod, 136
primitive root modulo an integer, 136
Primitive Roots and Discrete Logarithms, 135
Primitivity of Characters, 791
Print, 65
PrintAmbiguity, 782
PrintArray, 221
PrintCharacterTable, 714
PrintFactorsInt, 131
PrintFormattingStatus, 101
Printing Character Tables, 712
Printing Class Functions, 741
Printing Presentations, 470
Printing Tables of Marks, 672
PrintObj, 66
for character tables, 712, 741
for tables of marks, 673
PrintTo, 94
for streams, 100
problems, 817
Problems on Particular Systems, 805
procedure call, 50
Procedure Calls, 50
procedure call with arguments, 50
Process, 107
Process, 107
Product, 198
product, of words, 322
rational functions, 644
ProductCoeffs, 213
ProductSpace, 598
ProductX, 201
ProfileFunctions, 82
ProfileGlobalFunctions, 82
ProfileMethods, 82
ProfileOperations, 82
ProfileOperationsAndMethods, 82
PROFILETHRESHOLD, 82
Profiling, 81
ProjectedInducedPcgs, 428
ProjectedPcElement, 428
Projection, 294
example for direct products, 487
example for semidirect products, 489
example for subdirect products, 489
example for wreath products, 490
for group products, 491
ProjectionMap, 779
projections, find all, 377
ProjectiveActionHomomorphismMatrixGroup, 415
ProjectiveActionOnFullSpace, 415
ProjectiveGeneralLinearGroup, 497
ProjectiveGeneralUnitaryGroup, 497
ProjectiveOrder, 230
ProjectiveSpecialLinearGroup, 497
ProjectiveSpecialUnitaryGroup, 497
ProjectiveSymplecticGroup, 497
prompt, 64
partial, 64
Properties, 123
Properties and Attributes for Lists, 188
Properties and Attributes of (General) Mappings, 294
Properties and Attributes of Binary Relations, 303
Properties and Attributes of Matrices, 218
Properties and Attributes of Rational Functions, 645
Properties and basic functionality, 269
Properties of a Lie Algebra, 618
Properties of rewriting systems, 333
Properties of Rings, 548
Properties of Tables of Marks, 678
PRump, 347
PseudoRandom, 265
PSL, 497
PSP, 497
PSp, 497
PSU, 497
PthPowerImage, 624
PthPowerImages, 623
Pure p-adic Numbers, 662
PurePadicNumberFamily, 662
PushOptions, 87
Q
Quadratic, 160
quadratic residue, 136, 137
QuaternionAlgebra, 593
QUIT, emergency quit, 73
quit, in emergency, 72
quit, 67
QUITTING, 73
QuoInt, 127
Index

Quotient, 544
quotient, for finitely presented groups, 447
matrices, 217
matrix and matrix list, 218
matrix and scalar, 217
of free monoid, 533
of free semigroup, 531
of words, 322
rational functions, 644
scalar and matrix, 217
scalar and matrix list, 217
vector and matrix, 217
QuotientFromSCTable, 592
Quotient Methods, 459
QuotientMod, 553
QuotientPolynomialsExtRep, 658
QuotientRemainder, 551
Quotients, 523
Quotients and Remainders, 127
QuotientSemigroupCongruence, 523
QuotientSemigroupHomomorphism, 523
QuotientSemigroupPreimage, 523
QuotRemLaurpols, 652

R
rN, 158
RadicalGroup, 347
RadicalOfAlgebra, 601
Random, 264
for integers, 127
for rationals, 142
RandomBinaryRelationOnPoints, 305
random element, of a list or collection, 264
Random Elements, 264
RandomInvertibleMat, 221
RandomIsomorphismTest, 445
Random Isomorphism Testing, 445
Randomized Methods for Permutation Groups, 404
RandomList, 266
RandomMat, 221
Random Matrices, 221
random seed, 266
RandomTransformation, 536
RandomUnimodularMat, 221
Range, 295
range, 201
Ranges, 201
RankAction, 389
RankFilter, 116
RankMat, 222
RankOfTransformation, 537
RankPGroup, 351
Rat, 142
for strings, 247
RationalClass, 344
RationalClasses, 344
RationalFunctionByExtRep, 657
RationalFunctionByExtRepWithCancellation, 658
Rational Function Families, 655
RationalFunctionsFamily, 655
RationalizedMat, 161
Rationals, 141
RClassOfHClass, 524
Read, 93
for streams, 97
ReadAll, 99
ReadAllLine, 104
ReadAsFunction, 93
for streams, 98
ReadByte, 98
read eval print loop, 64
ReadLine, 98
ReadPkg, 823
ReadTest, 84
for streams, 98
RealClasses, 704
RealizableBrauerCharacters, 767
RecNames, 251
Recognizing Characters, 244
record, component access, 251
compartment assignment, 252
compartment variable, 252
compartment variable assignment, 253
Record Access Operations, 255
Record Assignment, 252
record assignment, operation, 256
record boundness test, operation, 256
record component, operation, 256
record unbind, operation, 256
Recovery from NoMethodFound-Errors, 77
recursion, 55
redisplay a help section, 24
redisplay with next help viewer, 24
ReduceCoeffs, 213
ReduceCoeffsMod, 213
ReducedAdditiveInverse, 333
ReducedCharacters, 750
ReducedClassFunctions, 750
ReducedComm, 333
ReducedConfluentRewritingSystem, 533
ReducedConjugate, 333
ReducedDifference, 333
ReducedForm, 332
ReducedInverse, 333
ReducedLeftQuotient, 333
ReducedOne, 333
ReducedPcElement, 423
ReducedPower, 333
ReducedProduct, 333
ReducedQuotient, 333
ReducedScalarProduct, 333
ReducedSum, 333
ReducedZero, 333
ReduceRules, 333
ReduceStabChain, 410
Reducing Virtual Characters, 750
Ree, 494
ReeGroup, 494
ReesCongruenceOfSemigroupIdeal, 522
ReesMatrixSemigroup, 525
ReesMatrixSemigroupElement, 525
Rees Matrix Semigroups, 525
ReesZeroMatrixSemigroup, 525
ReesZeroMatrixSemigroupElement, 525
ReesZeroMatrixSemigroupElementIsZero, 526
RefinedPcGroup, 440
ReflectionMat, 221
ReflexiveClosureBinaryRelation, 305
reflexive relation, 303
regular, 390
regular action, 386
RegularActionHomomorphism, 387
relations, 292
Relations Between Domains, 285
RelationsOfFpSemigroup, 532
RelativeBasis, 577
RelativeBasisNC, 577
relatively prime, 48
RelativeOrderOfPcElement, 422
RelativeOrders, of a pcgs, 422
Relators in a Presentation, 469
RelatorsOfFpGroup, 448
remainder, operation, 285
remainder of a quotient, 128
RemInt, 128
remove, an element from a set, 192
RemoveFile, 94
RemoveOuterCoeffs, 212
RemoveRelator, 472
RemoveSet, 192
RemoveStabChain, 410
Repeat, 52
repeat loop, 52
ReplacedString, 246
Representation, 119
representation, as a sum of two squares, 139
Representations for Associative Words, 325
Representations for Group Homomorphisms, 378
Representations of Algebras, 606
RepresentationsOfObject, 119
Representative, 262
representative, of a list or collection, 262
RepresentativeAction, 385
RepresentativeLinearOperation, 606
RepresentativeOperation, 394
RepresentativesContainedRightCosets, 342
RepresentativesFusions, 777
RepresentativeSmallest, 202
RepresentativesMinimalBlocks, 391
RepresentativesPerfectSubgroups, 359
RepresentativesPowerMaps, 772
RepresentativesSimpleSubgroups, 359
RepresentativeTom, 687
RepresentativeTomByGenerators, 687
RepresentativeTomByGeneratorsNC, 687
RequirePackage, 821
Reread, 95
REREADING, 95
ResetOptionsStack, 87
residue, quadratic, 136, 137
Residue Class Rings, 132
RespectsAddition, 299
RespectsAdditiveInverses, 299
RespectsInverses, 299
RespectsMultiplication, 298
RespectsOne, 299
RespectsScalarMultiplication, 300
RespectsZero, 300
RestoreStateRandom, 265
Restricted and Induced Class Functions, 748
RestrictedClassFunction, 748
Semidirect Products, 487
SemiEchelonBasis, 583
SemiEchelonBasisNC, 583
SemiEchelonMat, 224
SemiEchelonMatDestructive, 224
SemiEchelonMat, 225
SemiEchelonMat3Destructive, 225
SemiEchelonMatTransformation, 224
Semigroup, 520
semigroup, 520
SemigroupByGenerators, 520
SemigroupByMultiplicationTable, 521
SemigroupIdealByGenerators, 522
SemigroupOfRewritingSystem, 534
semiregular, 389
Semisimple Lie Algebras and Root Systems, 619
SemiSimpleType, 619
sequence, Bernoulli, 144
Fibonacci, 151
Lucas, 152
Series of Ideals, 617
Set, 259
SetAssertionLevel, 81
SetCommutator, 439
SetConjugate, 439
SetCrystGroupDefaultAction, 419
set difference, of collections, 264
SetEntrySCTable, 591
SetHelpViewer, 25
SetIndeterminateName, 643
SetInfoLevel, 80
SetName, 114
Set Operations via Boolean Lists, 205
SetParent, 279
SetPower, 439
SetPrintFormattingStatus, 101
SetRecursionTrapInterval, 85
Sets, 109
sets, 168, 190
Sets of Subgroups, 356
set stabilizer, 384
Setter, 121
setter, 120
Setter and Tester for Attributes, 120
SetX, 200
ShallowCopy, 113
for lists, 175
ShiftedCoeffs, 213
ShiftedPadicNumber, 662
Shifting and Trimming Coefficient Lists, 211
ShortestVectors, 239
ShortLexOrdering, 272
short vectors spanning a lattice, 236, 751
ShowArgument, 77
ShowArguments, 77
ShowDetails, 77
ShowImpliedFilters, 117
ShowMethods, 77
ShowOtherMethods, 78
ShrinkCoeffs, 214
ShrinkRowVector, 212
SiftedPcElement, 423
SiftedPermutation, 409
SiftedVector, 584
Sigma, 138
sign, of an integer, 126
Sign and Cycle Structure, 398
SignInt, 126
SignPartition, 150
SignPerm, 398
SimpleLieAlgebra, 615
SimpleSystem, 620
SimplifiedFpGroup, 466
SimplifiedFpGroup, 466
SimplifyPresentation, 473
SimsNo, 510
SimultaneousEigenvalues, 230
SingleCollector, 439
singlequote character, 242
singlequotes, 240
Size, 262
for character tables, 701
for groups with pcgs, 434
size, of a list or collection, 262
SizeBlist, 205
SizeConsiderFunction, 362
SizeNumbersPerfectGroups, 505
SizeOfFieldOfDefinition, 767
SizesCentralizers, 702
SizesConjugacyClasses, 702
SizeScreen, 75
SizeScreen, 75
SizesPerfectGroups, 504
SizeStabChain, 409
SL, 495
smaller, associative words, 322
elements of finitely presented groups, 447
nonassociative words, 315
pcwords, 437
rational functions, 644
SmallDegreePermutationRepresentation, 401
smaller or equal, 47
smaller test, 47
SmallestGeneratorPerm, 397
SmallestMovedPoint, 397
SmallestRootInt, 127
SmallGeneratingSet, 363
SmallGroup, 502
Small Groups, 501
SmallGroupsInformation, 502
Smash MeatAxe Flags, 669
SmithNormalFormIntegerMat, 233
SmithNormalFormIntegerMatInverseTransforms, 233
SmithNormalFormIntegerMatTransforms, 233
SMTX.AbsoluteIrreducibilityTest, 668
SMTX.AlgEl, 669
SMTX.AlgElCharPol, 669
SMTX.AlgElCharPolFac, 669
SMTX.AlgElMat, 669
SMTX.AlgElNullspaceDimension, 669
SMTX.AlgElNullspaceVec, 669
SMTX.CentMat, 669
SMTX.CentMatMinPoly, 669
SMTX.CompleteBasis, 669
SMTX.Getter, 668
SMTX.GoodElementGModule, 668
SMTX.IrreducibilityTest, 668
SMTX.MatrixSum, 669
SMTX.MinimalSubGModule, 669
SMTX.MinimalSubModules, 668
SMTX.RandomIrreducibleSubGModule, 668
SMTX.Setter, 668
SMTX.SortHomGModule, 668
SMTX.Subbasis, 669
SNFChouCollins, 232
SNFLLLDriven, 232
SNFNormDriven, 232
SNFofREF, 233
Sqrt, 285
square root, of an integer, 127
SquareRoots, 312
SSortedList, 259
StabChain, 405
StabChainBaseStrongGenerators, 407

SolutionMatDestructive, 223
Some Remarks about Character Theory in GAP, 691
Some Special Algebras, 593
Sort, 189
SortedCharacters, 724
SortedCharacterTable, 725
Sorted Character Tables, 724
SortedList, 259
sorted list, 188
Sorted Lists and Sets, 190
sorted lists as collections, 258
SortedSparseActionHomomorphism, 387
SortedTom, 674
Sortex, 190
Sorting Lists, 189
SortingPerm, 190
Sorting Tables of Marks, 674
SortParallel, 189
Source, 295
SP, 496
Sp, 496
space, 40
SparseActionHomomorphism, 387
SparseCartanMatrix, 621
Special Characters, 242
special character sequences, 242
Special Filenames, 91
Special Generating Sets, 362
SpecialLinearGroup, 495
SpecialOrthogonalGroup, 496
SpecialPcgs, attribute, 431
Specific and Parametrized Subgroups, 345
Specific Methods for Subgroup Lattice Computations, 360
SplitCharacters, 719
SplitExtension, 442
SplitExtensions, 443
SplitString, 245
SplittingField, 647

Some elements of finitely presented groups, 447
nonassociative words, 315
pcwords, 437
rational functions, 644
SmallDegreePermutationRepresentation, 401
smaller or equal, 47
smaller test, 47
SmallestGeneratorPerm, 397
SmallestMovedPoint, 397
SmallestRootInt, 127
SmallGeneratingSet, 363
SmallGroup, 502
Small Groups, 501
SmallGroupsInformation, 502
Smash MeatAxe Flags, 669
SmithNormalFormIntegerMat, 233
SmithNormalFormIntegerMatInverseTransforms, 233
SmithNormalFormIntegerMatTransforms, 233
SMTX.AbsoluteIrreducibilityTest, 668
SMTX.AlgEl, 669
SMTX.AlgElCharPol, 669
SMTX.AlgElCharPolFac, 669
SMTX.AlgElMat, 669
SMTX.AlgElNullspaceDimension, 669
SMTX.AlgElNullspaceVec, 669
SMTX.CentMat, 669
SMTX.CentMatMinPoly, 669
SMTX.CompleteBasis, 669
SMTX.Getter, 668
SMTX.GoodElementGModule, 668
SMTX.IrreducibilityTest, 668
SMTX.MatrixSum, 669
SMTX.MinimalSubGModule, 669
SMTX.MinimalSubModules, 668
SMTX.RandomIrreducibleSubGModule, 668
SMTX.Setter, 668
SMTX.SortHomGModule, 668
SMTX.Subbasis, 669
SNFChouCollins, 232
SNFLLLDriven, 232
SNFNormDriven, 232
SNFofREF, 233
Sqrt, 285
square root, of an integer, 127
SquareRoots, 312
SSortedList, 259
StabChain, 405
StabChainBaseStrongGenerators, 407

SolutionMatDestructive, 223
Some Remarks about Character Theory in GAP, 691
Some Special Algebras, 593
Sort, 189
SortedCharacters, 724
SortedCharacterTable, 725
Sorted Character Tables, 724
SortedList, 259
sorted list, 188
Sorted Lists and Sets, 190
sorted lists as collections, 258
SortedSparseActionHomomorphism, 387
SortedTom, 674
Sortex, 190
Sorting Lists, 189
SortingPerm, 190
Sorting Tables of Marks, 674
SortParallel, 189
Source, 295
SP, 496
Sp, 496
space, 40
SparseActionHomomorphism, 387
SparseCartanMatrix, 621
Special Characters, 242
special character sequences, 242
Special Filenames, 91
Special Generating Sets, 362
SpecialLinearGroup, 495
SpecialOrthogonalGroup, 496
SpecialPcgs, attribute, 431
Specific and Parametrized Subgroups, 345
Specific Methods for Subgroup Lattice Computations, 360
SplitCharacters, 719
SplitExtension, 442
SplitExtensions, 443
SplitString, 245
SplittingField, 647
Sqrt, 285
square root, of an integer, 127
SquareRoots, 312
SSortedList, 259
StabChain, 405
StabChainBaseStrongGenerators, 407
Index

SubmagmaWithInversesNC, 309
SubmagmaWithOne, 309
SubmagmaWithOneNC, 309
Submodule, 556
SubmoduleNC, 556
Submodules, 556
Submonoid, 527
SubmonoidNC, 527
SubnearAdditiveGroup, 541
SubnearAdditiveGroupNC, 541
SubnearAdditiveMagma, 540
SubnearAdditiveMagmaNC, 540
SubnearAdditiveMagmaWithZero, 540
SubnearAdditiveMagmaWithZeroNC, 540
SubnormalSeries, 353
Subring, 544
SubringNC, 544
SubringWithOne, 548
SubringWithOneNC, 548
Subroutines for the Construction of Class Fusions, 787
Subroutines for the Construction of Power Maps, 785
Subsemigroup, 520
SubsemigroupNC, 520
subsets, 145
subset test, for collections, 263
Subspace, 573
SubspaceNC, 573
Subspaces, 574
SubstitutedWord, 324
SubTom, 675
Substruct, 280
SubstructNC, 280
SubSyllables, 325
subtract, a set from another, 192
SubtractBlist, 206
subtraction, 48
matrices, 216
matrix and scalar, 216
rational functions, 644
scalar and matrix, 216
scalar and matrix list, 217
scalar and vector, 209
vector and scalar, 209
vectors, 209
SubtractSet, 192
Subword, 323
Successors, 304
Suitability for Compilation, 36
Sum, 199
Sum and Intersection of Pcgs, 430
SumFactorizationFunctionPcgs, 430
SumIntersectionMat, 225
SumX, 200
SupersolvableResiduum, 347
superuser, 820
SupportedCharacterTableInfo, 694
SurjectiveActionHomomorphismAttr, 394
SuzukiGroup, 494
SylowComplement, 348
SylowSubgroup, 347
Sylow Subgroups and Hall Subgroups, 347
SylowSystem, 348
Symbols, 40
Symmetric and Alternating Groups, 401
SymmetricClosureBinaryRelation, 305
SymmetricGroup, 494
symmetric group, powermap, 151
SymmetricParentGroup, 402
SymmetricParts, 757
SymmetricPowerOfAlgebraModule, 634
symmetric relation, 303
Symmetrizations, 756
symmetrizations, orthogonal, 757
symplectic, 758
Symmetrizations of Class Functions, 756
SymplecticComponents, 758
SymplecticGroup, 496
syntax errors, 64
system getter, 120
system setter, 120
System V, 800
SysV, 800
Sz, 494
T

$t_N$, 158
TableAutomorphisms, 727
table automorphisms, 777, 788
table of chapters for help books, 24
TableOfMarks, 671
TableOfMarksByLattice, 672
TableOfMarksComponents, 675
TableOfMarksCyclic, 689
TableOfMarksDihedral, 689
TableOfMarksFamily, 675
TableOfMarksFrobenius, 689
Table of Marks Objects in GAP, 670
tabulator, 40
 Tau, 138
Technical Details about Tables of Marks, 675
Technical Details about the Implementation of Magma Rings, 640
Technical Matters Concerning General Mappings, 301
TemporaryGlobalVarName, 45
TensorProductOfAlgebraModules, 633
Tensor Products and Exterior and Symmetric Powers, 633
test, for a primitive root, 136
for a rational, 141
for records, 251
for set equality, 191
TestConsistencyMaps, 782
Tester, 120
tester, 120
Test Files, 84
TestHomogeneous, 791
TestInducedFromNormalSubgroup, 793
Testing Finiteness of Finitely Presented Groups, 462
Testing for the System Architecture, 814
Testing Monomiality, 793
TestJacobi, 591
TestMonomial, 793
TestMonomialQuick, 794
TestMonomialUseLattice, 794
TestPackageAvailability, 823
TestPerm1, 764
TestPerm2, 764
TestPerm3, 764
TestPerm4, 764
TestPerm5, 764
TestQuasiPrimitive, 792
TestRelativelySM, 795
Tests for Actions, 389
Tests for the Availability of Methods, 366
TestSubnormallyMonomial, 795
The .gaprc file, 34
The Adjoint Representation, 624
The Compiler, 35
The Defining Attributes of Rational Functions, 656
The Dixon-Schneider Algorithm, 717
The Documentation, 815
The External Representation for Associative Words, 326
The family pegs, 436
The Interface between Character Tables and Groups, 696
The Interface between Tables of Marks and Character Tables, 688
The Library of Tables of Marks, 690
then, 51
The Natural Action, 490
The Pager Command, 26
The Permutation Image of an Action, 386
The Representations of Rational Functions, 655
The Smash MeatAxe, 668
The Syntax in BNF, 58
ThreeGroup library, 501
Tietze Options, 485
Tietze Transformations, 472
Tietze Transformations that introduce new Generators, 476
TietzeWordAbstractWord, 470
time, 81
Tod-Coxeter Procedure, 535
Trace, 218
for field elements, 562
of a matrix, 218
TracedCosetFpGroup, 450
TraceImmediateMethods, 79
TraceMat, 218
TraceMethods, 79
TracePolynomial, 562
Tracing generator images through Tietze transformations, 480
Tracing Methods, 79
TransferDiagram, 781
Transformation, 536
TransformationData, 536
TransformationFamily, 536
TransformationNC, 536
TransformationRelation, 538
TransformationType, 536
TransformingPermutations, 727
TransformingPermutationsCharacterTables, 727
transitive, 389
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransitiveClosureBinaryRelation, 305</td>
</tr>
<tr>
<td>TransitiveGroup, 500</td>
</tr>
<tr>
<td>TransitiveIdentification, 500</td>
</tr>
<tr>
<td>Transitive Permutation Groups, 500</td>
</tr>
<tr>
<td>transitive relation, 303</td>
</tr>
<tr>
<td>Transitivity, for characters, 747</td>
</tr>
<tr>
<td>for class functions, 747</td>
</tr>
<tr>
<td>for group actions, 389</td>
</tr>
<tr>
<td>transporter, 385</td>
</tr>
<tr>
<td>TransposedMat, 220</td>
</tr>
<tr>
<td>TransposedMatAttr, 220</td>
</tr>
<tr>
<td>TransposedMatDestructive, 220</td>
</tr>
<tr>
<td>TransposedMatImmutable, 220</td>
</tr>
<tr>
<td>TransposedMatMutable, 220</td>
</tr>
<tr>
<td>TransposedMatOp, 220</td>
</tr>
<tr>
<td>TransposedMatrixGroup, 414</td>
</tr>
<tr>
<td>Transversals, 341</td>
</tr>
<tr>
<td>Triangular Matrices, 226</td>
</tr>
<tr>
<td>TriangulizedIntegerMat, 234</td>
</tr>
<tr>
<td>TriangulizedIntegerMatInverseTransform, 234</td>
</tr>
<tr>
<td>TriangulizedIntegerMatTransform, 234</td>
</tr>
<tr>
<td>TriangulizedNullspaceMat, 222</td>
</tr>
<tr>
<td>TriangulizedNullspaceMatDestructive, 222</td>
</tr>
<tr>
<td>TriangulizedIntegerMat, 233</td>
</tr>
<tr>
<td>TriangulizeMat, 222</td>
</tr>
<tr>
<td>TrivialCharacter, 743</td>
</tr>
<tr>
<td>TrivialGroup, 492</td>
</tr>
<tr>
<td>TrivialIterator, 267</td>
</tr>
<tr>
<td>TrivialSubalgebra, 595</td>
</tr>
<tr>
<td>TrivialSubgroup, 345</td>
</tr>
<tr>
<td>TrivialSubmagmaWithOne, 312</td>
</tr>
<tr>
<td>TrivialSubmodule, 556</td>
</tr>
<tr>
<td>TrivialSubmonoid, 527</td>
</tr>
<tr>
<td>TrivialSubnearAdditiveMagmaWithZero, 541</td>
</tr>
<tr>
<td>TrivialSubspace, 574</td>
</tr>
<tr>
<td>TryConwayPolynomialForFrobeniusCharacter-Value, 767</td>
</tr>
<tr>
<td>TryCosetTableInWholeGroup, 453</td>
</tr>
<tr>
<td>TryGcdCancelExtRepPolynomials, 658</td>
</tr>
<tr>
<td>Tuples, 147</td>
</tr>
<tr>
<td>tuple stabilizer, 384</td>
</tr>
<tr>
<td>TwoClosure, 411</td>
</tr>
<tr>
<td>TwoCoboundaries, 441</td>
</tr>
<tr>
<td>TwoCocycles, 441</td>
</tr>
<tr>
<td>TwoCohomology, 441</td>
</tr>
<tr>
<td>TwoGroup library, 501</td>
</tr>
<tr>
<td>TwoSidedIdeal, 545</td>
</tr>
<tr>
<td>TwoSidedIdealByGenerators, 546</td>
</tr>
<tr>
<td>TwoSidedIdealNC, 546</td>
</tr>
<tr>
<td>TwoSquares, 139</td>
</tr>
<tr>
<td>type, boolean, 165</td>
</tr>
<tr>
<td>cyclotomic, 153</td>
</tr>
<tr>
<td>records, 251</td>
</tr>
<tr>
<td>strings, 240</td>
</tr>
<tr>
<td>TypeObj, 124</td>
</tr>
<tr>
<td>TypeOfDefaultGeneralMapping, 302</td>
</tr>
<tr>
<td>Types, 124</td>
</tr>
<tr>
<td>TzEliminate, 475</td>
</tr>
<tr>
<td>TzFindCyclicJoins, 476</td>
</tr>
<tr>
<td>TzGo, 472</td>
</tr>
<tr>
<td>TzGoGo, 474</td>
</tr>
<tr>
<td>TzImagesOldGens, 480</td>
</tr>
<tr>
<td>TzInitGeneratorImages, 480</td>
</tr>
<tr>
<td>TzNewGenerator, 472</td>
</tr>
<tr>
<td>TzOptions, 485</td>
</tr>
<tr>
<td>TzPreImagesNewGens, 481</td>
</tr>
<tr>
<td>TzPrint, 471</td>
</tr>
<tr>
<td>TzPrintGeneratorImages, 481</td>
</tr>
<tr>
<td>TzPrintGenerators, 470</td>
</tr>
<tr>
<td>TzPrintLengths, 471</td>
</tr>
<tr>
<td>TzPrintOptions, 486</td>
</tr>
<tr>
<td>TzPrintPairs, 471</td>
</tr>
<tr>
<td>TzPrintPresentation, 471</td>
</tr>
<tr>
<td>TzPrintRelators, 470</td>
</tr>
<tr>
<td>TzPrintStatus, 471</td>
</tr>
<tr>
<td>TzSearch, 475</td>
</tr>
<tr>
<td>TzSearchEqual, 476</td>
</tr>
<tr>
<td>TzSort, 464</td>
</tr>
<tr>
<td>TzSubstitute, 477</td>
</tr>
<tr>
<td>TzSubstituteCyclicJoins, 480</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>(a_N), 158</td>
</tr>
<tr>
<td>UglyVector, 586</td>
</tr>
<tr>
<td>ULTRIX, 800</td>
</tr>
<tr>
<td>Unbind, 43</td>
</tr>
<tr>
<td>for lists, 174</td>
</tr>
<tr>
<td>UnbindGlobal, 45</td>
</tr>
<tr>
<td>UnderlyingCharacteristic, 702</td>
</tr>
<tr>
<td>UnderlyingCharacterTable, 737</td>
</tr>
<tr>
<td>UnderlyingElement, fp group elements, 448</td>
</tr>
<tr>
<td>fp semigroup elements, 532</td>
</tr>
<tr>
<td>UnderlyingElementOfReesMatrixSemigroup-Element, 526</td>
</tr>
<tr>
<td>UnderlyingElementOfReesZeroMatrixSemigroup-Element, 526</td>
</tr>
</tbody>
</table>

U
Index

UnderlyingExternalSet, 394
UnderlyingFamily, 614
UnderlyingGeneralMapping, 295
UnderlyingGroup, for character tables, 696
  for tables of marks, 677
UnderlyingLeftModule, 577
UnderlyingLieAlgebra, 620
UnderlyingMagma, 637
UnderlyingRelation, 295
UnInstallCharReadHookFunc, 105
Union, 263
union, of collections, 264
  of sets, 192
Union2, 263
UnionBlist, 205
Unique, 193
UniteBlist, 206
UniteBlistList, 206
UniteSet, 192
Units, 549
Units and Factorizations, 549
UnivariatenessTestRationalFunction, 648
UnivariatePolynomial, 647
UnivariatePolynomialByCoefficients, 647
UnivariatePolynomialRing, 654
Univariate Polynomial Rings, 654
Univariate Polynomials, 647
UnivariateRationalFunctionByCoefficients, 652
Univariate Rational Functions, 652
UniversalEnvelopingAlgebra, 626
Universal Enveloping Algebras, 626
UNIX, features, 28
  installation, 800
  options, 28
UNIXSelect, 97
Unknown, 163
UnloadSmallGroupsData, 502
UnorderedTuples, 146
UnprofileFunctions, 82
UnprofileMethods, 82
until, 52
UntraceMethods, 79
unzoo, 797
UpdateMap, 779
UpEnv, 71
UpperCentralSeriesOfGroup, 354
UpperSubdiagonal, 226
UseBasis, 557
UseFactorRelation, 286
Useful Categories for all Elements of a Family, 289
Useful Categories of Elements, 287
UseIsomorphismRelation, 286
User Streams, 103
UseSubsetRelation, 285
utilities for editing GAP files, 75

V
  evN, 158
Valuation, 662
Value, 649
ValueCochain, 629
ValueGlobal, 44
ValueMolienSeries, 760
ValuePol, 213
ValuesOfClassFunction, 737
Variable Access in a Break Loop, 71
Variables, 42
Vectors as coefficients of polynomials, 212
VectorSpace, 573
VectorSpaceByPcgsOfElementaryAbelianGroup, 432
Vector Space Homomorphisms, 584
Vector Spaces Handled By Nice Bases, 586
vi, 75
View, 65
View and Print, 65
ViewObj, 66
Vim, 75
VirtualCharacter, 742
virtual character, 744
virtual characters, 735

W
  wx, 158
WeekDay, 249
WeightLexOrdering, 272
WeightOfGenerators, 273
WeightsTom, 678
WeightVecFFE, 212
WeylGroup, 621
WeylOrbitIterator, 623
Where, 70
While, 51
while loop, 51
Whitespaces, 40
Why Class Functions?, 735
Windows, 806
    installation, 807
WordAlp, 245
    words, in generators, 339
Working with large degree permutation groups, 412
WreathProduct, 490
    wreath product embedding, 491
WreathProductImprimitiveAction, 490
WreathProductOrdering, 274
WreathProductProductAction, 491
Wreath Products, 490
WriteAll, 100
WriteByte, 100
WriteLine, 100
www, 798

X
    $x_N$, 158

Y
    $y_N$, 158

Z
    Z, 565
    ZClassRepsQClass, 418
    Zero, 282
    ZeroAttr, 282
    ZeroCoefficient, 638
    ZeroCoefficientRatFun, 657
    ZeroImmutable, 282
    ZeroMapping, 293
    ZeroMutable, 282
    ZeroOp, 282
    ZeroSameMutability, 282
    ZeroSM, 282
    ZippedProduct, 658
    ZippedSum, 658
    ZmodnZ, 132
    ZmodnZObj, 132
    ZmodpZ, 132
    ZmodpZNC, 132
    ZumbroichBase, 570
    Zuppos, 360