

Handbook of Magma Functions

Part X: Finitely Presented Groups

Version 2.30-1

Sydney

November 10, 2026

Editors

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *Bruce Cox* *Brendan Creutz* *Steve Donnelly* *Tim Dokchitser* *Willem de Graaf* *Andreas-Stephan Elsenhans* *Claus Fieker* *Damien Fisher* *Volker Gebhardt* *Sergei Haller* *Michael Harrison* *Florian Hess* *Derek Holt* *David Howden* *Al Kasprzyk* *Markus Kirschmer* *David Kohel* *Axel Kohnert* *Dimitri Leemans* *Paulette Lieby* *Graham Matthews* *Scott Murray* *Eamonn O'Brien* *Dan Roozemond* *Ben Smith* *Bernd Souvignier* *William Stein* *Allan Steel* *Damien Stehlé* *Nicole Sutherland* *Don Taylor* *Bill Unger* *Alexa van der Waall* *Paul van Wamelen* *Helena Verrill* *John Voight* *Mark Watkins* *Greg White*

Production Editors

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production

Claus Fieker *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

77 Abelian Groups **1**
77.1 Introduction . . . 1
77.2 Construction of a Finitely Presented Abelian Group and its Elements . . . 1
77.3 Construction of a Generic Abelian Group . . . 3
77.4 Elements . . . 7
77.5 Construction of Subgroups and Quotient Groups . . . 9
77.6 Standard Constructions and Conversions . . . 11
77.7 Operations on Elements . . . 12
77.8 Invariants of an Abelian Group . . . 15
77.9 Canonical Decomposition . . . 15
77.10 Set-Theoretic Operations . . . 15
77.11 Coset Spaces . . . 17
77.12 Subgroup Constructions . . . 17
77.13 Subgroup Chains . . . 18
77.14 General Group Properties . . . 19
77.15 Representation Theory . . . 21
77.16 The Hom Functor . . . 21
77.17 Automorphism Groups . . . 22
77.18 Cohomology . . . 22
77.19 Homomorphisms . . . 23
77.20 Bibliography . . . 25

78 Free Groups **26**
78.1 Introduction . . . 26
78.2 Free Groups and Words . . . 26
78.3 Finitely Generated Subgroups of Free Groups . . . 31
78.4 The Automorphism Group of a Free Group . . . 33
78.5 Bibliography . . . 35

79 Introduction to Fp-Groups **36**
79.1 Introduction . . . 36
79.2 Construction of FP-Groups . . . 37
79.3 Properties of an FP-group . . . 45
79.4 Subgroups . . . 47
79.5 Finite FP-Groups . . . 58
79.6 Homomorphisms of FP-Groups . . . 59
79.7 Quotient Groups . . . 64
79.8 Bibliography . . . 75

80 Finitely Presented Groups **76**
80.1 Introduction . . . 76
80.2 Construction of an FP-Group . . . 77
80.3 Simplification . . . 83
80.4 Standard Constructions . . . 89
80.5 Properties of an FP-Group . . . 95
80.6 Subgroups . . . 99
80.7 Coset Spaces and Tables . . . 117

80.8 Subgroups of Finite Index . . . 130
80.9 Finite FP-Groups . . . 144
80.10 Homomorphisms . . . 146
80.11 Quotient Group Methods . . . 155
80.12 KG-Modules . . . 194
80.13 Some Developed Examples . . . 197
80.14 Bibliography . . . 201

81 Polycyclic Groups 203
81.1 Introduction . . . 203
81.2 Polycyclic Groups and Polycyclic Presentations . . . 203
81.3 Subgroups, Quotient Groups, Homomorphisms, and Extensions . . . 209
81.4 Conversion between Categories . . . 213
81.5 Access Functions for Groups . . . 215
81.6 Set-Theoretic Operations in a Group . . . 215
81.7 Coset Spaces . . . 217
81.8 The Subgroup Structure . . . 219
81.9 General Group Properties . . . 220
81.10 Normal Structure and Characteristic Subgroups . . . 222
81.11 Conjugacy . . . 225
81.12 Representation Theory . . . 226
81.13 Power Groups . . . 231
81.14 Bibliography . . . 231

82 Braid Groups 233
82.1 Introduction . . . 233
82.2 Constructing and Accessing Braid Groups . . . 238
82.3 Creating Elements of a Braid Group . . . 239
82.4 Working with Elements of a Braid Group . . . 244
82.5 Homomorphisms . . . 265
82.6 Bibliography . . . 270

83 Groups Defined by Rewrite Systems 272
83.1 Introduction . . . 272
83.2 Constructing Confluent Presentations . . . 273
83.3 Properties of a Rewrite Group . . . 278
83.4 Arithmetic with Words . . . 279
83.5 Operations on the Set of Group Elements . . . 281
83.6 Homomorphisms . . . 282
83.7 Conversion to a Finitely Presented Group . . . 283
83.8 Bibliography . . . 284

84 Automatic and Hyperbolic Groups 285
84.1 Introduction . . . 285
84.2 Automatic Groups . . . 285
84.3 Hyperbolic Groups . . . 298
84.4 Bibliography . . . 304

85 Groups of Straight-Line Programs 305
85.1 Introduction . . . 305
85.2 Construction of an SLP-Group and its Elements . . . 305
85.3 Arithmetic with Elements . . . 306
85.4 Addition of Extra Generators . . . 306
85.5 Creating Homomorphisms . . . 307
85.6 Operations on Elements . . . 308
85.7 Set-Theoretic Operations . . . 308
85.8 Bibliography . . . 310

86 Finitely Presented Semigroups 311

86.1 Introduction . . . 311
86.2 The Construction of Free Semigroups and their Elements . . . 311
86.3 Elementary Operators for Words . . . 312
86.4 Specification of a Presentation . . . 313
86.5 Subsemigroups, Ideals, and Quotients . . . 314
86.6 Extensions . . . 316
86.7 Elementary Tietze Transformations . . . 316
86.8 String Operations on Words . . . 316

87 Monoids Given by Rewrite Systems **318**
87.1 Introduction . . . 318
87.2 Construction of a Rewrite Monoid . . . 319
87.3 Basic Operations . . . 323
87.4 Homomorphisms . . . 326
87.5 Set Operations . . . 327
87.6 Conversion to a Finitely Presented Monoid . . . 328
87.7 Bibliography . . . 329

References **330**

Index **333**

Chapter 77

Abelian Groups

77.1 Introduction

This Chapter contains a description of the Magma machinery provided for computing with abstract abelian groups. Such a group may be described either in terms of defining relations (category `GrpAb`) or by defining the group operations on some finite set (category `GrpAbGen`). In the case of finitely presented groups, the abelian groups may be finite and infinite, the only restriction being that the group be finitely generated.

77.2 Construction of a Finitely Presented Abelian Group and its Elements

77.2.1 The Free Abelian Group

`FreeAbelianGroup(n): RngIntElt -> GrpAb`

Construct the free abelian group F on n generators, where n is a positive integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the *left hand side* of the assignment statement.

Example: Free Abelian Group (ex-c01eb3)

The statement

```
> F := FreeAbelianGroup(2);
```

creates the free abelian group on two generators. Here the generators may be referenced using the standard names, $F.1$ and $F.2$.

The statement

```
> F<x, y> := FreeAbelianGroup(2);
```

defines F to be the free abelian group on two generators and assigns the names x and y to the generators.

77.2.2 Relations

`w1 = w2: GrpAbElt, GrpAbElt -> Rel`

Given words w_1 and w_2 over the generators of an abelian group A, create the relation $w_1 = w_2$. Note that this relation is not automatically added to the existing set of defining relations R for S. It may be added to R, for example, through use of the `quo`-constructor (see below).

`r[1]: GrpAbRel, RngIntElt -> GrpAbElt`

`LHS(r): Rel -> GrpAbElt`

Given a relation r over the generators of A, return the left hand side of the relation r. The object returned is a word over the generators of A.

`r[2]: GrpAbRel, RngIntElt -> GrpAbElt`

`RHS(r): Rel -> GrpAbElt`

Given a relation r over the generators of A, return the right hand side of the relation r. The object returned is a word over the generators of A.

`Parent(r): RelElt -> GrpAb`

Group over which the relation r is taken.

Example: Relations in a Free Abelian Group (ex-0f1824)

We may define a group and a set of relations as follows:

```
> F<x, y> := FreeAbelianGroup(2);
> rels := { 2*x = 3*y, 4*x + 4*y = Id(F)} ;
```

To replace one side of a relation, the easiest way is to reassign the relation. So for example, to replace the relation $2x = 3y$ by $2x = 4y$:

```
> r := 2*x = 3*y;
> r := LHS(r) = 4*y;
```

77.2.3 Specification of a Presentation

An abelian group with non-trivial relations is constructed as a quotient of an existing abelian group, possibly a free abelian group.

`AbelianGroup< X | R >: List(Var), List(GrpAbRel) -> GrpAb, Hom(GrpAb)`

Given a list X of variables $x_1, \ldots, x_r$, and a list of relations R over these generators, let F be the free abelian group on the generators $x_1, \ldots, x_r$. Then construct (a) an abelian group A isomorphic to the quotient of F by the subgroup of F defined by R, and (b) the natural homomorphism $\phi : F \to A$.

Each term of the list R is either a *word*, a *relation*, a *relation list*, or a *subgroup* of F.

- A *relation* consists of a pair of words, separated by '='.
- A *word* w is interpreted as a *relator*, that is, it is equivalent to the relation $w = 0$. (See above).
- A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=': $w_1 = w_2 = \cdots = w_r$. This is interpreted as the set of relations $w_1 = w_r, \ldots, w_{r-1} = w_r$. Note that the relation list construct is only meaningful in the context of the `quo`-constructor.

A *subgroup* H appearing in the list R contributes its generators to the relation set for A, i.e., each generator of H is interpreted as a relator for A. The group F may be referred to by the special symbol `$` in any word appearing to the right of the '`|`' symbol in the `quo`-constructor. Also, in the context of the `quo`-constructor, the identity element (empty word) may be represented by the digit 0.

The function returns:

(a) The quotient group A;

(b) The natural homomorphism $\phi : F \to A$.

Example: Abelian Group from a Presentation (ex-bd5820)

We create the abelian group defined by the presentation $< a, b, c \mid 7a + 4b + c, 8a + 5b + 2c, 9a + 6b + 3c >$.

```
> F<a, b, c> := FreeAbelianGroup(3);
> A := quo< F | 7*a + 4*b + c, 8*a + 5*b + 2*c, 9*a + 6*b + 3*c >;
> A;
AbelianGroup isomorphic to Z_3 + Z
Defined on 2 generators
Relations:
    3*A.1 = 0
```

A simple way of specifying an abelian group is as a product of cyclic groups.

AbelianGroup([n_1,...,n_r]): [RngIntElt] -> GrpAb

Construct the abelian group defined by the sequence $[n_1, \ldots, n_r]$ of non-negative integers as an abelian group. The function returns the direct product of cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$, where C_0 is interpreted as an infinite cyclic group.

Example: Abelian Group from Cyclic Factors (ex-ea4c01)

We create the abelian group $Z_2 \times Z_3 \times Z_4 \times Z_5 \times Z_6 \times Z \times Z$.

```
> G<[x]> := AbelianGroup([2,3,4,5,6,0,0]);
> G;
Abelian Group isomorphic to Z/2 + Z/6 + Z/60 + Z + Z
Defined on 7 generators
Relations:
    2*G.1 = 0
    3*G.2 = 0
    4*G.3 = 0
    5*G.4 = 0
    6*G.5 = 0
```

77.2.4 Accessing the Defining Generators and Relations

The functions described here provide access to basic information stored for an abelian group A.

A . i: GrpAb, RngIntElt -> GrpAbElt

The i-th defining generator for A.

Generators(A): GrpAb -> { GrpAbElt}

A set containing the generators for A.

NumberOfGenerators(A): GrpAb -> RngIntElt

Ngens(A): GrpAb -> RngIntElt

The number of generators for A.

Parent(u): GrpAbElt -> GrpAb

The parent group A of the word u.

Relations(A): GrpAb -> [Rel]

A sequence containing the defining relations for A.

RelationMatrix(A): GrpAb -> Mtrx

A matrix where each row corresponds to one of the defining relations of A.

77.3 Construction of a Generic Abelian Group

The term generic abelian group refers to the situation in which a group A is described by giving a set X together with functions acting on X that perform the fundamental group operations for A. Specifically, the user must provide functions which compute the product, the inverse, and the identity. For efficiency, the user may also optionally specify the order and a generating set for the group. This is made explicit below.

Going from such a definition of an abelian group A to a presentation for A will often be extremely expensive and so a small number of operations are implemented so as to not require this information. The two key operations are computing the order of an element and computing the discrete logarithm of an element to a given base. For many abelian group operations, knowledge of a presentation is required and if such an operation is invoked, the structure of A (and hence a presentation) will be automatically constructed.

There are two possible ways of computing the structure of the group. If the order of the group is known (or can be computed) then it is relatively easy to construct each of the p-Sylow subgroups. If the order is not available, the structure is computed from a set of generators supplied by the user.

77.3.1 Specification of a Generic Abelian Group

`GenericAbelianGroup(U: parameters): . -> GrpAbGen`

```
IdIntrinsic      : MonStg                    Default:
AddIntrinsic     : MonStg                    Default:
InverseIntrinsic : MonStg                    Default:
UseRepresentation: Bool                      Default: false
Order            : RngIntElt                 Default:
UserGenerators   : SetEnum                   Default:
ProperSubset     : Bool                      Default: false
RandomIntrinsic  : MonStg                    Default:
ComputeStructure : Bool                      Default: false
UseUserGenerators: Bool                      Default: false
PollardRhoRParam : RngInt                    Default: 20
PollardRhoTParam : RngInt                    Default: 8
PollardRhoVParam : RngInt                    Default: 3
```

Construct the generic abelian group A over the domain U. The domain U can be an aggregate (set, indexed set, or sequence) of elements or it can be any structure, for example, an elliptic curve, a jacobian, a magma of quadratic forms.

If the parameters `IdIntrinsic`, `AddIntrinsic`, and `InverseIntrinsic` are not set, then the identity, the composition, and the inverse function of A are taken to be the identity, the composition, and the inverse function of U or of `Universe(U)` if U is an aggregate. If the parameters `IdIntrinsic`, `AddIntrinsic`, and `InverseIntrinsic` are set, they define the identity, the composition, and the inverse function of A.

The parameter `IdIntrinsic` must be a function name whose sole argument is U or `Universe(U)` if U is an aggregate. `AddIntrinsic` must be a function name whose only two arguments are elements of U or of `Universe(U)` if U is an aggregate. `InverseIntrinsic` must be a function name whose only two arguments are elements of U or of `Universe(U)` if U is an aggregate. That is, it is required that `InverseIntrinsic` be a binary operation (see the example below where `InverseIntrinsic := "/"` is indeed binary). Defining any of the three above parameters implies that the other two must be defined as well.

Setting the parameter `UseRepresentation` to true implies that all elements of A will be internally represented as linear combinations of the generators of A obtained while computing the structure of A. This can be a costly procedure, since such a representation is akin to solving the discrete logarithm problem. The advantage of such a representation is that arithmetic for elements of A as well as computing the order of elements of A are then essentially trivial operations.

The parameter `Order` can be set to the order of the group, if known. Knowledge of the order may save a considerable amount of time for operations such as computing a Sylow subgroup, determining the group structure, or solve a discrete logarithm problem. More importantly, if A is a proper subset of U, or of `Universe(U)` if U is an aggregate, then one of `Order` or `UserGenerators` must be set.

One can set `UserGenerators` to some set of elements of U, or of `Universe(U)` if U is an aggregate, which generate the group A. This can be useful when A is a subset of U (`Universe(U)`), or more

generally when the computation of the group structure of A is made from a set of generators. Finally, setting `UserGenerators` may be an alternative to setting `RandomIntrinsic`.

The parameter `ProperSubset` must be set when A is a proper subset of U (`Universe(U)`).

The parameter `RandomIntrinsic` indicates the random function to use. If it is not set, the random function (if it is required) is taken to be the random function applying to U (`Universe(U)`).

The parameter `RandomIntrinsic` must be the name of a function taking as its sole argument U (`Universe(U)`) and returning an element of U (`Universe(U)`) which is also an element of the group A, which is important when A is a proper subset of U (`Universe(U)`. Therefore, if A is a proper subset of U (`Universe(U)`), then `RandomIntrinsic` must be set, unless the parameter `UserGenerators` is set.

The parameter `Structure` indicates that the group structure should be determined at the time of creation. If this parameter is set then the user may also want to set the following parameters:

Their meaning is detailed in Section *Computing Abelian Group Structure*.

Example: Generic Abelian Group of Units `(ex-5df08f)`

In our first example we create the subset U of $\mathbb{Z}/34384\mathbb{Z}$ corresponding to its unit group as a generic abelian group G. Note that U is an indexed set.

```
> m :=  34384;
> Zm := Integers(m);
> U := {@ x : x in Zm | GCD(x, m) eq 1 @};
> G := GenericAbelianGroup(U :
>        IdIntrinsic := "Id",
>        AddIntrinsic := "*",
>        InverseIntrinsic := "/");
> G;
Generic Abelian Group over
Residue class ring of integers modulo 34384
```

In our next example we construct unique representatives for the classes of binary quadratic forms corresponding to elements of a subgroup of the class group of the imaginary quadratic field of discriminant -4000004.

```
> n := 6;
> Dk := -4*(10^n + 1);
> Q := QuadraticForms(Dk);
>
> p := 2;
> S := { };
> while #S lt 10 do
>    if KroneckerSymbol(Dk,p) eq 1 then
>       I := Reduction(PrimeForm(Q,p));
>       Include(~S, I);
>    end if;
>    p := NextPrime(p);
> end while;
>
> QF := GenericAbelianGroup(Q : UserGenerators := S,
>                               ComputeStructure := true,
>                               UseUserGenerators := true);
> QF;
Generic Abelian Group over
Binary quadratic forms of discriminant -4000004
Abelian Group isomorphic to Z/2 + Z/516
Defined on 2 generators
```

(continues on next page)

(continued from previous page)

```
Relations:
  2*$.1 = 0
  516*$.2 = 0
```

So the structure of the subgroup is $Z_2 \times Z_{516}$

77.3.2 Accessing Generators

These functions give access to generating sets for a generic group A. If a generating set is requested and none has been supplied by the user then this will require the determination of the group structure which could be very expensive. Note the distinction between `Generators` and `UserGenerators`. From now on, unless otherwise specified, whenever mention is made of the generators of A, we refer to the generators as given by the `Generators` function.

`Universe(A): GrpAbGen`

The universe over which the generic abelian group A is defined.

`A . i: GrpAbGen, RngIntElt -> GrpAbGenElt`

The i-th generator for the generic abelian group A.

`Generators(A): GrpAbGen -> [ GrpAbGenElt ]`

A sequence containing a generating set for the generic abelian group A as elements of A. The set of generators returned for A is a reduced set of generators as constructed during the structure computation. Therefore, if the group structure of A has been computed from a user-supplied set of generators, it is unlikely that `Generators(A)` and `UserGenerators(A)` will be the same.

`UserGenerators(A): GrpAbGen -> [ GrpAbGenElt ]`

A sequence containing the user-supplied generators for the generic abelian group A as elements of A.

`NumberOfGenerators(A): GrpAbGen -> RngIntElt`

`Ngens(A): GrpAbGen -> RngIntElt`

The number of generators for the generic abelian group A.

77.3.3 Computing Abelian Group Structure

If the order of a generic abelian group A can be obtained then the structure of A is found by constructing each p-Sylow subgroup of A. The p-Sylow subgroups are built from random elements of the underlying set X of A. Recall that U (`Universe(U)`) is the domain of A. Random elements are obtained using either a random function attached to X or using a user-supplied function (the `RandomIntrinsic` parameter), or using user-supplied generators (the `UserGenerators` parameter). It is therefore vital that user-supplied generators truly generate the group one wishes to construct. A drawback of this method of obtaining the structure of A is the memory needed to store all the elements of a specific p-Sylow subgroup while under construction. This algorithm is mostly based on work by Michael Stoll.

The second approach computes the group structure from a set of generators as supplied by the user, removing the need to compute the order of A. This can be particularly useful when computing this order is expensive. Note that computing the structure of a group from a set of generators is akin to solving a number of the discrete logarithm problems. This second algorithm uses a variant of the Pollard–Rho algorithm and is due to Edlyn Teske [Tes98a].

If A is a subgroup of a generic abelian group, G say, then the structure of G is computed first. The rationale is that once the structure of G is known, computing the structure of A is almost always cheap.

`AbelianGroup(A: parameters): GrpAbGen -> GrpAb, Map`

```
UseUserGenerators: Bool                    Default: false
PollardRhoRParam : RngInt                  Default: 20
PollardRhoTParam : RngInt                  Default: 8
PollardRhoVParam : RngInt                  Default: 3
```

Compute the group structure of the generic abelian group A, which may be a subgroup as created by the subgroup constructor or the Sylow function. The two values returned are the abstract abelian group and the invertible map from the latter into A.

If `UseUserGenerators` is false, then the group structure computation is made via the construction of each p-Sylow subgroup, using the factorization of the order of A.

If `UseUserGenerators` is set to `true`, the group structure computation uses the user-supplied set of generators for A. In this case, the additional parameters `PollardRhoRParam`, `PollardRhoTParam`, and `PollardRhoVParam` can be supplied. Their values will be passed to the Pollard–Rho algorithm used in the group structure computation: `PollardRhoRParam` sets the size of the r-adding walks, `PollardRhoTParam` sets the size of the internal storage of elements, and `PollardRhoVParam` is used for an efficient finding of the periodic segment. It is conjectured that the default values as given above are 'best' (see [Tes98b]), therefore there should be no need to set these parameters in general.

Example: Structure of a Unit Group `(ex-0d76ee)`

The following statement computes the structure of the unit group of $\mathbb{Z}_{34384}$.

```
> G := AbelianGroup(G); G;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/6 + Z/612
Defined on 4 generators
Relations:
  2*G.1 = 0
  2*G.2 = 0
  6*G.3 = 0
  612*G.4 = 0
```

77.4 Elements

77.4.1 Construction of Elements

Unless otherwise stated, the operations in this section apply to fp-abelian groups and generic abelian groups.

`A ! [a1, ... ,an]: GrpAb, [RngIntElt] -> GrpAbElt`

`A ! [a1, ... ,an]: GrpAbGen, [RngIntElt] -> GrpAbGenElt`

Given an abelian group A with generators $e_1, \ldots, e_r$ and a sequence $Q = [a_1, \cdots, a_r]$ of integers, construct the element $a_1e_1 + \cdots + a_re_r$ of A.

`A ! e: GrpAbGen, Elt -> GrpAbGenElt`

Given a generic abelian group A and an element e of the domain over which it is defined, return e as an element of A. If A is a proper subset of its underlying domain, then e must be a linear combination of the generators (which may be user-supplied) of A.

`A ! g: GrpAbGen, GrpAbGenElt -> GrpAbGenElt`

Given a generic abelian group A and an element g of the underlying set X of A, return g as an element of A.

`A ! n: GrpAb, RngIntElt -> GrpAbElt`

Given an abelian group A with exactly one generator x, construct the element nx.

`Random(A): GrpAbGen -> GrpAbGenElt`

Given either a finite fp-abelian group or a generic abelian group A, return a random element of A.

`Identity(A): GrpAb -> GrpAbElt`

`Id(A): GrpAb -> GrpAbElt`

`A ! 0: GrpAb, RngIntElt -> GrpAbElt`

Construct the identity element (empty word) for the abelian group A.

Let A be a generic abelian group defined in the universe U of A. If g is an element of A, then $U!g$ is an element of U.

77.4.2 Representation of an Element

An element g of an abelian group A can be represented as a linear combination with respect to a given generating sequence. The coefficients appearing in this linear combination provide an alternative representation for g. If A is a fp-group, the generating set will be the one on which the group was defined. In the case of a generic group, the generating set can either be that obtained when constructing a presentation for A or a user-supplied generating set.

`Representation(g): GrpAbGenElt -> [RngIntElt]`

`ElementToSequence(g): GrpAbGenElt -> [RngIntElt]`

`Eltseq(g): GrpAbGenElt -> [RngIntElt]`

Let A be an abelian group with generating set $e_1, \ldots, e_n$ and suppose g is an element of A, where $g = a_1e_1 + \ldots + a_ne_n$. These functions return the sequence Q of n integers defined by $Q[i] = a_i$, for $i = 1, \ldots, n$. Moreover, each a_i, $i = 1, \ldots, n$, is the integer residue modulus the order of the ith generator.

`UserRepresentation(g): GrpAbGenElt -> [RngIntElt]`

Let A be a generic abelian group with a user-supplied set of generators $u_1, \ldots, u_n$ and suppose g is an element of A, where $g = a_1u_1 + \ldots + a_nu_n$. This function returns the sequence Q of n integers defined by $Q[i] = a_i$, for $i = 1, \ldots, n$. Moreover, each a_i, $i = 1, \ldots, n$, is the integer residue modulus the order of the ith generator.

`Representation(S, g): SeqEnum, GrpAbGenElt -> [RngIntElt], RngIntElt`

Let A be a generic abelian group and let $S = [s_1, \ldots, s_m]$ be any sequence of elements of A. Assume g is an element of A such that $bg = a_1s_1 + \ldots + a_ms_m$. This function returns as its first value the sequence Q of m integers defined by $Q[i] = a_i$, for $i = 1, \ldots, m$. The second value returned is the coefficient b of g. Note that b might not be 1.

Example: Creating and Representing Elements (ex-f86c65)

We use the quadratic forms example considered above to illustrate these functions.

```
> Generators(QF);
[ <2,2,500001>, <206,-102,4867> ]
> g := QF ! [5, 6];
> g;
<837,-766,1370>
> Representation(g);
[ 1, 6 ]
>
> g := Random(QF);
> Representation(g);
[ 1, 270 ]
>
> UserRepresentation(g);
[ 377, 0, 515, 0, 0, 0, 0, 0, 0, 0 ]
>
> S := [];
```

(continues on next page)

(continued from previous page)

```
> for i in [1..3] do
>     d := Random(QF);
>     Include(~S, d);
> end for;
> seq, coeff := Representation(S, g);
> seq; coeff;
[ -170, -3, 0 ]
1
```

77.4.3 Arithmetic with Elements

If the generic abelian group A has been constructed with the flag `UseRepresentation` set true, then arithmetic with elements of A is trivial.

`u + v: GrpAbElt, GrpAbElt -> GrpAbElt`

Given elements u and v belonging to the same abelian group A, return the sum of u and v.

`- u: GrpAbElt -> GrpAbElt`

The inverse of element u.

`u - v: GrpAbElt, GrpAbElt -> GrpAbElt`

Given elements u and v belonging to the same abelian group A, return the sum of u and the inverse of v.

`m * u: RngIntElt, GrpAbElt -> GrpAbElt`

`u * m: GrpAb, RngIntElt -> GrpAbElt`

Given an integer m, return the element $w+w+\cdots w$ ($|m|$ summands), where $w = u$, if m is positive and $w = -u$ if m is negative.

77.5 Construction of Subgroups and Quotient Groups

The operations in this section apply to both, free abelian groups and arbitrary abelian groups.

77.5.1 Construction of Subgroups

`sub<A | L>: GrpAb, List -> GrpAb, Map`

Construct the subgroup B of the abelian group A generated by the elements specified by the terms of the *generator list* L. A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable to A;

(b) A sequence of integers representing an element of A;

(c) A subgroup of A;

(d) A set or sequence of type (a), (b), or (c).

The collection of words and groups specified by the list must all belong to the group A and the group B will be constructed as a subgroup of A.

Example: Creating a Subgroup of an Abelian Group (ex-f256c6)

We create a subgroup of the group $A = Z_2 + Z_3 + Z_4 + Z_5 + Z_6 + Z + Z$.

```
> A<[x]> := AbelianGroup([2,3,4,5,6,0,0]);
> A;
Abelian Group isomorphic to Z/2 + Z/6 + Z/60 + Z + Z
Defined on 7 generators
```

(continues on next page)

(continued from previous page)

```
Relations:
    2*x[1] = 0
    3*x[2] = 0
    4*x[3] = 0
    5*x[4] = 0
    6*x[5] = 0
> B<[y]> := sub< A | x[1], x[3], x[5], x[7] >;
> B;
Abelian Group isomorphic to Z/2 + Z/2 + Z/12 + Z
Defined on 4 generators in supergroup A:
    y[1] = 2*x[3] + 3*x[5]
    y[2] = x[1]
    y[3] = 3*x[3] + x[5]
    y[4] = x[7]
Relations:
    2*y[1] = 0
    2*y[2] = 0
    12*y[3] = 0
```

In the case of subgroups of generic groups, a number of parameters are provided.

sub<A | L: parameters>: GrpAbGen, List -> GrpAbGen

```
Order            : RngInt                    Default:
RandomIntrinsic  : MonStg                    Default:
ComputeStructure : Bool                      Default: false
UseUserGenerators: Bool                      Default: false
PollardRhoRParam : RngInt                    Default: 20
PollardRhoTParam : RngInt                    Default: 8
PollardRhoVParam : RngInt                    Default: 3
```

Construct the subgroup of the generic abelian group A generated by the elements specified by the terms of the *generator list* L. A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable into A;

(b) A sequence of integers representing an element of A;

(c) A set or sequence whose elements may be of either of the above types.

An element liftable into A may be an element of A itself, or it may be an element of U (`Universe(U)`), U being as usual the domain over which A is defined. For consistency with the construction, the values of the following parameters may also be passed to the subgroup constructor:

In particular, it is possible to construct a subgroup by giving its order and a random function generating elements of the *subgroup*. In this case, the list L would be empty since calculation of the subgroup structure would result in the construction of the p-Sylow subgroups from random elements of the subgroup. Further, when the structure of A is already known and if the subgroup is defined in terms of a set of generators in L then the subgroup structure is computed at the time of creation.

Example: Subgroup of a Quadratic Forms Group (ex-8e5822)

We create a subgroup of the quadratic forms group considered above.

```
> S := [];
> for j in [1..2] do
>     P := Random(QF);
```

(continues on next page)

(continued from previous page)

```
>     Include(~S, P);
> end for;
> S;
[ <45,26,22226>, <937,-930,1298> ]
> QF1 := sub< QF | S>;
> QF1;
Generic Abelian Group over
Binary quadratic forms of discriminant -4000004
Abelian Group isomorphic to Z/2 + Z/258
Defined on 2 generators in supergroup A:
  QF1.1 = QF.1
  QF1.2 = 2*QF.2
Relations:
  2*QF1.1 = 0
  258*QF1.2 = 0
>
```

77.5.2 Construction of Quotient Groups

`quo<F | R>: GrpAb, List -> GrpAb, Hom(GrpAb)`

Given an abelian group F, and a set of relations R in the generators of F, construct (a) an abelian group A isomorphic to the quotient of F by the subgroup of F defined by R, and (b) the natural homomorphism $\phi : F \to A$.

The expression defining F may be either simply the name of a previously constructed group, or an expression defining an abelian group. The possibilities for the relation list R are the same as for the `AbelianGroup` construction.

The function returns:

(a) The quotient group A;

(b) The natural homomorphism $\phi : F \to A$.

`A / B: GrpAb, GrpAb -> GrpAb`

Given a subgroup B of the abelian group A, construct the quotient of A by B.

77.6 Standard Constructions and Conversions

`AbelianGroup(GrpAb, Q): Cat, [ RngIntElt ] -> GrpAb`

`AbelianGroup(Q): [ RngIntElt ] -> GrpAb`

Let $Q = [a_1, \ldots, a_r]$ be a sequence of non-negative integers. This function creates the abelian group $\mathbb{Z}_1 + \cdots + \mathbb{Z}_r$, where Z_i is the cyclic group of order $|a_i|$ if $a_i \neq 0$ or the infinite cyclic group $\mathbb{Z}$ otherwise, $i = 1, \ldots, r$.

`AbelianGroup(G): Grp -> GrpAb, Hom`

Given an abelian permutation, matrix, or polycyclic group G, represent it as an abelian group A. The function also returns the isomorphism $\phi : G \to A$ as its second value.

`AbelianQuotient(G): Grp -> GrpAb, Hom`

Given a finitely presented, permutation, matrix, or polycyclic group G, return the maximal abelian quotient A of G. The function returns the natural homomorphism $\phi : G \to A$ as its second value.

`DirectSum(A, B): GrpAb, GrpAb -> GrpAb`

The direct sum of abelian groups A and B.

`PCGroup(A): GrpAb -> GrpPC, Hom(Grp)`

A pc-group representation G of A. The isomorphism $\phi : A \to G$ is also returned.

`PermutationGroup(A): GrpAb -> GrpPerm, Hom(Grp)`

A permutation group representation of A. The particular group G is generated by disjoint cycles whose lengths are the abelian invariants of A. The isomorphism $\phi : G \to A$ is also returned.

`FPGroup(A): GrpAb -> GrpFP, Hom(Grp)`

A fp-group representation of A. The particular group G is generated by commuting generators whose orders are the abelian invariants of A. The isomorphism $\phi : G \to A$ is also returned.

`CommutatorSubgroup(G): GrpAb -> GrpAb`
`DerivedSubgroup(G): GrpAb -> GrpAb`
`DerivedGroup(G): GrpAb -> GrpAb`

The derived subgroup of G, that is the trivial group, since G is abelian.

`CommutatorSubgroup(H, K): GrpAb, GrpAb -> GrpAb`
`CommutatorSubgroup(G, H, K): GrpAb, GrpAb, GrpAb -> GrpAb`

The commutator subgroup of groups H and K in their common overgroup G.

`Centralizer(G, a): GrpAb, GrpAbElt -> GrpAb`
`Centraliser(G, a): GrpAb, GrpAbElt -> GrpAb`

The centraliser of a in G.

`Core(G, H): GrpAb, GrpAb -> GrpAb`

The maximal normal subgroup of G that is contained in the subgroup H of G. Since G is abelian, this is H itself.

`Centre(G): GrpAb -> GrpAb`
`Center(G): GrpAb -> GrpAb`

The center of G, ie. G itself.

`FittingGroup(G): GrpAb -> GrpAb`
`FittingSubgroup(G): GrpAb -> GrpAb`

The Fitting subgroup of G.

`Hypercentre(G): GrpAb -> GrpAb`
`Hypercenter(G): GrpAb -> GrpAb`

The hypercentre of G.

77.7 Operations on Elements

77.7.1 Order of an Element

`Order(x): GrpAbElt -> RngIntElt`

Order of the element x belonging to an fp-abelian group. If the element has infinite order, the value zero is returned.

Example: Orders of Abelian Group Elements (ex-5056af)

We compute the orders of some elements in the group $Z_2 \times Z_3 \times Z_4 \times Z_5 \times Z$.

```
> G<[x]> := AbelianGroup([2,3,4,5,0]);
> Order( x[1] + 2*x[2] + 3*x[4]);
30
> Order( x[1] + x[5] );
0
```

It is possible to determine the order of an element of a generic group with first calculating the structure of the group. The order function involves the use of one of several algorithms:

– an improved baby-step giant-step algorithm, due to J. Buchmann, M.J. Jacobson, E. Teske [BJT97],

– the Pollard–Rho method based algorithm, described above [Tes98a].

When computing the order of an element whose lower and upper bounds are known, or where lower and upper bounds for the group order are known, the following two algorithms have been shown to be significantly faster than the two algorithms mentioned above.

– the standard baby-step giant-step Shanks algorithm,

– another variant of the Pollard–Rho method which is due to P. Gaudry and R. Harley [GH].

To avoid confusion we will distinguish the algorithms due to Teske *et al* and name them the `T` baby-step giant-step algorithm and the `T` Pollard–Rho algorithm respectively. The Pollard–Rho algorithm has smaller space requirements than the baby-step giant-step algorithm, so it is recommended for use in very large groups.

In all cases, if the group order is known beforehand, the element order is computed using this knowledge. This is a trivial operation.

`Order(g: parameters): GrpAbGenElt -> RngIntElt`

```
ComputeGroupOrder: Bool                    Default: true
BSGSLowerBound   : RngIntElt               Default: 0
BSGSStepWidth    : RngIntElt               Default: 0
```

Assume that g is an element of the generic abelian group A. This functions returns the order of the element g. Note that if `UseRepresentation` is set to true for A, then this is a trivial operation.

If the parameter `ComputeGroupOrder` is true, the order of A is computed first (unless it is already known). The order of g is then computed using this knowledge; this last computation is trivial.

If `ComputeGroupOrder` is false, the order of g is computed using the `T` baby-step giant-step algorithm.

`BSGSLowerBound` and `BSGSStepWidth` are parameters which can be passed to the baby-step giant-step algorithm.

`BSGSLowerBound` sets a lowerbound on the order of the element g, and `BSGSStepWidth` sets the step-width in the algorithm.

`Order(g, l, u: parameters): GrpAbGenElt, RngIntElt, RngIntElt -> RngIntElt`

```
Alg         : MonStg                       Default: "Shanks"
UseInversion: BoolElt                      Default: false
```

Assume that g is an element of the generic abelian group A such that the order of g or the order of A is bounded by u and l. This function returns the order of the element g.

If the parameter `Alg` is set to `"Shanks"` then the generic Shanks algorithm is used, and when `Alg` is set to `"PollardRho"`, the Gaudry–Harley Pollard–Rho variant is used. Setting `UseInversion` halves the search space. To be effective element inversion must be fast.

`Order(g, l, u, n, m: parameters): GrpAbGenElt, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

```
Alg         : MonStg                       Default: "Shanks"
UseInversion: BoolElt                      Default: false
```

Assume that g is an element of the generic abelian group A such that the order of g or the order of A is bounded by u and l. Assume also that `Order(g)` $\equiv n$ `(mod m)` or that $\#A \equiv$ `(mod m)` This function returns the order of the element g. The two parameters `Alg` and `UseInversion` have the same meaning as for the previous `Order` function.

77.7.2 Discrete Logarithm

`Log(g, d: parameters): GrpAbGenElt, GrpAbGenElt -> RngIntElt`

```
ComputeGroupOrder    : Bool                          Default: true
AlInPohligHellmanLoop: MonStg                        Default: "BSGS"
BSGSStepWidth        : RngIntElt                     Default: 0
PollardRhoRParam     : RngInt                        Default: 20
PollardRhoTParam     : RngInt                        Default: 8
PollardRhoVParam     : RngInt                        Default: 3
```

Assume that g and d are elements of the generic abelian group A. This function returns the discrete logarithm of d to the base g. If `ComputeGroupOrder` is true, then the group order is computed first (if not already known) and from this the order of the base g is computed. The discrete logarithm problem is then solved using a Pohlig–Hellman loop calling either the `T` baby-step giant-step algorithm (`AlInPohligHellmanLoop := "BSGS"`) or the `T` Pollard–Rho algorithm (`AlInPohligHellmanLoop := "PollardRho"`).

The parameter `BSGSStepWidth` has the same meaning as for the Order function. Parameters `PollardRhoRParam`, `PollardRhoTParam`, and `PollardRhoVParam` have the same meaning as they do for the determination of structure (function AbelianGroup). If `ComputeGroupOrder` is false then the discrete logarithm problem is solved using the `T` baby-step giant-step algorithm. Here again the parameter `BSGSStepWidth` applies.

Example: Discrete Logarithm for Quadratic Forms (ex-cb963b)

It is assumed that the structure of the groups QF has already been computed. We illustrate the computation of the discrete logarithm relative to a given base.

```
> n := 38716;
> Ip := Reduction(PrimeForm(Q,11));
> g := GA_qf!Ip;
> d := n * g;
>
> l1 := Log(g, d : BSGSStepWidth := Floor((-Dk)^(1/4)/2));
> l2 := Log(g, d : AlInPohligHellmanLoop := "PollardRho");
> l3 := Log(g, d : ComputeGroupOrder := false);
> l4 := Log(g, d: ComputeGroupOrder := false,
>                 BSGSStepWidth := Floor((-Dk)^(1/4)/2));
> assert l1 eq l2 and l2 eq l3 and l3 eq l4;
> assert IsDivisibleBy(n - l1, Order(g));
```

77.7.3 Equality and Comparison

`u eq v: GrpAbElt, GrpAbElt -> BoolElt`

Returns `true` if the elements u and v are identical (as elements of the appropriate free abelian group), `false` otherwise.

`u ne v: GrpAbElt, GrpAbElt -> BoolElt`

Returns `true` if the elements u and v are not identical (as elements of the appropriate free abelian group), `false` otherwise.

`IsIdentity(u): GrpAbElt -> BoolElt`

`IsId(u): GrpAbElt -> BoolElt`

Returns `true` if the element u, belonging to the abelian group A, is the identity element (zero), `false` otherwise.

77.8 Invariants of an Abelian Group

`ElementaryAbelianQuotient(G, p): GrpAb, RngIntElt -> GrpAb, Map`

The maximal p-elementary abelian quotient of the group G as `GrpAb`. The natural epimorphism is returned as second value.

`FreeAbelianQuotient(G): GrpAb -> GrpAb, Map`

The maximal free abelian quotient of the group G as `GrpAb`. The natural epimorphism is returned as second value.

`PrimaryAbelianInvariants(A): GrpAb -> [ RngIntElt ]`

`AbelianInvariants(A): GrpAb -> [ RngIntElt ]`

The p-primary invariants of the abelian group A. Each infinite cyclic factor is represented by zero. The non-primary form gives the Smith form invariants, i.e. each element of the sequence divides the next.

`PrimaryAbelianBasis(A): GrpAb -> [ GrpAbElt ], [ RngIntElt ]`

`AbelianBasis(A): GrpAb -> [ GrpAbElt ], [ RngIntElt ]`

Returns sequences B and I, where I are the p-primary invariants of A, and B are generators for A with orders as in I. The non-primary form uses the Smith form invariants, i.e. each element of the sequence divides the next.

`TorsionFreeRank(A): GrpAb -> RngIntElt`

The torsion-free rank of the abelian group G.

`TorsionInvariants(A): GrpAb -> [ RngIntElt ]`

The torsion invariants of the abelian group G.

`pPrimaryInvariants(A, p): GrpAb, RngIntElt -> [ RngIntElt ]`

The p-primary invariants of the abelian group G.

77.9 Canonical Decomposition

`TorsionFreeSubgroup(A): GrpAb -> GrpAb`

The torsion-free subgroup of the abelian group G.

`TorsionSubgroup(A): GrpAb -> GrpAb`

The torsion subgroup of the abelian group G.

`pPrimaryComponent(A, p): GrpAb, RngIntElt -> GrpAb`

The p-primary component of the abelian group G.

77.10 Set-Theoretic Operations

77.10.1 Functions Relating to Group Order

`Order(G): GrpAb -> RngIntElt`

`# G: GrpAb -> RngIntElt`

The order of the group G, returned as an ordinary integer. If G is an infinite group, the value zero is returned. Note that if G is a generic group then determining the order will require the structure of G to be determined.

`FactoredOrder(G): GrpAb -> [<RngIntElt, RngIntElt>]`

The factored order of the group G, returned as a sequence of prime-exponent pairs. If G is an infinite group, the empty sequence is returned. Note that if G is a generic group then determining the order will require the structure of G to be determined.

`Exponent(G): GrpAb -> RngIntElt`

The exponent of the group G. If the group is infinite, the value zero is returned. Note that if G is a generic group then determining the exponent will require the structure of G to be determined.

`IsFinite(G): GrpAb -> BoolElt`

Return `true` if the group G is finite.

`IsInfinite(G): GrpAb -> BoolElt`

Return `true` if G is infinite, `false` otherwise.

77.10.2 Membership and Equality

`g in G: GrpAbElt, GrpAb -> BoolElt`

Given an element g and a group G, return `true` if g is an element of G, `false` otherwise.

`g notin G: GrpAbElt, GrpAb -> BoolElt`

Given an element g and a group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G: { GrpAbElt}, GrpAb -> BoolElt`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G: { GrpAbElt}, GrpAb -> BoolElt`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G: GrpAb, GrpAb -> BoolElt`

`IsSubgroup(H, G): GrpAb, GrpAb -> BoolElt`

Given groups G and H, subgroups of some common overgroup, return `true` if H is a subgroup of G, and `false` otherwise.

`H notsubset G: GrpAb, GrpAb -> BoolElt`

Given groups G and H, subgroups of some common overgroup, return `true` if H is not a subgroup of G, and `false` otherwise.

`G eq H: GrpAb, GrpAb -> BoolElt`

Given groups G and H, subgroups of some common overgroup, return `true` if G and H are identical, and `false` otherwise.

`G ne H: GrpAb, GrpAb -> BoolElt`

Given groups G and H, subgroups of some common overgroup, return `true` if G and H are distinct groups, and `false` otherwise.

77.10.3 Set Operations

`NumberingMap(G): GrpAb -> Map`

A bijective mapping from the finite group G onto the set of integers $\{1 \dots |G|\}$. The actual mapping depends upon choice of standard generators for G.

`RandomProcess(G): GrpAb -> Process`

```
Slots   : RngIntElt                    Default: 10
Scramble: RngIntElt                    Default: 100
```

Create a process to generate randomly chosen elements from the finite group G. The process is based on the product-replacement algorithm of [CLGM+95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G) + 1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call

chooses one of the elements in the slots and adds it into the accumulator. The element in that slot is replaced by the sum of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such sum-replacement operations to be performed before the process is returned. Note that this algorithm cannot produce well-distributed random elements of an infinite group.

`Random(P): Process -> GrpAbElt`

Given a random element process P created by the function `RandomProcess(G)` for the finite abelian group G, return the next random element of G defined by the process.

`Random(G): GrpAb -> GrpAbElt`

`Random(G, n): GrpAb, RngIntElt -> GrpAbElt`

An element chosen at random from the abelian group G. The argument n gives the maximum exponent for infinite generators, and defaults to 10 if not supplied.

`Rep(G): GrpAb -> GrpAbElt`

`Representative(G): GrpAb -> GrpAbElt`

A representative element of G.

77.11 Coset Spaces

`Transversal(G, H): GrpAb, GrpAb -> {@ GrpAbElt @}, Map`

`RightTransversal(G, H): GrpAb, GrpAb -> {@ GrpAbElt @}, Map`

Given a group G and a subgroup H of G, this function returns:

(a) An indexed set of elements T of G forming a right transversal for G over H; and,

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = \{t_1, \ldots, t_r\}$ and g in G, ϕ is defined by $\phi(g) = t_i$, where $g \in Ht_i$.

77.11.1 Coercions Between Groups and Subgroups

`G ! g: GrpAb, GrpAbElt -> GrpAbElt`

Given an element g belonging to the subgroup H of the group G, rewrite g as an element of G.

`H ! g: GrpAb, GrpAbElt -> GrpAbElt`

Given an element g belonging to the group G, and given a subgroup H of G containing g, rewrite g as an element of H.

`K ! g: GrpAb, GrpAbElt -> GrpAbElt`

Given an element g belonging to the group H, and a group K, such that H and K are subgroups of G, and both H and K contain g, rewrite g as an element of K.

`Morphism(H, G): GrpAb, GrpAb -> ModMatRngElt`

The integer matrix defining the inclusion monomorphism from the subgroup H of G into G.

77.12 Subgroup Constructions

Although, in the case of an abelian group, many of the standard subgroup constructors are trivial, they are all implemented for the sake of uniformity. Here we document only those which are meaningful in the context of abelian groups.

`H meet K: GrpAb, GrpAb -> GrpAb`

Given subgroups H and K of some group G, construct their intersection.

`H meet:= K: GrpAb, GrpAb -> GrpAb`

Replace H with the intersection of groups H and K.

`H + K: GrpAb, GrpAb -> GrpAb`

Given subgroups H and K of some group G, construct the smallest subgroup containing both.

`n * G: RngIntElt, GrpAb -> GrpAb, Map`

For an integer n and some abelian group G, construct the subgroup nG. The second return value is the map $G \to G$ sending $g \to ng$.

`FrattiniSubgroup(G): GrpAb -> GrpAb`

The Frattini subgroup of the finite abelian group G.

`SylowSubgroup(G, p : parameters): GrpAb, RngIntElt -> GrpAb`

`Sylow(G, p : parameters): GrpAb, RngIntElt -> GrpAb`

Structure: Bool	Default: false

The Sylow p-subgroup for the group G. If G is a generic group and the parameter `Structure` is true, or if the group structure of A is known, then the group structure of the Sylow subgroup is computed.

Example: Sylow 2-Subgroup of a Unit Group (ex-c74c85)

In the following example, we construct the Sylow 2-subgroup of $G = Z_{34384}$.

```
> m := 34384;
> Zm := Integers(m);
> U := {@ x : x in Zm | GCD(x, m) eq 1 @};
> G := GenericAbelianGroup(U : IdIntrinsic := "Id",
>    AddIntrinsic := "*", InverseIntrinsic := "/");
> _ := AbelianGroup(G);
> Factorization(#G);
> Sylow(G, 2);
2-Sylow subgroup: Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4
Defined on 4 generators in supergroup G:
  GAp.1 = G.1
  GAp.2 = G.2
  GAp.3 = 3*G.3
  GAp.4 = 153*G.4
Relations:
  2*GAp.1 = 0
  2*GAp.2 = 0
  2*GAp.3 = 0
  4*GAp.4 = 0
```

77.13 Subgroup Chains

Series involving centres or commutator subgroups are likely to be trivial for abelian groups, but are noted here for completeness.

`ChiefSeries(G): GrpAb -> [GrpAb]`

`CompositionSeries(G): GrpAb -> [GrpAb]`

A composition series for the finite abelian group G returned as a sequence of subgroups.

`Agemo(G, i): GrpAb, RngIntElt -> GrpAb`

Given a finite p-group G, return the characteristic subgroup of G generated by the elements x^{p^i}, $x \in G$, where i is a positive integer.

`Omega(G, i): GrpAb, RngIntElt -> GrpAb`

Given a finite p-group G, return the characteristic subgroup of G generated by the elements of order dividing p^i, where i is a positive integer.

`ElementaryAbelianSeries(G): GrpAb -> [ GrpAb ]`

A descending series of subgroups of G where each quotient is elementary abelian.

`DerivedSeries(G): GrpAb -> [ GrpAb ]`

`LowerCentralSeries(G): GrpAb -> [ GrpAb ]`

The derived series of G.

`UpperCentralSeries(G): GrpAb -> [ GrpAb ]`

The upper central series of G.

`SubnormalSeries(G, H): GrpAb, GrpAb -> [ GrpAb ]`

A subnormal series from G to the subnormal subgroup H.

77.14 General Group Properties

`IsCyclic(G): GrpAb -> BoolElt`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G): GrpAb -> BoolElt`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsFree(G): GrpAb -> BoolElt`

Returns `true` if G is free, `false` otherwise.

`IsMixed(G): GrpAb -> BoolElt`

Returns `true` if G is a mixed group, `false` otherwise. An abelian group is mixed if it is neither a torsion group nor free.

`IspGroup(G): GrpAb -> BoolElt`

Returns `true` if the finite group G is a p-group, i.e. if all elements have order a power of p.

`DerivedLength(G): GrpAb -> RngIntElt`

The length of the derived series of G.

77.14.1 Properties of Subgroups

`IsMaximal(G, H): GrpAb, GrpAb -> BoolElt`

Returns `true` if the subgroup H of the finite group G is a maximal subgroup of G, `false` otherwise.

`Index(G, H): GrpAb, GrpAb -> RngIntElt`

The index of the subgroup H in the group G, returned as an ordinary integer. If H has infinite index in G, the value zero is returned.

`FactoredIndex(G, H): GrpAb, GrpAb -> [<RngIntElt, RngIntElt>]`

The factored index of the subgroup H in the group G, returned as a sequence of prime-exponent pairs. If H has infinite index in G, the empty sequence is returned.

`IsPure(G, H): GrpAb, GrpAb -> BoolElt`

Returns `true` if the subgroup H of the finite group G is pure, ie. if for all n we have $nG \cap H = nH$.

`IsNeat(G, H): GrpAb, GrpAb -> BoolElt`

Returns `true` if the subgroup H of the finite group G is neat, i.e., if for all primes p we have $pG \cap H = pH$.

77.14.2 Enumeration of Subgroups

MaximalSubgroups(G): GrpAb -> [GrpAb]

The maximal subgroups of the finite group G returned as a sequence of subgroups.

Subgroups(G:parameters): GrpAb -> [Rec]

```
Sub : [RngIntElt]                    Default: []
Quot: [RngIntElt]                    Default: []
```

The subgroups of the finite group G are returned as a sequence of records. The record fields are `subgroup`, storing the actual group; `order`, storing the group order; and `length`, storing the length of the conjugacy class, which is always 1 for abelian groups.

If the parameter `Sub` is set, only subgroups with invariants equal to the given sequence are found. The given sequence should contain positive integers, such that each divides the following.

If the parameter `Quot` is set, only subgroups such that the quotient group has invariants equal to the given sequence are found. The given sequence should contain positive integers, such that each divides the following.

NumberOfSubgroupsAbelianPGroup(A): SeqEnum -> SeqEnum

Return the number of subgroups of each non-trivial order in the abelian p-group G where $A = [a_1, a_2, \ldots]$ and $G = C_{a_1} \times C_{a_2} \times \ldots$. The m-th entry in the sequence returned is the number of subgroups of order p^m.

HasComplement(G, U): GrpAb, GrpAb -> BoolElt, GrpAb

For a finite abelian group G and a subgroup U decide if there exist some other subgroup V such that $G = U + V$ and $U \cap V = \{0\}$. In case such a V exists, it is returned as the second value.

Example: Subgroups of an Abelian Group of Order 12 (ex-604d09)

We look at subgroups of an abelian group of order 12.

```
> G := AbelianGroup([2,6]);
> s := Subgroups(G); #s;
10
> s[7];
rec<recformat<order, length, subgroup, presentation> |
   order := 3, length := 1,
   subgroup := Abelian Group isomorphic to Z/3
Defined on 1 generator in supergroup G:
  $.1 = 2*G.2
Relations:
  3*$.1 = 0>
> [x`order:x in s];
[ 12, 6, 4, 2, 6, 6, 3, 2, 2, 1 ]
```

Now we find the elementary abelian subgroup of order 4.

```
> s22 := Subgroups(G:Sub := [2,2]); #s22;
1
> s22;
Conjugacy classes of subgroups
------------------------------

[1]     Order 4            Length 1
        Abelian Group isomorphic to Z/2 + Z/2
        Defined on 2 generators in supergroup G:
          $.1 = G.1
          $.2 = 3*G.2
```

(continues on next page)

(continued from previous page)

```
        Relations:
          2*$.1 = 0
          2*$.2 = 0
```

There is more than one subgroup of index 2 in G.

```
> q2 := Subgroups(G:Quot := [2]); #q2;
3
> q2[3]`subgroup;
Abelian Group isomorphic to Z/6
Defined on 1 generator in supergroup G:
  $.1 = G.1 + G.2
Relations:
  6*$.1 = 0
```

77.15 Representation Theory

CharacterTable(G): GrpAb -> TabChtr

The table of irreducible characters for the group G.

77.16 The Hom Functor

Hom(G, H): GrpPC, GrpPC -> GrpAb, Map

Hom(G, H): GrpAb, GrpAb -> GrpAb, Map

Given finite abelian groups G and H, return an abelian group A isomorphic to $\mathrm{Hom}(G, H)$, and a transfer map t such that, given an element a of A, $t(a)$ yields the corresponding (Magma `Map` type) homomorphism from G to H. The structure of $\mathrm{Hom}(G, H)$ may thus be analyzed by examining A.

HomGenerators(G, H): GrpAb, GrpAb -> GrpAb, Map

Given finite abelian groups G and H, return a sequence of (Z-module) generators of $\mathrm{Hom}(G, H)$. The generators are returned as actual (Magma `Map` type) homomorphisms. Note that $\mathrm{Hom}(G, H)$ is usually not free, so it is difficult to generate all homomorphisms uniquely using the generators alone (use `Hom` or `Homomorphisms` if that is desired).

AllHomomorphisms(G, H): GrpAb, GrpAb -> [Map]

AllHomomorphisms(G, H): GrpPC, GrpPC -> [Map]

Homomorphisms(G, H): GrpAb, GrpAb -> [Map]

Given finite abelian groups G and H, return a sequence containing all elements of $\mathrm{Hom}(G, H)$. The elements are returned as actual (Magma `Map` type) homomorphisms. Note that this function simply uses `Hom`, transferring each element of the returned group to the actual Magma `Map` type homomorphism.

Example: Hom Functor on Abelian Groups (ex-0d5388)

We examine $A = \mathrm{Hom}(G, H)$, for certain abelian groups G and H.

```
> G := AbelianGroup([2, 3]);
> H := AbelianGroup([4, 6]);
> A, t := Hom(G, H);
> #A;
12
> A;
Abelian Group isomorphic to Z/2 + Z/6
Defined on 2 generators
```

(continues on next page)

(continued from previous page)

```
Relations:
    2*A.1 = 0
    6*A.2 = 0
> h := t(A.1);
> h;
Mapping from: GrpAb: G to GrpAb: H
> h(G.1);
3*H.2
> h(G.2);
0
```

We now enumerate all elements of A and examine the images of each generator of G under each homomorphism. We note that each possible list of images occurs only once.

```
> I := [<h(G.1), h(G.2)> where h is t(x): x in A];
> I;
[
    <0, 0>,
    <3*H.2, 0>,
    <2*H.1, 2*H.2>,
    <2*H.1 + 3*H.2, 2*H.2>,
    <0, 4*H.2>,
    <3*H.2, 4*H.2>,
    <2*H.1, 0>,
    <2*H.1 + 3*H.2, 0>,
    <0, 2*H.2>,
    <3*H.2, 2*H.2>,
    <2*H.1, 4*H.2>,
    <2*H.1 + 3*H.2, 4*H.2>
]
> #I;
12
> #Set(I);
12
```

77.17 Automorphism Groups

AutomorphismGroup(G): GrpAb -> GrpAuto

The full automorphism group of the abelian group G.

IsIsomorphic(G, H): GrpAb, GrpAb -> BoolElt, Hom(Grp)

Test whether or not the two abelian groups G and H are isomorphic as abstract groups. If so, both the result `true` and an isomorphism from G to H is returned. If not, the result `false` is returned.

77.18 Cohomology

Dual(G): GrpAb -> GrpAb, Map

Computes the dual group G^* of G and a map M from $G \times G^* \to \mathbb{Z}/m\mathbb{Z}$ for m the exponent of G that allows G^* to act on G. The group G must be finite.

H2_G_QmodZ(G): GrpAb -> GrpAb, Map

Computes $H := H^2(G, \mathbb{Q}/\mathbb{Z})$ and a map $f : H \to (G \times G \to \mathbb{Z}/m\mathbb{Z})$ that will give the cocycles as maps from $G \times G \to \mathbb{Z}/m\mathbb{Z}$, $m := \#G$.

`Res_H2_G_QmodZ(U, H2): GrpAb, GrpAb -> GrpAb, Map`

For a subgroup U of G and $H2 = H^2(G, \mathbb{Q}/\mathbb{Z})$ computes $H^2(U, \mathbb{Q}/\mathbb{Z})$ in a compatible way together with the restriction map into $H2$.

The abelian group $H2$ must be the result of `H2_G_QmodZ` as this function relies on the attributes stored in there.

77.19 Homomorphisms

Two functions are provided to construct homomorphisms or isomorphisms from one group into another, where either of the groups, or both, may be generic abelian groups.

`hom< A -> B | L>: Grp, Grp, List -> Map`

Given groups A and B, construct a homomorphism from A to B as defined by the extension L. If one or both of A and B are generic abelian groups this works as usual, with one minor difference as explained below. Suppose that the generators of A are $g_1, \ldots, g_n$, and that $\phi(g_i) = h_i$ for each i, where ϕ is the homomorphism one wishes to construct. The list L as required by the constructors must be one of the following:

(a) a list of the n 2-tuples $< g_i, h_i >$ (order not important);

(b) a list of the n arrow-pairs $g_i \to h_i$ (order not important);

(c) $h_1, \ldots, h_n$ (order is important).

If A is a generic abelian group this rule is relaxed somewhat in the following sense: If L is a list of n 2-tuples or of n arrow-pairs, the elements g_i need not be the defining generators of A. The only requirement is that the set $\{g_1, \ldots, g_n\}$ does actually generate the whole of A.

`Homomorphism(A, B, X, Y): Grp, Grp, [ GrpElt ], [ GrpElt ] -> Map`

Creates a homomorphism from A into B as given by the mapping of X into Y. The arguments A and B may be any type of group, including of course, generic abelian groups.

The function `Homomorphism` does not require the elements of the argument X to be generators of A as given by `Generators(A)`, so it allows more freedom when creating a homomorphism. If, however, these elements fail to generate the whole of A then the subsequent map application will fail.

`iso< A -> B | L>: Grp, Grp, List -> Map`

Given groups A and B, construct an isomorphism from A to B as defined by the extension L. If one or both of A and B are generic abelian groups this works as usual, with one minor difference as explained below. Suppose that the generators of A are $g_1, \ldots, g_n$, and that $\phi(g_i) = h_i$ for each i, where ϕ is the isomorphism one wishes to construct. The list L as required by the constructors must be one of the following:

(a) a list of the n 2-tuples $< g_i, h_i >$ (order not important);

(b) a list of the n arrow-pairs $g_i \to h_i$ (order not important);

(c) $h_1, \ldots, h_n$ (order is important).

If A is a generic abelian group this rule is relaxed somewhat in the following sense: If L is a list of n 2-tuples or of n arrow-pairs, the elements g_i may not necessarily be generators of A as given by the function `Generators(A)`. The only requirement is that the set $\{h_1, \ldots, h_n\}$ does actually generate the whole of B.

`Isomorphism(A, B, X, Y): Grp, Grp, [ GrpElt ], [ GrpElt ] -> Map`

Creates a isomorphism from A into B as given by the mapping of X into Y. The arguments A and B can be any type of group, including of course, generic abelian groups.

The function `Isomorphism` does not require the elements of the argument X to be generators of A as given by `Generators(A)`, so it allows more freedom when creating an isomorphism. If, however, these elements fail to generate the whole of A then the subsequent map application will fail.

Example: Homomorphisms of Generic Abelian Groups (ex-32bea9)

Recall that we defined the subgroups $GH1_{Zm}$ and $GH2_{Zm}$ of G as:

```
> GH1_Zm;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/6 + Z/612
Defined on 2 generators in supergroup G:
  GH1_Zm.1 = G.2 + G.3
  GH1_Zm.2 = G.4
Relations:
  6*GH1_Zm.1 = 0
  612*GH1_Zm.2 = 0
> GH2_Zm;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/6 + Z/612
Defined on 4 generators in supergroup G:
  GH2_Zm.1 = G.1
  GH2_Zm.2 = G.2
  GH2_Zm.3 = G.3
  GH2_Zm.4 = G
Relations:
  2*GH2_Zm.1 = 0
  2*GH2_Zm.2 = 0
  6*GH2_Zm.3 = 0
  612*GH2_Zm.4 = 0
```

We construct the homomorphism

```
> h := hom<GH1_Zm -> GH2_Zm | GH2_Zm.1, GH2_Zm.2 >;
> h(GH1_Zm);
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators in supergroup GH2_Zm:
  $.1 = GH2_Zm.2
  $.2 = GH2_Zm.1
Relations:
  2*$.1 = 0
  2*$.2 = 0
```

but we cannot construct the isomorphism

```
> i := iso<GH1_Zm -> GH2_Zm | GH2_Zm.1, GH2_Zm.2 >;

>> i := iso<GH1_Zm -> GH2_Zm | GH2_Zm.1, GH2_Zm.2 >;
             ^
Runtime error in map< ... >: Images do not generate the (whole) codomain
```

An alternative way of creating the homomorphism h would be

```
> h := Homomorphism(GH1_Zm, GH2_Zm, Generators(GH1_Zm), [GH2_Zm.1, GH2_Zm.2]);
```

77.20 Bibliography

- [BJT97] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.
- [CLGM+95] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.
- [GH] P. Gaudry, R. Harley. "Counting Points on Hyperelliptic Curves over Finite Fields." 313–332.
- [Tes98a] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.
- [Tes98b] E. Teske. "Better Random Walks for Pollard's Rho Method." 1998.

Chapter 78

Free Groups

78.1 Introduction

This chapter presents the functions designed for computing with free groups and their words.

78.2 Free Groups and Words

78.2.1 Construction of a Free Group

`FreeGroup(n): RngIntElt -> GrpFP`

Construct the free group F of rank n, where n is a positive integer.

The i-th generator of F may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the *left hand side* of the assignment statement:

```
F< v_1, ... v_n > := FreeGroup(n);
```

Example: Creating a Free Group (ex-f04611)

The statement

```
> F := FreeGroup(2);
```

creates the free group of rank 2. Here the generators may be referenced using the standard names, `F.1` and `F.2`. The statement

```
> F<x, y> := FreeGroup(2);
```

defines F to be the free group of rank 2 and assigns the names x and y to the two generators.

78.2.2 Construction of Words

The operations in this section apply to both, free groups and arbitrary fp-groups.

`G ! [ i1, ..., is ]: GrpFP, [ RngIntElt ] -> GrpFPElt`

Given a group G defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[-r, r]$, excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \cdots * G.|i_s|^{\epsilon_s}$$

where ϵ_j is $+1$ if i_j is positive, and -1 if i_j is negative.

`Identity(G): GrpFP -> GrpFPElt`

`Id(G): GrpFP -> GrpFPElt`

`G ! 1: GrpFP, RngIntElt -> GrpFPElt`

Construct the identity element, represented as the empty word, for the fp-group G. For a sample application of this function, see Example *Length, Weight, and Leading Generator of Words*.

`Random(G, m, n): GrpFP, RngIntElt, RngIntElt -> GrpFPElt`

A random word of length l in the generators of the group G, where $m \leq l \leq n$. For a sample application of this function, see Example *Length, Weight, and Leading Generator of Words*.

78.2.3 Access Functions for Words

This section describes some basic access functions for words. These operations apply to both, free groups and arbitrary fp-groups.

`# w: GrpFPElt -> RngIntElt`

The length of the word w.

`ElementToSequence(w): GrpFPElt -> [ RngIntElt ]`

`Eltseq(w): GrpFPElt -> [ RngIntElt ]`

The sequence Q obtained by decomposing the word w into its constituent generators and generator inverses. Suppose w is a word in the group G. Then, if $w = G.i_1^{e_1} \cdots G.i_m^{e_m}$, with each $e_i = \pm 1$, then $Q[j] = i_j$ if $e_j = +1$ and $Q[j] = -i_j$ if $e_j = -1$, for $j = 1, \ldots, m$.

`ExponentSum(w, x): GrpFPElt, GrpFPElt -> RngIntElt`

`Weight(w, x): GrpFPElt, GrpFPElt -> RngIntElt`

Given a word w, and the name of a generator x of a group G, compute the sum of the exponents of the generator x in the word w. For a sample application of this function, see Example *Length, Weight, and Leading Generator of Words*.

`GeneratorNumber(w): GrpFPElt -> RngIntElt`

Suppose w is a word belonging to a group G. Assume x is the name of the i-th generator of G. Then

(i) if $w =$ `Identity(G)`, `GeneratorNumber(w)` is 0;

(ii) if $w = x * w'$, w' a word in G, `GeneratorNumber(w)` is i;

(iii) if $w = x^{-1} * w'$, w' a word in G, `GeneratorNumber(w)` is $-i$.

`LeadingGenerator(w): GrpFPElt -> GrpFPElt`

Suppose w is a word belonging to a group G. If $w = x^\epsilon * w'$, w' a word in G, x a generator of G, and $\epsilon \in \{-1, +1\}$, the functions returns x^ϵ. If $w =$ `Identity(G)`, it returns `Identity(G)`. For a sample application of this function, see Example *Length, Weight, and Leading Generator of Words*.

`Parent(w): GrpFPElt -> GrpFP`

The parent group G of the word w.

Example: Abelianised Relation Matrix from Word Weights (ex-117fb7)

Consider the free group F on 6 generators

```
> F<u,v,w,x,y,z> := FreeGroup(6);
```

and the sequence of words

```
> rels := [ (u*v)^42, (v,x), (x*z^2)^4,
>           v^2*y^3, (v*z)^3, y^4, (x*z)^3 ];
```

The abelianised relation matrix of the quotient

$$\langle u, v, w, x, y, z \mid (uv)^{42}, (v, x), (xz^2)^4, v^2y^3, (vz)^3, y^4, (xz)^3 \rangle$$

can be obtained using the following construction

```
> R := Matrix(Integers(),
>        [ [ Weight(r, F.j) : j in [1..6] ] : r in rels ]);
> R;
[42 42  0  0  0  0]
[ 0  0  0  0  0  0]
[ 0  0  0  4  0  8]
[ 0  2  0  0  3  0]
[ 0  3  0  0  0  3]
[ 0  0  0  0  4  0]
[ 0  0  0  3  0  3]
```

(The function `Matrix` constructs a matrix from a sequence of row vectors.)

78.2.4 Arithmetic Operators for Words

Suppose G is an fp-group for which generators have already been defined. This subsection defines the elementary arithmetic operations on words that are derived from the multiplication and inversion operators. The availability of the operators defined here enables the user to construct an element (*word*) of G in terms of the generators as follows:

(i) A generator is a word;

(ii) The expression (u) is a word, where u is a word;

(iii) The product $u * v$ of the words u and v is a word;

(iv) The conjugate u^v of the word u by the word v, is a word (u^v expands into the word $v^{-1} * u * v$);

(v) The power of a word, u^n, where u is a word and n is an integer, is a word;

(vi) The commutator (u, v) of the words u and v is a word ((u, v) expands into the word $u^{-1} * v^{-1} * u * v$). Note that (u, v, w) is equivalent to $((u, v), w)$, i.e. commutators are *left-normed*.

The word operations defined here may be applied either to the words of a free group or the words of a group with non-trivial relations. If such an operator is applied to a group possessing non-trivial relations, only free reduction will be applied to the resulting words.

`u * v: GrpFPElt, GrpFPElt -> GrpFPElt`

Given words u and v belonging to the same fp-group G, return the product of u and v.

`u ^ n: GrpFPElt, RngIntElt -> GrpFPElt`

The n-th power of the word u, where n is an integer. When invoked with $n = -1$, the function computes the inverse of u. When invoked with $n = 0$, the function returns the identity element.

`u ^ v: GrpFPElt, GrpFPElt -> GrpFPElt`

Given words u and v belonging to the same fp-group G, return the conjugate $v^{-1} * u * v$ of the word u by the word v.

`(u, v): GrpFPElt, GrpFPElt -> GrpFPElt`

Given words u and v belonging to the same fp-group G, return the commutator $u^{-1}v^{-1}uv$ of the words u and v.

`(u1, ..., un): List(GrpFPElt) -> GrpFPElt`

Given the n words $u_1, \ldots, u_n$ belonging to the same fp-group G, return their commutator. Commutators are *left-normed*, so that they are evaluated from left to right.

78.2.5 Comparison of Words

Words in an fp-group may be compared both for equality and for their relationship with respect to a natural lexicographic ordering. It should be noted that even when a pair of words belong to a group defined by non-trivial relations, only the free reductions of the words are compared. Thus, a pair of

words belonging to a group G may be declared to be distinct even though they may represent the same element of G. The words of an fp-group G are ordered first by length and then lexicographically. The lexicographic ordering is determined by the following ordering on the generators and their inverses:

$$G.1 < G.1^{-1} < G.2 < G.2^{-1} < \cdots$$

Here, u and v are words belonging to some common fp-group.

`u eq v: GrpFPElt, GrpFPElt -> BoolElt`

Return `true` if the free reductions of the words u and v are identical.

`u ne v: GrpFPElt, GrpFPElt -> BoolElt`

Return `true` if the free reductions of the words u and v are not identical.

`u lt v: GrpFPElt, GrpFPElt -> BoolElt`

Return `true` if the word u precedes the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

`u le v: GrpFPElt, GrpFPElt -> BoolElt`

Return `true` if the word u either precedes, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

`u ge v: GrpFPElt, GrpFPElt -> BoolElt`

Return `true` if the word u either follows, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

`u gt v: GrpFPElt, GrpFPElt -> BoolElt`

Return `true` if the word u follows the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

Example: Length, Weight, and Leading Generator of Words (ex-65f8bd)

We construct the free group on three generators and generate a random word w of length between 4 and 6.

```
> F<a,b,c> := FreeGroup(3);
>
> w := Random(F, 4, 6);
> w;
b^-1 * a^-1 * b^2 * a^-1
```

We print the length of w and the weight (the exponent sum) of generator a in w.

```
> #w;
5
>
> Weight(w, a);
-2
```

We now strip the generators from w one by one, using arithmetic and comparison operators for words and the access function *LeadingGenerator*.

```
> while w ne Identity(F) do
>     g := LeadingGenerator(w);
>     print g;
>     w := g^-1 * w;
> end while;
b^-1
```

(continues on next page)

(continued from previous page)

```
a^-1
b
b
a^-1
```

78.2.6 String Operations on Words

The functions described in this section perform low level string operations like substitution, elimination, or substring matching on elements of fp-groups.

`Eliminate(u, x, v): GrpFPElt, GrpFPElt, GrpFPElt -> GrpFPElt`

Given words u and v, and a generator x, all belonging to a group G, return the word obtained from u by replacing each occurrence of x by v and each occurrence of x^{-1} by v^{-1}.

`Eliminate(U, x, v): { GrpFPElt }, GrpFPElt, GrpFPElt -> { GrpFPElt }`

Given a set of words U, a word v, and a generator x, all belonging to a group G, return the set of words obtained by taking each element u of U in turn, and replacing each occurrence of x in u by v and each occurrence of x^{-1} by v^{-1}.

`Match(u, v, f): GrpFPElt, GrpFPElt, RngIntElt -> BoolElt, RngIntElt`

Suppose u and v are words belonging to the same group G, and that f is an integer such that $1 \leq f \leq \#u$. The function seeks the least integer l such that:

(a) $l \geq f$; and

(b) v appears as a subword of u, starting at the l-th letter of u.

If such an integer l is found `Match` returns the value `true` and l. If no such l is found, `Match` returns the value `false`.

`RotateWord(u, n): GrpFPElt, RngIntElt -> GrpFPElt`

The word obtained by cyclically permuting the word u by n places. If n is positive, the rotation is from left to right, while if n is negative the rotation is from right to left. In the case where n is zero, the function returns u.

`Substitute(u, f, n, v): GrpFPElt, RngIntElt, RngIntElt, GrpFPElt -> GrpFPElt`

Given words u and v belonging to a group G, and non-negative integers f and n, this function replaces the substring of u of length n, starting at position f, by the word v. Thus, if $u = x_{i_1}^{e_1} \cdots x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}} \cdots x_{i_m}^{e_m}$ then the substring $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$ is replaced by v. If the function is invoked with $v = \mathrm{Id(G)}$, then the substring $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$ of u is deleted.

`Subword(u, f, n): GrpFPElt, RngIntElt, RngIntElt -> GrpFPElt`

The subword of the word u comprising the n consecutive letters commencing at the f-th letter of u.

Example: String Operations on Words (ex-b14d33)

We demonstrate some of these operations in the context of the free group on generators x, y, and z.

```
> F<x, y, z> := FreeGroup(3);
> u := (x, y*z);
> w := u^(x^2*y);
> #w;
12
> w;
y^-1 * x^-3 * z^-1 * y^-1 * x * y * z * x^2 * y
```

We replace each occurrence of the generator x in w by the word $y * z^-1$.

```
> Eliminate(w, x, y*z^-1);
y^-1 * z * y^-1 * z * y^-1 * z * y^-1 * z^-2 * y * z * y * z^-1 * y * z^-1 * y
```

We count the number of occurrences of each generator in w.

```
> [ ExponentSum(w, F.i) : i in [1..Ngens(F)] ];
[ 0, 0, 0 ]
> GeneratorNumber(w);
-2
```

We locate the start of the word u in the word w.

```
> b, p := Match(w, u, 1);
> b, p;
true 4
```

We now replace the subword u in w by the word $y * x$.

```
> t := Substitute(w, p, #u, y*x);
> t;
y^-1 * x^-2 * y * x^3 * y
```

We create the set of all distinct cyclic permutations of the word u.

```
> rots := { RotateWord(u, i) : i in [1 ..#u] };
> rots;
{ y^-1 * x * y * z * x^-1 * z^-1,  x * y * z * x^-1 * z^-1 * y^-1,
x^-1 * z^-1 * y^-1 * x * y * z,  z * x^-1 * z^-1 * y^-1 * x * y,
z^-1 * y^-1 * x * y * z * x^-1,  y * z * x^-1 * z^-1 * y^-1 * x }
```

78.3 Finitely Generated Subgroups of Free Groups

A number of functions are available that apply specifically to finitely generated subgroups of free groups which, by the Nielsen–Schreier theorem, are themselves free. For example, it is possible to determine the index of one such subgroup in another, which may be infinite, and computation of intersections and normalisers of subgroups, and conjugacy testing are all possible. The relevant algorithms are described in [Sim94, Chapter 4], for example.

Supergroup(F): GrpFP -> GrpFP

The largest (with respect to containment) **free group** of which F is known to be a subgroup. The remaining functions will fail and exit with errors if their arguments do not have a common supergroup.

x **in** H: GrpFP, GrpFPElt -> BoolElt

Returns `true` in $x \in H$, where x is required to be an element of **Supergroup**(F).

IsSubgroup(H, K): GrpFP, GrpFP -> BoolElt

Given groups H and K, subgroups of some common overgroup, return `true` if H is a subgroup of K, and `false` otherwise.

H **eq** K: GrpFP, GrpFP -> BoolElt

Determines whether the (free) subgroups H and K of a common free group are equal.

Index(F, H): GrpFP, GrpFP -> RngIntElt

The index, which may be ∞, in the free group F of subgroup H. An error will occur if H is not a subgroup of F.

HasFiniteIndex(F, H): GrpFP, GrpFP -> BoolElt

Returns **true** if the index in the free group F of subgroup H is finite. An error will occur if H is not a subgroup of F.

FreeGenerators(H): GrpFP -> SeqEnum, GrpFP

Returns a sequence of free generators of free group H and also the same group H defined on those generators.

Rewrite(H, x): GrpFP, GrpFPElt -> BoolElt, GrpFPElt

If x is an element of H, return **true** and the corresponding word in the generators of H; otherwise return **false**.

SubgroupElements(G, H, n): GrpFP, GrpFP, RngIntElt -> SetEnum

Given a subgroup H of a finitely presented group G, form the coset table for H in G, and use it to return the *words* in the generators of G of length at most n that represent elements of H.

H meet K: GrpFP, GrpFP -> GrpFP

Computes the intersection of free groups H and K.

Centraliser(F, x): GrpFP, GrpFPElt -> GrpFP

Centralizer(F, x): GrpFP, GrpFPElt -> GrpFP

Computes the centralizer in free group F of element x. An error will occur if x is not in F.

IsConjugate(F, x, y): GrpFP, GrpFPElt, GrpFPElt -> BoolElt, GrpFPElt

Determines whether elements x and y of free group F are conjugate in F and if so returns a conjugating element. An error will occur if either x or y is not in F.

Centraliser(F, H): GrpFP, GrpFP -> GrpFP

Centralizer(F, H): GrpFP, GrpFP -> GrpFP

Computes the centralizer in free group F of subgroup H. An error will occur if H is not a subgroup of F.

Normaliser(F, H): GrpFP, GrpFP -> GrpFP

Normalizer(F, H): GrpFP, GrpFP -> GrpFP

Computes the normalizer in free group F of subgroup H. An error will occur if H is not a subgroup of F.

IsConjugate(F, H, K): GrpFP, GrpFP, GrpFP -> BoolElt, GrpFPElt

Determines whether subgroups H and K of free group F are conjugate in F and if so returns a conjugating element. An error will occur if either H or K is not a subgroup of F.

Example: Subgroups of a Free Group (ex-f8ece0)

Starting with the free group of rank 3 some subgroup calculations will be demonstrated.

```
> F<a,b,c> := FreeGroup(3);
> F;
Finitely presented group F on 3 generators (free)

> H<x, y> := sub< F | a, b>;
> H;
Finitely presented group H on 2 generators
Generators as words in group F
    x = a
    y = b
> Index(F, H);
0
```

The next step is to produce a more interesting subgroup of F. To get one, the subgroup H will be conjugated by an element of F.

```
> K := H^(a*b*c);
> K;
Finitely presented group K on 2 generators
Generators as words in group F
    K.1 = c^-1 * b^-1 * a * b * c
    K.2 = c^-1 * b^-1 * a^-1 * b * a * b * c
> H eq K;
false
> H meet K;
Finitely presented group on 1 generator
Generators as words in group F
    $.1 = Id(F)

> N := Normalizer(F, K);
> N;
Finitely presented group N on 2 generators
Generators as words in group F
    N.1 = a^c
    N.2 = b^c
> K eq N;
true
```

The normaliser N of K has nice generators but it is equal to K so K is self-normalising.

78.4 The Automorphism Group of a Free Group

AutomorphismGroup(F): GrpFP -> GrpAuto

This intrinsic constructs the automorphism group of a free group F.

InverseAutomorphismFreeGroup(F, Q): GrpFP, SeqEnum -> GrpAutoElt

Given a free group F and a sequence Q of images of its generators, return the automorphism of F defined by Q.

WhiteheadReduction(F, Q): GrpFP, SeqEnum -> BoolElt, SeqEnum, GrpAutoElt

Perform Whitehead reduction of a sequence Q of elements of a free group F. Return true if the list Q is part of a free basis, the Whitehead reduction, and corresponding automorphism of F.

Example: Automorphism Group of a Free Group (ex-baab12)

We find the automorphism group of the free group F of rank 3.

```
> F := FreeGroup(3);
> A, B := AutomorphismGroup(F);
> A;
A group of automorphisms of GrpFP: F
Generators:
    Automorphism of GrpFP: F which maps:
        F.1 |--> F.2
        F.2 |--> F.1
        F.3 |--> F.3
    Automorphism of GrpFP: F which maps:
        F.1 |--> F.2
        F.2 |--> F.3
        F.3 |--> F.1
    Automorphism of GrpFP: F which maps:
        F.1 |--> F.1^-1
        F.2 |--> F.2
```

(continues on next page)

(continued from previous page)

```
        F.3 |--> F.3
    Automorphism of GrpFP: F which maps:
        F.1 |--> F.1 * F.2
        F.2 |--> F.2
        F.3 |--> F.3
> B;
Finitely presented group B on 4 generators
Relations
    B.1^2 = Id(B)
    B.2^3 = Id(B)
    (B.2 * B.1)^2 = Id(B)
    B.3^2 = Id(B)
    B.3^-1 * B.1^-1 * B.2^-1 * B.3 * B.2 * B.1 = Id(B)
    B.3^-1 * B.2^-1 * B.3^-1 * B.2 * B.3 * B.2^-1 * B.3 * B.2 = Id(B)
    B.3^-1 * B.2^-1 * B.1^-1 * B.2 * B.3 * B.2^-1 * B.1 * B.2 = Id(B)
    B.4^-1 * B.3^-1 * B.4^-1 * B.3^-1 * B.4 * B.3 * B.4 * B.3 = Id(B)
    B.4^-1 * B.1^-1 * B.2^-1 * B.3^-1 * B.4^-1 * B.3^-1 * B.2 * B.1^-1 * B.4 *
    B.1 * B.2^-1 * B.3 * B.4 * B.3 * B.2 * B.1 = Id(B)
    (B.1 * B.3 * B.1 * B.4)^2 = Id(B)
    B.4^-1 * B.1 * B.4 * B.1 * B.3 * B.4 * B.3 * B.1 * B.3 = Id(B)
    B.4^-1 * B.2^-1 * B.1^-1 * B.2 * B.1^-1 * B.2^-1 * B.4 * B.2 * B.1 * B.2^-1
    * B.1 * B.2 = Id(B)
    B.4^-1 * B.2^-2 * B.3^-1 * B.2^2 * B.4 * B.2^-2 * B.3 * B.2^2 = Id(B)
    B.4^-1 * B.1^-1 * B.2^-1 * B.1^-1 * B.2 * B.1^-1 * B.4^-1 * B.1^-1 * B.2^-1
    * B.1^-1 * B.2 * B.1^-1 * B.4 * B.1 * B.2^-1 * B.1 * B.2 * B.1 * B.4 * B.1 *
    B.2^-1 * B.1 * B.2 * B.1 = Id(B)
    B.1 * B.2^-1 * B.4 * B.2 * B.1 * B.2^-1 * B.4 * B.2 * B.4 * B.2^-1 * B.4^-1
    * B.2 * B.4^-1 = Id(B)
```

The first return value is the automomorphism group given as a group of mappings. The second return value is the automorphism group as an fp-group. A different type of group is needed in order to return the presentation of the automomorphism group.

Example: Automorphism Group of a Free Subgroup `(ex-1a5108)`

In this example the automorphism group of a subgroup of the free group F of rank 3 is found.

```
> F<a, b, c> := FreeGroup(3);
> H<x, y> := sub< F | a, b>;
> K := H^(a*b*c);
> K;
Finitely presented group K on 2 generators
Generators as words in group F
    K.1 = c^-1 * b^-1 * a * b * c
    K.2 = c^-1 * b^-1 * a^-1 * b * a * b * c
> A, B := AutomorphismGroup(K);
> A;
A group of automorphisms of GrpFP: K
Generators:
    Automorphism of GrpFP: K which maps:
        K.1 |--> K.2
        K.2 |--> K.1
    Automorphism of GrpFP: K which maps:
        K.1 |--> K.1^-1
        K.2 |--> K.2
    Automorphism of GrpFP: K which maps:
        K.1 |--> K.1 * K.2
```

(continues on next page)

(continued from previous page)

```
       K.2 |--> K.2
> B;
Finitely presented group B on 3 generators
Relations
    B.1^2 = Id(B)
    B.2^2 = Id(B)
    (B.2 * B.1)^4 = Id(B)
    B.3^-1 * B.2^-1 * B.3^-1 * B.2^-1 * B.3 * B.2 * B.3 * B.2 = Id(B)
    (B.1 * B.2 * B.1 * B.3)^2 = Id(B)
    (B.3 * B.1 * B.2)^3 = Id(B)
```

78.5 Bibliography

- [Sim94] Charles C. Sims. *Computation with Finitely Presented Groups.* Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

Chapter 79

Introduction to Fp-Groups

79.1 Introduction

79.1.1 Overview

Finitely presented groups (henceforth referred to as fp-groups) arise in many branches of mathematics and computation plays a significant role in their study. Unfortunately, answering most basic questions about a given fp-group is often computationally difficult. Over the past 50 years a range of computational tools for investigating fp-groups have been developed and most of these are installed in Magma. Some are straightforward to use but others are complicated.

The *word problem* for fp-groups asks whether there is an algorithm which given any element of an fp-group can decide whether or not it is the identity. In 1955, Novikov showed that there are specific fp-groups for which the word problem is insoluble. Further results show that other decision problems for fp-groups are also insoluble. Consequently, this limits what we can compute for fp-groups. The algorithms (or procedures) that have been developed can provide some information about many fp-groups but there will be many more groups for which they fail.

The Handbook documentation describing the fp-group facilities runs to more than 200 pages. In order to make the material more accessible this chapter is a terse introduction to the more important tools. Less important intrinsics are omitted and advanced features of those intrinsics that are discussed are often also omitted. The hope is that new users will be able to quickly get enough information from this chapter to be able to make straightforward application of the more important intrinsics and then dip into the detailed expositions in chapter *Finitely Presented Groups* when they become more proficient.

This chapter is concerned with groups defined by general presentations. Before starting this chapter the user should read the (very short) chapter *Free Groups* on free groups which is assumed knowledge for this chapter. For groups defined by power-conjugate presentations the user should consult chapters Finite Soluble Groups and *Polycyclic Groups*. For users interested in automatic and hyperbolic groups most of the information concerning them can be found in chapter *Automatic and Hyperbolic Groups*. The complete list of intrinsics for fp-groups together with their advanced features can be found in chapter *Finitely Presented Groups*.

The intrinsics considered here are designed for doing what is sometimes referred to as *combinatorial group theory*. A description of the fundamental algorithms for finitely presented groups can be found in Sims [Sim94] or Holt [HEO05]. The facilities provided for fp-groups in Magma fall into the following categories:-

- Construction of presentations for fp-groups and their simplification.
- Determination of various properties such as non-trivial, finite/infinite, perfect, small cancellation, large, automatic, and hyperbolic.
- Calculations with subgroups of finite index including the enumeration of all subgroups having index less a specified bound.

- Construction of particular types of quotient group: abelian quotient, p-quotient, nilpotent quotient, soluble quotient, and simple group quotients.

The corresponding Magma categories are `GrpFP` for fp-groups and `GrpFPElt` for their elements.

79.1.2 Definitions and Notation

Let X be a finite set of symbols, called generators in this context. Strings can be formed from X by concatenating the symbols x and x^{-1} for $x \in X$. Defining multiplication $(*)$ of strings over X as concatenation and forcing this multiplication rule to satisfy the group axioms puts a group structure on strings over X. In this group the identity is the empty string and the elements are called *words*. A group of this type is called the *free group* of rank n, where n is the cardinality of X. These groups are discussed in detail in chapter *Free Groups*.

A *relation* is an equality between two words while a *relator* consists of a word that equals the identity, that is, it is a relation where the right hand side is understood to be the identity. If a set R of non-trivial relations are imposed on F its elements are partitioned into equivalence classes. This set of equivalence classes form a new group. If X and R are both finite the group they define is called a *finitely-presented group* (abbreviated to fp-group) and is the subject of this chapter. The pair $\langle X|R\rangle$ is called a *presentation* for G. Note that any free group of finite rank is also an fp-group where the set R is empty.

The set of relations R can be converted into a set of relators and the relator words generate a subgroup H of F. If N is the normal closure of H in F then the group G is the quotient group F/N. This observation is the basis for the way in which fp-groups are constructed in Magma. The free group of rank n is constructed by the intrinsic `FreeGroup`(n). Having constructed an appropriate free group F, an fp-group G is constructed by specifying relations on the generators of F that must be satisfied by the corresponding generators of G. The details of how this is achieved are discussed in the following sections.

79.2 Construction of FP-Groups

An fp-group is normally either constructed by giving a presentation in terms of generators and relations or it is defined to be a subgroup or quotient group of an existing fp-group. However, fp-groups may be also created from finite permutation and matrix groups. Additionally, Magma has separate types for a number of families of fp-groups such as soluble groups which can be defined by power-conjugate presentations thus providing a solution to the word problem. Functions are provided to convert a member of one of these families into a general fp-group.

A finitely presented group G is a quotient group of some free group having finite rank. The free group of rank n is constructed by the intrinsic `FreeGroup`(n). Having constructed an appropriate free group F, an fp-group G is constructed by specifying relations on the generators of F that must be satisfied by the corresponding generators of G. The details of how this is achieved are discussed in the following sections.

79.2.1 Relations

A *relation* is an equality between two words in an fp-group while a *relator* consists of a word that equals the identity, that is, it is a relation where the right hand side is understood to be the identity. A *presentation* is a set of symbols called generators together with a set of relations on those generators. To facilitate working with relations, a *relation type* is provided so that relations can be treated in a similar way to any other object in Magma. Below are the three operations that are specific to relations.

Intrinsics:

`w_1 = w_2` : Given words w_1 and w_2 belonging to fp-group G create the relation $w_1 = w_2$.

`LHS`(r) : Given a relation r over the generators of G, return the left hand side of the relation r as a word in G.

`RHS`(r) : Given a relation r over the generators of G, return the right hand side of the relation r as a word in G.

79.2.2 Quotient Group Constructors

A group with non-trivial relations is constructed as a quotient of an existing group, usually a free group. For convenience, the necessary free group may be constructed in-line.

Constructors:

quo$\langle\ F \mid R\ \rangle$: Given an fp-group F, and a set R of relations and/or relators in the generators of F, construct the quotient G of F by the normal subgroup of F defined by R. The group G is defined by means of a presentation which consists of the relations for F (if any), together with the additional relations defined by the list R. The constructor returns the quotient group G and the natural homomorphism $\phi : F \to G$.

G / H : Given a subgroup H of the fp-group G, construct the quotient of G by the normal closure N of H. The quotient is formed by taking the presentation for G and including the generating words of H as additional relators.

Notes:

(i) If explicit names are not available, a standard indexing notation is provided for referencing the generators of any group G. Thus, `G.i` denotes the i-th generator of G.

(ii) Individual names may be given to the generators by means of the *generator-assignment* syntax. Suppose that the group G is defined on r generators. Then if E is an expression creating G the special assignment

$G\ \langle\ v_1,\ v_2,\ \ldots,\ v_r\ \rangle\ := E;$

is equivalent to the statements

$G := E;\ v_1 := G.1;\ v_2 := G.2;\ \ldots;\ v_r := G.r;$

Example: Defining S4 as a Quotient Group (ex-755b4c)

Three variations of the constructor are used to define the fp-group G isomorphic to the symmetric group of degree 4 with presentation $< a, b \mid a^2, b^3, (ab)^4 >$. Note that as F (the free group) and G (the S_4-quotient) are two distinct groups it is necessary to have different names for the generators of each. The relations given in the `quo` expression are relations on the free group generators.

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi := quo< F | a^2, b^3, (a*b)^4 >;
Finitely presented group G on 2 generators
Relations
    x^2 = Id(G)
    y^3 = Id(G)
    (x * y)^4 = Id(G)
```

In the next example, the relations are given as a relations list.

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi :=  quo< F | a^2 = b^3 = (a*b)^4 = 1>;
> G;
Finitely presented group G on 2 generators
Relations
    x^2 = Id(G)
    y^3 = Id(G)
    (x * y)^4 = Id(G)
```

Another alternative is to provide a set of relators:

```
> F<a, b> := FreeGroup(2);
> rels := { a^2, b^3, (a*b)^4 };
> G<x, y>, phi :=  quo< F | rels >;
> G;
```

(continues on next page)

(continued from previous page)

```
Finitely presented group G on 2 generators
Relations
    x^2 = Id(G)
    (x * y)^4 = Id(G)
    y^3 = Id(G)
```

The relations printed out are in a different order than in the set `rels` because sets are not ordered. If it is desired to have the relations appear in some particular order then a sequence should be used.

Example: Quotient Constructor without a Free Group (ex-17dd50)

A group may be defined using the quo-constructor without first assigning a free group. The $ symbol is used to reference a group whose quotient is being formed. Recalling the indexing notation for generators, the generic names $.1, $.2, etc can be used in this situation.

```
> S4<x, y> := quo< FreeGroup(2) | $.1^2, $.2^3, ($.1*$.2)^4 >;
> S4;
Finitely presented group S4 on 2 generators
Relations
      x^2 = Id(S4)
      y^3 = Id(S4)
      (x * y)^4 = Id(S4)
```

79.2.3 FP-Group constructor

For convenience, a constructor is provided which allows the user to define an fp-group without explicitly constructing a free group.

Intrinsics:

`FPGroup`$\langle\ X \mid R\ \rangle$: Given a list X of variables $x_1, \ldots, x_r$, and a list of relations R over these generators, first construct the free group F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the normal subgroup of F defined by the relations R. The syntax for the relations R is the same as for the quo-constructor. The intrinsic returns the quotient group G and the natural homomorphism $\phi : F \to G$.

Example: Binary Tetrahedral Group (ex-20a432)

The `FPGroup`-constructor is used to create the binary tetrahedral group using the presentation $< r, s \mid r^3 = s^3 = (rs)^2 >$:

```
> G<r, s> := FPGroup< r, s | r^3 = s^3 = (r*s)^2 >;
> G;
Finitely presented group G on 2 generators
Relations
    r^3 = (r * s)^2
    s^3 = (r * s)^2
```

Again, using the `FPGroup`-constructor, the group $< r, s, t \mid r^2, s^2, t^2, rst = str = trs >$ would be specified as:

```
> G<r, s, t> := FPGroup<r, s, t | r^2, s^2, t^2, r*s*t = s*t*r = t*r*s>;
> G;
Finitely presented group G on 3 generators
Relations
    r^2 = Id(G)
    s^2 = Id(G)
    t^2 = Id(G)
```

(continues on next page)

(continued from previous page)

```
    r * s * t = t * r * s
    s * t * r = t * r * s
```

Example: Quotient of an FP-Group (ex-a5e77e)

As shown in this example, the *quo< F | R >*-constructor is not restricted to defining the quotient of a free group.

```
> F<x, y> := FPGroup< x, y | x^2 = y^3 = (x*y)^7 = 1 >;
> F;
Finitely presented group F on 2 generators
Relations
      x^2 = Id(F)
      y^3 = Id(F)
      (x * y)^7 = Id(F)
> G<a, b> := quo< F | (x, y)^8 >;
> G;
Finitely presented group G on 2 generators
Relations
      a^2 = Id(G)
      b^3 = Id(G)
      (a * b)^7 = Id(G)
      (a, b)^8 = Id(G)
> #G;
10752
```

Example: Coxeter's Parametrised Family of Groups (ex-17488a)

In this example Magma functions are used to represent parametrised families of groups. In the notation of Coxeter, the symbol $(l, m|n, k)$ denotes the family of groups having presentation

$$< a, b \mid a^l, b^m, (a * b)^n, (a * b^{-1})^k > .$$

```
> Glmnk := func< l, m, n, k | FPGroup< a, b | a^l, b^m, (a*b)^n, (a*b^-1)^k > >;
> G<a, b> := Glmnk(3, 3, 4, 4);
> G;
Finitely presented group G on 2 generators
Relations
      a^3 = Id(G)
      b^3 = Id(G)
      (a * b)^4 = Id(G)
      (a * b^-1)^4 = Id(G)
> Order(G);
168
> G<a, b> := Glmnk(2, 3, 4, 5);
> G;
Finitely presented group G on 2 generators
Relations
    a^2 = Id(G)
    b^3 = Id(G)
    (a * b)^4 = Id(G)
    (a * b^-1)^5 = Id(G)
> Order(G);
1
```

Thus $(2, 3 \mid 4, 5)$ is the trivial group.

79.2.4 Presentation Operations

Intrinsics:

`AddGenerator` (G) : Add a new generator to the presentation defining G.

`AddGenerator` $(G,\ w)$: Add a new generator z and the relation $z = w$ to the presentation defining G.

`AddRelation` $(G,\ r)$: Add the relation r to the presentation defining G.

`AddRelation` $(G,\ r,\ i)$: Add the relation r after the i-th relation of the presentation defining G.

`DeleteGenerator` $(G,\ x)$: Delete the generator x and all relations involving x from the presentation defining G.

`ReplaceRelation` $(G,\ s,\ r)$: Here each of r and s may be a relation or a word. The relation r must be part of the presentation for G. The relation r is replaced by the relation s in the presentation defining G.

`DeleteRelation` $(G,\ r)$: Delete the relation r from the presentation defining group G.

`Generators` (G) : The set of generators for G.

`NumberOfGenerators` (G) : The number of generators used to define G.

`PresentationLength` (G): The length of the presentation defining G. The length is defined to be the sum of the lengths of the words defining each relation or relator.

`Simplify`(G) : A simplified presentation for G is found by using Tietze transformations and substring searching. A new group K defined by the simplified presentation is returned together with the isomorphism $f : G \to K$.

`SimplifyLength`(G) : This has the same effect as `Simplify` except that the simplification process terminates when the presentation length starts to increase. While it may be possible to eliminate further generators it is likely that this will significantly lengthen the simplified presentation.

Notes:

(i) The intrinsic `Simplify` works by repeatedly eliminating generators and subsequently shortening relators by substitution of substrings that correspond to the left or right hand side of a relation. The transformation process terminates when no more eliminations of generators and no more length reducing substring replacements are possible. The choice of simplification strategy can either be left to Magma or selected by the user, The simplification process can be controlled by a set of parameters described in chapter *Finitely Presented Groups*.

Example: Simplifying the Fibonacci Group F(8) `(ex-c32ec7)`

Consider the Fibonacci group $F(8)$.

```
> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>     quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>              x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;
```

The intrinsic `Simplify` will be applied to obtain a presentation of $F(8)$ on fewer generators. It is returned as the presentation for a new group H. As the number of generators used in the simplified presentation for H is unknown it is necessary to use a variation of the generator assignment expression: `H<[y]>`. This will give the generators of H the names `y[1], y[2], ...`. Here y is any variable name.

```
> H<[y]>, f := Simplify(F8);
> H;
Finitely presented group H on 2 generators
Generators as words in group F8
    y[1] = x3
    y[2] = x4
```

(continues on next page)

(continued from previous page)

```
Relations
    y[2] * y[1]^-2 * y[2] * y[1]^-1 * y[2]^2 * y[1] * y[2]^2 *
       y[1]^-1 = Id(H)
    y[1] * y[2] * y[1] * y[2]^2 * y[1] * y[2] * y[1]^2 * y[2]^-1
       * y[1] = Id(H)
```

The isomorphism f can be used to express the old generators in terms of the new ones.

```
> f;
Mapping from: GrpFP: F8 to GrpFP: H
> f(x1);
y[1]^2 * y[2]^-1
```

79.2.5 Operations on Words

The words u and v in the intrinsics below must belong to a common fp-group G.

Intrinsics:

$u * v$: The product of words u and v.

u^{-1} : The inverse of word u.

`Identity` (G) : The identity element for group G. Other notations are `Id` (G) and $G!1$.

u^n : The n-th power of u.

u^v : The conjugate $v^{-1} * u * v$ of words u and v.

(u, v) : The commutator $u^{-1} * v^{-1} * u * v$ of words u and v.

u `eq` v : Returns `true` if the words u and v are identical when freely reduced. NB When G is not a free group and `false` is returned, this does not imply that u and v do not represent the same element of G.

u `ne` v : Returns `true` if the words u and v are distinct when freely reduced. NB When G is not a free group and `true` is returned, this does not imply that u and v do not represent the same element of G.

79.2.6 Presentations for Popular Groups

Intrinsics are provided for constructing presentations for a number of familiar groups.

Intrinsics:

`AbelianFPGroup` (Q) : The abelian group defined by the sequence of integers Q.

`CyclicFPGroup` (n) : The cyclic group of order n.

`AlternatingFPGroup` (n) : The alternating group of degree n on two generators.

`BraidFPGroup` (n) : The braid group on n strings on $n-1$ Artin generators.

`DihedralFPGroup` (n) : For $n > 2$, the dihedral group of order $2n$ on two generators.

`SymmetricFPGroup`(n) : The symmetric group of degree n on two generators.

Example: Abelian Group as an FP-Group `(ex-318133)`

An abelian group is presented as a direct product of cyclic groups. This direct product is defined by a sequence of integers giving the orders of the cyclic factors, where 0 indicates $\mathbb{Z}$. To illustrate this the code below constructs the abelian group $\mathbb{Z}_4 \times \mathbb{Z}_9 \times \mathbb{Z}$.

```
> A<a, b, c> := AbelianFPGroup([4, 9, 0]);
> A;
Finitely presented group A on 3 generators
Relations
```

(continues on next page)

(continued from previous page)

```
    a^4 = Id(A)
    b^9 = Id(A)
    a * b = b * a
    a * c = c * a
    b * c = c * b
```

Example: Symmetric Group Sym(8) as an FP-Group (ex-169352)

In this example the symmetric group Sym(8) is constructed as an fp-group:-

```
> S8<x, y> := SymmetricFPGroup(8);
> S8;
Finitely presented group S8 on 2 generators
Relations
    x^8 = Id(S8)
    y^2 = Id(S8)
    (x * y)^7 = Id(S8)
    (x^-1 * y * x * y)^3 = Id(S8)
    (y * x^-2 * y * x^2)^2 = Id(S8)
    (y * x^-3 * y * x^3)^2 = Id(S8)
    (y * x^-4 * y * x^4)^2 = Id(S8)
```

79.2.7 Constructions for FP-Groups

Intrinsics are provided to construct products of fp-groups and also to transform groups defined by various types of polycyclic presentations into fp-groups. In addition there are intrinsics which construct a presentation for a finite permutation or matrix group G and return a fp-group isomorphic to G.

Intrinsics:

`DirectProduct` (G, H) : The direct product of the fp-groups G and H.

`FreeProduct` (G, H) : The free product of the fp-groups G and H.

`FPGroup` (T) : If T is a finite nilpotent or soluble group defined by a polycyclic presentation return an fp-group G isomorphic to T together with the isomorphism $\phi : T \to G$.

`FPGroup` (T) : If T is a finite permutation or matrix group of moderate size then return an fp-group G isomorphic to T together with the isomorphism $\phi : G \to T$.

`FPGroupStrong` (T) : If T is a finite permutation or matrix group which may have order much greater than a million return an fp-group G isomorphic to T. While `FPGroup` returns a presentation on the permutation/matrix generators, `FPGroupStrong` constructs a presentation on a larger set of generators (*strong generators* for T). An isomorphism $\phi : G \to T$ is also returned.

Notes:

(i) Note that the intrinsic `FPGroup` applied to permutation or matrix groups may need to construct the coset table of G over the identity and hence is limited to groups of order less than a million. The exact limit is determined by the amount of memory available.

Example: Direct and Free Products of FP-Groups (ex-7dc82f)

The direct product and free product of the alternating group of degree 7 and the cyclic group of order 2 are created as follows:-

```
> A7<a, b> := AlternatingFPGroup(7);
> A7;
Finitely presented group A7 on 2 generators
Relations
    a^5 = Id(A7)
    b^3 = Id(A7)
```

(continues on next page)

(continued from previous page)

```
    (a * b)^7 = Id(A7)
    (b * a^-1 * b * a)^2 = Id(A7)
    (b * a^-2 * b * a^2)^2 = Id(A7)
> Z2<c> := CyclicFPGroup(2);
Finitely presented group Z2 on 1 generator
Relations
    c^2 = Id(Z2)
> G<x, y, z> := DirectProduct(A7, Z2);
> G;
Finitely presented group G on 3 generators
Relations
    x^5 = Id(G)
    y^3 = Id(G)
    (x * y)^7 = Id(G)
    (y * x^-1 * y * x)^2 = Id(G)
    (y * x^-2 * y * x^2)^2 = Id(G)
    z^2 = Id(G)
    x * z = z * x
    y * z = z * y
> H<x, y, z> := FreeProduct(A7, Z2);
> H;
Finitely presented group H on 3 generators
Relations
    x^5 = Id(H)
    y^3 = Id(H)
    (x * y)^7 = Id(H)
    (y * x^-1 * y * x)^2 = Id(H)
    (y * x^-2 * y * x^2)^2 = Id(H)
    z^2 = Id(H)
```

Note that the free product is always an infinite group.

Example: FP-Group Isomorphic to GL(3,4) `(ex-c105bf)`

In this example, an fp-group G that is isomorphic to $\mathrm{GL}(3,4)$ of order 181440 is created.

```
> k<u> := GF(4);
> T := GL(3, k); #T;
181440
> T;
MatrixGroup(3, GF(2^2)) of order 2^6 * 3^4 * 5 * 7
Generators:
    [    u     0     0]
    [    0     1     0]
    [    0     0     1]

    [    1     0     1]
    [    1     0     0]
    [    0     1     0]

> G<x, y> := FPGroup(T);
> G;
Finitely presented group G on 2 generators
Relations
      x^3 = Id(G)
      y^7 = Id(G)
      (x, y)^3 = Id(G)
```

(continues on next page)

(continued from previous page)

```
(x*y^-1*x^-1*y^2)^3 = Id(G)
x^-1*y*x^-1*y^-1*x^-1*y*x^-1*y^-1*x*y*x*y^-1*x*y*x*y^-1 = Id(G)
y^-1*x^-1*y*x^-1*y^-1*x*y^-1*x^-1*y^-1*x*y*x*y^-1*x*y^3*x^-1 = Id(G)
(y^-1*x^-1*y^-1*x*y^-2)^3 = Id(G)
```

79.3 Properties of an FP-group

Most of the group properties discussed in this section are usually hard to determine. Consequently, many of the intrinsics that test an fp-group G for a given property P only return a useful answer in those cases for which they are able to prove that G has property P. If they cannot do this then no conclusion can be drawn. The group property intrinsics that are available in Magma are listed below:-

Intrinsics:

`IsPerfect` (G) : Returns *true* if G is perfect and *false* if not. Note that this and the small cancellation intrinsic are the only ones in this list that can definitively decide whether or not the condition holds.

`IsFinite` (G) : Returns *true* if Magma is able to prove that G has finite order. Otherwise no conclusion can be drawn. If successful the order of G is also returned.

`IsInfinite` (G) : Returns *true* if Magma is able to prove that G is infinite. Otherwise no conclusion can be drawn.

`IsLarge` (G, l, u) : Attempts to show that G has a subgroup with homomorphic image the free group of rank 2. If successful, `true` is returned together with a subgroup witness to this property. If not successful, return `false`. The algorithm will consider subgroups of G with index h such that $l \leq h \leq u$.

`SmallCancellationConditions` (G) : Returns the small cancellation conditions $T(i)$, $C(j)$, and $C'(k)$ that are satisfied by the presentation for G. Specifically it returns $i,\ j,\ k'$, where $T(i)$, $C(j)$, and $C'(k)$ are satisfied for all $k > k'$.

`IsAutomaticGroup` (G) : Returns *true* if Magma is able to prove that G is an automatic group. Otherwise no conclusion can be drawn. If successful the automatic structure for G is also returned.

`IsHyperbolic` (G) : Returns *true* if Magma is able to prove that G is a hyperbolic group. Otherwise no conclusion can be drawn. If successful a Dehn function for the group may be returned.

Notes:

(i) There are two basic approaches for computing structural information about an fp-group G: finding subgroups of finite index in G where the index is moderate in size and finding quotients of G. These approaches are the basis for determining the first four properties above.

(ii) Intrinsic `IsAutomatic` uses the Knuth–Bendix method for constructing a confluent presentation. The intrinsic works best for groups G having two or three generators and short relators. Even so the running time can be substantial. Given the automatic structure for G, the word problem for G can be solved, whether or not G is finite or infinite can be easily determined, and a growth function for G can be computed. See Chapter *Automatic and Hyperbolic Groups* for detailed information and examples.

(iii) The algorithm underpinning the intrinsic `IsHyperbolic` is based on small cancellation theory. Care has to be taken in setting up the presentation and the user should read the relevant section in chapter *Automatic and Hyperbolic Groups* before attempting to use the intrinsic. As distinct from many of the other intrinsics described in this section, `IsHyperbolic` reaches a decision quite quickly. Preparing a presentation for input to this intrinsic is a challenge and the reader should consult the section on hyperbolic groups in chapter *Automatic and Hyperbolic Groups* for detailed information and examples.

Example: Proving a Coxeter Group Infinite `(ex-5d2db4)`

Some of the above intrinsics are applied to the Coxeter group

$$G = \langle\ x,\ y \mid x^2,\ y^3,\ (x*y)^7,\ (x,y)^9\ \rangle$$

First a proof that the group is infinite is produced:-

```
> G<x,y> := FPGroup< x, y |  x^2, y^3, (x * y)^7, (x, y)^9 >;
> yes, proof := IsInfinite(G);
> yes;
true
```

So the group is infinite. The second return argument outlines a proof of this fact.

```
> proof;

! The following lines sketch a proof that the fp-group F is infinite:-
! F has the group PSL(2, 8) of order 504 as a quotient G (L2Quotient).
! Let k denote the rational field.
! Let M be the ext square of perm module of G as a kG-module. It has dim 36.
! Let N be the kF-module given by the induced action of F on M.
! The dimension of the first integral cohomology group of the kF-module N is 1.
! As H^1(F, N) is non-zero, the Holt-Plesken criterion shows that F is infinite.
```

Next the group is tested for being perfect.

```
> IsPerfect(G);
true
```

So G is perfect. Next a subgroup of finite index in G is defined and tested for being perfect.

```
> H := sub< G |  x, y * x * y * x * y * x * y^-1 >;
> Index(G, H);
9
> IsPerfect(H);
false
> AbelianQuotient(H);
Abelian Group isomorphic to Z/7
Defined on 1 generator
Relations:
    7*$.1 = 0
```

So the subgroup H is not perfect and its derived subgroup has index 7 in H. Note that if H had infinite or very large index in G then the `IsPerfect` test would fail.

The final step is to determine the small cancellation conditions satisfied by G.

```
> a := SmallCancellationConditions(G);
This presentation satisfies T( 3 ).
This presentation satisfies C( 2 ).
This presentation satisfies C'(k) for all k > 1/2.
```

Example: Automatic Structure of the Infinite Dihedral Group (ex-52e6c5)

The infinite dihedral group is defined and then its automatic structure is computed.

```
> G<a, b> := FPGroup< a, b | a^2, b^a = b^-1 >;
> G;
Finitely presented group G on 2 generators
Relations
    a^2 = Id(G)
    b^a = b^-1
> yes, A := IsAutomaticGroup(G);
> yes;
true
```

The second return value A is the group defined by the automatic structure of G. The growth function f for A is calculated. The fact that the group is infinite means that the growth function has to be a rational function rather than a polynomial. Consequently, it is necessary to compute the power series for the rational function f.

```
> R<x> := RationalFunctionField(Integers());
> f := GrowthFunction(A);
> f;
(-x^2 - 2*x - 1)/(x - 1)
```

This is a nice simple rational function. For a typical group the growth function has much larger degree numerator and denominator! The power series expansion of f is obtained by coercing f into the appropriate power series ring.

```
> PSR := PowerSeriesRing(Integers(): Precision := 15);
> s<t> := PSR!f;
> s;
1 + 3*t + 4*t^2 + 4*t^3 + 4*t^4 + 4*t^5 + 4*t^6 + 4*t^7 + 4*t^8 +
    4*t^9 + 4*t^10 + 4*t^11 + 4*t^12 + 4*t^13 + 4*t^14 + O(t^15)
```

So there are three elements of G of length 1 and four elements of length 2.

79.4 Subgroups

79.4.1 Subgroup Constructor

Constructor:

sub$\langle$ G | L $\rangle$: Construct the subgroup H of the fp-group G generated by the words specified by the terms of the *generator list* $L = L_1, \ldots, L_r$. A term L_i of the generator list may consist of a word, a set or sequence of words, a subgroup or a set or sequence of subgroups. The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

Notes:

(i) The generators of H consist of the words specified directly by terms L_i together with the stored generating words for any groups specified by terms of L_i. Repetitions of an element and occurrences of the identity element are removed (unless H is trivial). If the `sub`-constructor is invoked with an empty list L, the trivial subgroup will be constructed.

Example: Subgroup of Index 448 `(ex-7c2095)`

The group $(8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (ab^{-1})^3 >,$$

and has a subgroup of index 448 generated by the words a^2 and ab^{-1}:

```
> G<a, b> := FPGroup<a, b| a^8, b^7, (a * b)^2, (a * b^-1)^3>;
> G;
Finitely presented group G on 2 generators
Relations
     a^8 = Id(G)
     b^7 = Id(G)
     (a * b)^2 = Id(G)
     (a * b^-1)^3 = Id(G)
> H<x, y> := sub< G | a^2, a * b^-1 >;
> H;
Finitely presented group H on 2 generators
Generators as words in group G
```

(continues on next page)

(continued from previous page)

```
    x = a^2
    y = a * b^-1
```

79.4.2 Coset Enumeration

Intrinsics:

`Index` $(G,\ H)$: The index of H in G.

`CosetTable` $(G,\ H)$: The coset table for H in G (right action).

`Transversal` $(G,\ H)$: A (right) Schreier system of coset representatives for H in G together with the coset map.

`SchreierGenerators` $(G,\ H)$: Schreier generators for H as a set of words in G.

`ToddCoxeter` (G, H) : The Todd–Coxeter procedure for coset enumeration is applied to construct the coset table for the subgroup H of fp-group G. The convention here is that G acts on the right. If successful the index of H in G and the coset table are returned otherwise zero is returned for the index.

`SetGlobalTCParameters` (~: *parameters*) : Set the parameter values used for indirect invocations of the Todd-Coxeter coset enumeration procedure.

`UnsetGlobalTCParameters` (~) : Restore the default values for the parameters used for indirect invocations of the Todd–Coxeter coset enumeration procedure.

Notes:

(i) While strictly speaking the user does not need to invoke `ToddCoxeter` explicitly, it underpins many of the algorithms used to compute with subgroups in fp-groups and so a serious user needs to understand how to set the parameters in the case of computations that involve hard coset enumerations.

(ii) The `ToddCoxeter` intrinsic has a large number of parameters which provide the user with control over the enumeration. Descriptions of these may be found in chapter *Finitely Presented Groups*. Perhaps the most useful are `CosetLimit` which specifies the number of cosets which may be defined before abandoning an enumeration, `Time` which limits the maximum CPU time that can be expended, and `Strategy` which if set to ``Hard" chooses a strategy which has the best chance of succeeding with a hard enumeration.

(iii) Two intrinsics, `SetGlobalTCParameters` and `UnsetGlobalTCParameters`, allow a user to set the parameters for implicit calls to the Todd–Coxeter algorithm.

(iv) Descriptions of the Todd-Coxeter algorithm can be found in Chapter 5 of Holt [HEO05] and Chapter 5 of Sims [Sim94].

Example: Coset Table of an Index 15 Subgroup `(ex-476bb1)`

These intrinsics are illustrated using the infinite group G defined by

$$x^2 = y^3 = (xy)^7 = 1$$

and H a certain subgroup of G having index 15.

```
> G<x, y> := FPGroup< x, y | x^2, y^3, (x*y)^7 >;
> G;
Finitely presented group G on 2 generators
Relations
    x^2 = Id(G)
    y^3 = Id(G)
    (x * y)^7 = Id(G)
>
> H := sub< G | x, y*x*y*x*y^-1*x*y^-1, y^-1*x*y^-1*x*y*x*y>;
```

(continues on next page)

(continued from previous page)

```
> Index(G, H);
15
>
> CosetTable(G, H);
  Mapping from: Cartesian Product<{ 1 .. 15 }, GrpFP: G> to { 1 .. 15 }
       $1  $2 -$2
   1.   1   2   3
   2.   4   3   1
   3.   5   1   2
   4.   2   6   7
   5.   3   8   9
   6.   6   7   4
   7.  10   4   6
   8.  11   9   5
   9.   9   5   8
  10.   7  11  12
  11.   8  12  10
  12.  13  10  11
  13.  12  14  15
  14.  15  15  13
  15.  14  13  14
>
> T := Transversal(G, H);
> T;
{@ Id(G), y, y^-1, y*x, y^-1*x, y*x*y, y*x*y^-1, x^y, y^-1*x*y^-1,
   y*x*y^-1*x, y^-1*x*y*x, y*x*y^-1*x*y^-1, y*x*y^-1*x*y^-1*x,
   y*x*y^-1*x*y^-1*x*y, y*x*y^-1*x*y^-1*x*y^-1 @}
```

79.4.3 Coset Action

The columns of the coset table for a subgroup H in fp-group G are permutations giving the action of the generators of G on the cosets of H. If these columns are treated as permutation generators of a group P then the homomorphism $\phi : G \to P$ is an epimorphism of G. This quotient group and its kernel often provide useful information about the structure of G. In Magma the group G acts on the right of cosets and this should be assumed unless left action is explicitly stated.

Intrinsics:

`CosetAction` (G, H) : The permutation representation ϕ of G given by the action of G on the cosets of H in G. It returns ϕ and the image group $\phi(G)$. If the kernel of ϕ can be computed easily then it will also be returned.

`CosetImage` (G, H) : The image of the permutation action of G on the cosets of H.

`CosetKernel` (G, H) : The kernel of the permutation action of G on the cosets of H.

Example: Permutation Representation of G2(3) `(ex-6c4e10)`

The exceptional group $G_2(3)$ is a homomorph of the fp-group G defined below. First, a permutation representation $G1$ on 351 points is constructed for $G_2(3)$, and then the subgroup generated by the images of the first four generators of G in $G1$ is computed. (The permutation group intrinsics are described in chapter Permutation Groups.)

```
> F<a,b,c,d,y,x,w> := FreeGroup(7);
> z := y*c*a^2*b;
> u := x*d;
> t := w*c*a*d*b^2*c;
> G<a,b,c,d,y,x,w>, psi :=
>            quo< F | a^4, b^4, c^2, d^2, (a,b), (a*c)^2, (b*c)^2,
```

(continues on next page)

(continued from previous page)

```
>                         (c*d)^2, d*a*d*b^-1, y^3, (a^-1*b)^y*d*a^-1*b^-1,
>                         (c*d*a^-1*b)^y*b^-1*a*d*c, a*d*y*d*a^-1*y, x^3,
>                         a^x*b^-1, b^x*a*b, c^x*c, (x*d)^2, (u*z)^6, w^3,
>                         (w,y), (a*b)^w*b^-1*a*d*c, (c*d*a^-1*b)^w*d*c*b^2,
>                         (b^2*c*d)^w*a^-1*b^-1, (c*a^2*b*w)^2,
>                        (a^-1*b)^w*d*a^-1*b^-1, (t*u)^3 >;
> z1 := psi(z);
> u1 := psi(u);
> t1 := psi(t);

> H := sub< G | z1*a^2*b^2, u1*c, t1*a^2*b^2 >;
> phi, G1, K := CosetAction(G, H);
> Degree(G1);
351

> print Order(G1), FactoredOrder(G1);
4245696 [ <2, 6>, <3, 6>, <7, 1>, <13, 1> ]

> CompositionFactors(G1);
    G
    |  G(2, 3)
    1

> S := sub< G1 | phi(a), phi(b), phi(c), phi(d) >;
> BSGS(S);
> S;
Permutation group S of degree 351
Order = 64 = 2^6
```

Thus the images of a, b, c, and d in G1 generate the Sylow 2-subgroup of $G_2(3)$.

79.4.4 Enumeration of FI-Subgroups

If an fp-group G has subgroups of finite index and their index is moderate then they may be enumerated. There are two methods, the first of which enumerates all subgroups down to a given index while the second enumerates normal subgroups down to a given index.

Intrinsics:

`LowIndexSubgroups` (G, R) : Given a group G (possibly the free group), and an expression R defining a positive integer range (see below), determine the conjugacy classes of subgroups of G whose indices lie in the range specified by R. The expression R is either an integer n which indicates the range $[1..n]$ or a tuple $\langle m,\ n\rangle$ which represents the range $[m..n]$.

`LowIndexNormalSubgroups`$(G,\ n)$: The normal subgroups of G having index in the range $[1..n]$ are enumerated. The subgroups are returned as a sequence of records (ordered by subgroup index). The upper bound for n is 10^{10}.

Notes:

(i) The `LowIndexSubgroups` intrinsic has many applications and in favourable situations it can locate subgroups of index several hundred. The intrinsic also has parameters which can be found in chapter *Finitely Presented Groups*. Two useful parameters are:

- `Limit` which accepts a positive integer giving a limit on the number of subgroups satisfying expression R that are sought.

- `Print` which accepts an integer in the range $[0\ \ldots\ 3]$ which controls verbose output. Thus 0 specifies no verbose output while the integers greater than 0 turn on increasing amounts of verbose output.

(ii) Another version of the `LowIndexSubgroups` intrinsic implements what is called a **process**. This version can be restarted and returns each subgroup immediately it is found. If more subgroups are needed then the computation is restarted. This is particularly useful in that a subgroup satisfying the user's requirements may be found long before all the subgroups specified by R are computed.

(iii) The algorithm used to find subgroups of small index is described in Section 5.6 of Sims [Sim94] and Section 5.4 of Holt [HEO05].

Example: Tutte's 8-Cage and the Conder Graph (ex-c69596)

(Peter Lorimer) The two graphs known as Tutte's 8-cage and the Conder graph may be constructed as the Cayley graphs of two conjugacy classes of subgroups having index 10 in the finitely presented group

$$< p, q, r, s, h, a \mid h^3 = a^2 = p^2 = 1, p^h = p, p^a = q, q^h = r, r^a = s,$$
$$h^s = h^{-1}, r^h = pqr, sr = pqrs, pq = qp, pr = rp, ps = sp, qr = rq, qs = sq > .$$

The low index intrinsic will be used to construct these subgroups.

```
> G<p, q, r, s, h, a> := FPGroup<p, q, r, s, h, a |
>                         h^3 = a^2 = p^2 = 1, p^h = p, p^a = q,
>                         q^h = r, r^a = s, h^s = h^-1, r^h = p * q * r,
>                         s * r = p * q * r * s, p * q = q * p,
>                         p * r = r * p, p * s = s * p, q * r = r * q,
>                         q * s = s * q>;
> LowIndexSubgroups(G, <10, 10>);
[
       Finitely presented group on 6 generators
       Index in group G is 10 = 2 * 5
       Generators as words in group G
              $.1 = p
              $.2 = s
              $.3 = h
              $.4 = q^-2
              $.5 = a * h^-1 * a * r^-1
              $.6 = a * h * a * h^-1 * a * q^-1,

       Finitely presented group on 6 generators
       Index in group G is 10 = 2 * 5
       Generators as words in group G
              $.1 = p
              $.2 = s
              $.3 = h
              $.4 = q^-2
              $.5 = a * h^-1 * a * r^-1
              $.6 = a * h * a * h^-1 * a
]
```

Example: Conjugacy Classes via Low Index Subgroups (ex-c3d6e8)

A fairly surprising application for the low index subgroup algorithm is the enumeration of the conjugacy classes of subgroups of a finite fp-group. Consider the group

$$\mathrm{PGL}_2(9) \simeq G = \langle a, b | a^2,\ b^3,\ (ab)^8,\ [a, b]^5,\ [a, (ba)^3 b^{-1}]^2 \rangle.$$

```
> G<a,b> := FPGroup< a,b | a^2, b^3, (a*b)^8, (a,b)^5,
>                         (a,(b*a)^3*b^-1)^2  >;
> Order(G);
720
```

In an infinite fp-group, finding all conjugacy classes of subgroups up to index 720 using the low index subgroup algorithm would be extremely difficult. In the case of a fp-group G of order 720, however, it is not so difficult.

```
> time sgG := LowIndexSubgroups(G, Order(G));
Time: 31.859
> #sgG;
26
```

A list of representatives of the 26 conjugacy classes of subgroups of G have been found. For every representative, its index in G and a set of generating words are stored. Here are two examples:-

```
> sgG[10];
Finitely presented group on 2 generators
Index in group G is 60 = 2^2 * 3 * 5
Generators as words in group G
    $.1 = b
    $.2 = a * b * a * b * a * b^-1 * a * b * a * b * a * b^-1 * a
       * b^-1 * a
> sgG[21];
Finitely presented group on 1 generator
Index in group G is 180 = 2^2 * 3^2 * 5
Generators as words in group G
    $.1 = (b * a)^2
```

79.4.5 Operations for FI-Subgroups

Operations on subgroups that construct a new subgroup or quotient group are listed. Here G will be a finitely presented group with a presentation while H and K will be subgroups having finite index in G unless otherwise stated. The subgroups H and K are assumed to be defined either by generators as words in G or by the coset table for H in G.

Intrinsics:

`H^u` : The conjugate H^u where u is a word in G.

H `meet` K : The intersection of subgroups H and K as a subgroup of G.

`G / H` : The quotient group of G by the normal closure of subgroup H.

`Core` $(G,\ H)$: The core of H in G.

`DerivedSubgroup` (G) : The derived subgroup of G.

`TorsionFreeRank` (G) : The torsion-free rank of the derived quotient group of G.

`MaximalOverGroup` $(G,\ H)$: A maximal overgroup of H in G.

`MinimalOverGroup` $(G,\ H)$: A minimal overgroup of H in G.

`NormalClosure` $(G,\ H)$: The normal closure of H in G.

`Normaliser` $(G,\ H)$: The normaliser of H in G.

Notes:

(i) The first four intrinsics involve using the Todd–Coxeter algorithm. In the case of `DerivedSubgroup` it is necessary for Magma to be able to compute the coset table for the derived group of G. Thus the intrinsic will fail if the derived group has very large or infinite index in G.

(ii) If H is defined by a coset table then the `SchreierGenerators` intrinsic can be used to produce generating words for H. In many cases this will be done automatically.

Example: Operations on a Subgroup of Index 448 (ex-f46c39)

The group $(8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (ab^{-1})^3 >,$$

and has a subgroup of index 448 generated by the words a^2 and ab^{-1}:

```
> G<a, b> := FPGroup<a, b| a^8, b^7, (a * b)^2, (a * b^-1)^3>;
> G;
Finitely presented group G on 2 generators
Relations
       a^8 = Id(G)
       b^7 = Id(G)
       (a * b)^2 = Id(G)
       (a * b^-1)^3 = Id(G)
> H<x, y> := sub< G | a^2, a * b^-1 >;
> H;
Finitely presented group H on 2 generators
Generators as words in group G
      x = a^2
      y = a * b^-1
```

The subgroup functions will now be demonstrated on the subgroup H. First of all its index and order are calculated.

```
> Index(G, H);
448
> Order(H);
24
```

The derived group of G is trivial:-

```
> DerivedSubgroup(G);
Finitely presented group
Index in group G is 1 = 1
Subgroup of group G defined by coset table
```

The group G is perfect so it is necessary to look elsewhere to find some other subgroups.

```
> P := MinimalOvergroup(G, H);
> P;
Finitely presented group P on 3 generators
Index in group G is 224 = 2^5 * 7
Generators as words in group G
    P.1 = a^2
    P.2 = a * b^-1
    P.3 = a * b^2 * a^3 * b^-1 * a^3 * b^3
> Order(P);
48
```

So H sits inside a subgroup P having twice its order.

```
> N := Normalizer(G, H);
> N;
Finitely presented group N on 4 generators
Index in group G is 112 = 2^4 * 7
Generators as words in group G
    N.1 = a^2
```

(continues on next page)

(continued from previous page)

```
    N.2 = a * b^-1
    N.3 = b^-1 * a * b^2 * a^4 * b^-1 * a * b^2
    N.4 = a * b^2 * a^3 * b^-1 * a^3 * b^3
> Order(N);
96
```

The subgroup H has a normaliser N of order 96.

```
> Q := MaximalOvergroup(G, H);
> Q;
Finitely presented group Q on 3 generators
Index in group G is 7
Generators as words in group G
    Q.1 = a^2
    Q.2 = a * b^-1
    Q.3 = b^-1 * a * b^2
> #Q;
1536
```

So the largest subgroup Q that contains H has index 7 in G.

```
> M := NormalClosure(G, H);
> M;
Finitely presented group M on 7 generators
Index in group G is 1 = 1
Generators as words in group G
    M.1 = a^2
    M.2 = a * b^-1
    M.3 = a^2 * b^-1 * a^-1
    M.4 = b^-1 * a
    M.5 = b * a * b^-2
    M.6 = (b^-1 * a * b)^2
    M.7 = b^-1 * a
```

To summarise, proper subgroups of G having orders 48, 96, and 1536 that contain H have been found. The normal closure of H is G.

79.4.6 Properties of Subgroups

The operations described in this subsection all require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, refer to chapter *Finitely Presented Groups*.

Intrinsics:

`u in H` : True if and only if the word u in G is a member of H.

`u notin H` : True if and only if the word u in G is not a member of H.

`H eq K` : True if and only if subgroups H and K are equal.

`H ne K` : True if and only if subgroups H and K are not equal.

`H subset K` : True if and only if the subgroup H is a subset of K.

`IsConjugate`(G, H, K) : True if and only if subgroups H and K are conjugate in G. If H and K are conjugate an element that conjugates H into K is also returned.

`IsMaximal`(G, H) : True if subgroup H is maximal in G.

`IsNormal`(G, H) : True if subgroup H is normal in G.

`IsSelfNormalizing`(G, H) : True if subgroup H is a self-normalising subgroup of G.

`IsPerfect`(H) : True if the subgroup H is perfect.

Example: Subgroup Predicates in a Space Group `(ex-b1118e)`

Some of these subgroup predicates are illustrated by applying them to some subgroups of the two-dimensional space group $p4g =< r,\ s \mid r^2,\ s^4,\ (r,s)^2 >$.

```
> p4g<r, s> := FPGroup< r, s | r^2 = s^4 = (r,s)^2 = 1 >;
> h := sub< p4g | (s^-1*r)^4, s*r >;
> k := sub< p4g | (s^-1*r)^2, (s*r)^2 >;
```

Subgroups h and k have the same index in $p4g$ but they are neither equal nor conjugate.

```
> Index(p4g, h), Index(p4g, k);
8 8
> h eq k;
false
> IsConjugate( p4g, h, k);
false
```

Are subgroups h and k normal in $p4g$?

```
> IsNormal(p4g, h), IsNormal(p4g, k);
false  true
```

Since h is not normal it makes sense to construct its normal closure n in $p4g$. It turns out that n is maximal in $p4g$ and contains k.

```
> n := NormalClosure(p4g, h);
Index( p4g, n)1
2
> IsMaximal(p4g, n);
true
> k subset n;
true
```

Here is a new subgroup l of $p4g$ that turns out to be conjugate to h.

```
> l := sub< p4g | (s*r)^4, s^-1*r >;
> IsConjugate(p4g, h, l);
true r^-1
```

i.e. $l = h^{r^{-1}}$. The intersection of h and $h^{r^{-1}}$ already yields the core of h in $p4g$.

```
> (h meet l) eq Core(p4g, h);
true
```

79.4.7 Presentations for Subgroups

Let H be a subgroup of finite index in the finitely presented group G. It frequently happens that it is desirable to construct a set of defining relations for H from those of G. Such a presentation can be obtained either on a set of Schreier generators for H or on the given generators of H using the

Reidemeister–Schreier rewriting technique [MKS76], if necessary together with extended coset enumeration [AR84, HKRR84]. It should be emphasised that if the user wishes only to determine the structure of the maximal abelian quotient of H, then the function `AbelianQuotientInvariants` should be used. In this case there is no need to first construct a presentation for H using the `Rewrite` function described below, since `AbelianQuotientInvariants` employs a special form of the Reidemeister–Schreier rewriting process which abelianises each relator as soon as it is constructed. Thus, compared to the function `Rewrite`, the function `AbelianQuotientInvariants` can be applied to subgroups of much larger index.

Intrinsics:

`Rewrite` $(G,\ H)$: Given an fp group G and a subgroup H having finite index in G, return a group K isomorphic to H with a presentation on some of the Schreier generators of H in G. The group K is created as a subgroup of G with a presentation on the new generators. The isomorphism from H onto K is returned as second return value. Note that the generators of K will, in general, not correspond to the generators of H.

`Rewrite` $(G,\ \sim H)$: Given a finitely presented group G and a subgroup H having finite index in G, a set of defining relations for H is constructed on the given generators. The information stored for H is changed to include these defining relations.

Notes:

(i) The first version of `Rewrite` selects its own generators and consequently is faster and often produces a simpler presentation. The second version constructs a presentation on the given generators. This is a more complicated process and the resulting presentation is often large. Consequently, by default, simplification is attempted using Tietze transformations that do not involve any changes to the generators of H. If is not necessary to obtain a presentation for H on the original generators of H, the first method is strongly recommended.

Example: Presentation of an Index 3 Subgroup `(ex-d4d467)`

Starting with the group G defined by

$$< x, y \mid x^2,\ y^3,\ (xy)^{12},\ (xy)^6(xy^{-1})^6 >,$$

a subgroup H of index 3 is generated by the words x, yxy^{-1}, and $yxy^{-1}xy^{-1}xy$:

```
> G<x, y> := FPGroup< x, y | x^2, y^3, (x*y)^12, (x*y)^6*(x*y^-1)^6 >;
> G;

Finitely presented group G on 2 generators
Relations
       x^2 = Id(G)
       y^3 = Id(G)
       (x * y)^12 = Id(G)
       x * y * x * y * x * y * x * y * x * y * x * y * x * y * x * y^-1 * x *
          y^-1 * x * y^-1 * x * y^-1 * x * y^-1 * x * y^-1 = Id(G)

> H := sub< G | x, y*x*y^-1, y*x*y^-1*x*y^-1*x*y >;
> H;
Finitely presented group H on 3 generators
Generators as words in group G
       H.1 = x
       H.2 = y * x * y^-1
       H.3 = y * x * y^-1 * x * y^-1 * x * y
> Index(G, H);
3
```

A presentation for H is now computed. Then using `AbelianQuotientInvariants`, H is found to have a non-trivial soluble quotient. Using the p-quotient algorithm described later, H is found to have a 2-quotient of order2^{62}!

```
> K := Rewrite(G, H);
> K;
Finitely presented group K on 3 generators
Generators as words in group G
    K.1 = x
    K.2 = y * x * y^-1
    K.3 = x^y
Relations
    K.1^2 = Id(T)
    K.2^2 = Id(T)
    K.3^2 = Id(T)
    (K.3 * K.2 * K.1 * K.3 * K.2)^2 = Id(K)
    (K.1 * K.3 * K.2 * K.1 * K.3)^2 = Id(K)
    (K.1 * K.2 * K.1 * K.3 * K.2)^2 = Id(K)
> AbelianQuotientInvariants(K);
[ 2, 2, 2 ]

> Q2 := pQuotient(K, 2, 30);
> FactoredOrder(Q2);
[ <2, 62> ]
```

Example: Rewriting a Presentation of L2(7) (ex-282060)

This example demonstrates how the intrinsic `Rewrite` can be used to obtain a presentation of a finitely presented group on a different set of generators.

Firstly, a presentation for $L_2(7)$ on two generators x and y is used to define the group.

```
> L27<x, y> := FPGroup< x, y | x^3 = 1, y^3 = 1, (x*y)^4 = 1,
>                           (y*y^x)^2 = y^x*y >;
```

The group is also generated by the elements $a = (xy)^2$ and $b = y$.

```
> H<a, b> := sub<L27 | (x*y)^2, y >;
> Index(L27, H);
1
```

At this point, no defining relations of $H \cong L27$ on the generators a and b are known.

```
> H;
Finitely presented group H on 2 generators
Index in group L27 is 1
Generators as words in group L27
    a = (x * y)^2
    b = y
```

The intrinsic `Rewrite` is applied to H as a subgroup of $L27$ in order to compute defining relations on the generators a and b.

```
> Rewrite(L27, ~H);
> H;
Finitely presented group H on 2 generators
Index in group L27 is 1
Generators as words in group L27
    a = (x * y)^2
    b = y
Relations
    a^2 = Id(H)
```

(continues on next page)

(continued from previous page)

```
    b^3 = Id(H)
    (a * b)^7 = Id(H)
    (a * b^-1 * a * b)^4 = Id(H)
    (b * a * b^-1 * a * b * a)^4 = Id(H)
```

The fifth relation turns out to be redundant: a and b are standard generators for $L_2(7)$ as is shown here:

```
> Order(DeleteRelation(H, 5)) eq Order(H);
true
```

79.5 Finite FP-Groups

Finite fp-groups are often of interest and so a small number of intrinsics are provided for such groups.

Intrinsics:

`IsFinite` (G) : Returns true if G is proved to be finite, false if unknown. If G is found to be finite, its order is also returned.

`Order` (G) : Given a finite fp-group G return the order of G. If the order cannot be determined or is found to be infinite, then the value zero or `Infinity`, respectively, is returned.

`PermutationGroup` (G) : Given a finite fp-group G return a permutation group P that is isomorphic to G together with the isomorphism $\phi : G \to P$.

`CayleyGraph` (G) : If G is an fp-group with generating set X return the Cayley graph CG of G with respect to the generating set X. The Cayley graph is returned as a Magma directed graph with its vertices and edges labelled. A parameter `Labelled` can be set to *false* to prevent the edges and vertices being labelled.

Notes:

Note that the intrinsics `IsFinite`, `Order`, `PermutationGroup`, and `CayleyGraph` may need to construct the coset table of G over the identity and hence are generally restricted to groups of order less than a million. The exact limit is determined by the amount of memory available.

Example: Order of a Finite FP-Group `(ex-fb0cc3)`

The intrinsic `Order` is invoked without any parameter settings to compute the order of the group

$$G = \langle a, b \mid a^8, b^7, (ab)^2, (ab^{-1})^3 \rangle$$

```
> G<a, b> := FPGroup<a,b | a^8, b^7, (a*b)^2, (a*b^-1)^3>;
> G;
Finitely presented group G on 2 generators
Relations
    a^8 = Id(G)
    b^7 = Id(G)
    (a * b)^2 = Id(G)
    (a * b^-1)^3 = Id(G)
> Order(G);
10752
```

Example: Cayley Graph of a Finite FP-Group `(ex-19bd36)`

Now the Cayley graph of the group defined in the previous example is constructed. Next the greatest distance between any pair of vertices (the diameter) is found.

```
> G<a, b> := FPGroup<a, b | a^8, b^7, (a*b)^2, (a*b^-1)^3>;
> CG := CayleyGraph(G);
> Diameter(CG);
22
```

Finally, something more sophisticated is demonstrated. First the shortest path g (geodesic) between vertex 1 and vertex 1000 is found. Then the labels on the edges of this path are multiplied together yielding the element of G corresponding to vertex 1000.

```
> V := Vertices(CG);
> E := Edges(CG);
> g := Geodesic(V!1, V!1000);
> g;
[ 1, 2, 6, 16, 38, 33, 73, 82, 163, 309, 168, 318, 579, 999, 1642, 1000 ]
> w := &*[ Label(E![g[i], g[i+1]]) : i in [1.. #g-1] ];
> w;
a^5 * b * a^3 * b^4 * a * b
```

79.6 Homomorphisms of FP-Groups

In this section the following topics are covered:

(i) The specialisation of the general `hom`-constructor to the case in which the domain is an fp-group

(ii) Searching for homomorphisms from an fp-group to a given permutation group.

(iii) Searching for an isomorphism between two fp-groups.

While this section presents general machinery for fp-group homomorphisms the following section covers more specialised tools, specifically those that search for specific types of epimorphisms.

79.6.1 Homomorphism Constructor

The specialisation of the general `hom`–constructor to the case in which the domain is an fp-group is described in this section.

Constructor/Intrinsic:

`hom`$\langle\ P \to G \mid S\ \rangle$: Returns the homomorphism from the n-generator fp-group P to the group G defined by the assignment S. Here S is either a list, sequence, or indexed set containing the images of the n generators of P or a list, sequence, enumerated set, or indexed set, containing n tuples $\langle x_i,\ y_i\rangle$ or n arrow pairs $x_i \to y_i$ defining the n generator-image pairs.

`IsSatisfied`(U, E) : Let U be a set or sequence of words or relations constituting the defining presentation of an n-generator fp-group H and E be a sequence of n elements in a group G for which there is a unique representation for each element. This intrinsic evaluates the words/relations of U in G where the i-th generator of H maps to the i-th element of G and returns `true` if and only if each word in U evaluates to the identity or each relation in U is satisfied.

Notes:

(i) The `hom` constructor does not check that the given mapping is a homomorphism and so the `IsSatisfied` intrinsic is provided to allow the user to verify the correctness of the definition of those homomorphisms that are from an fp-group to a group whose elements have unique representations.

(ii) The kernel of a homomorphism having a domain of type `GrpFP` can be calculated using the intrinsic `Kernel` in many cases. This is the case if the codomain is one of the group types `GrpPerm`, `GrpMat`, `GrpAb`, `GrpPC`, `GrpGPC`, as well as the image being finite and its order small. The kernel may also be computable when the codomain is of the type `GrpFP` if the image is finite, sufficiently small, and has a known presentation.

(iii) The standard tools for working with homomorphisms are available for fp-groups: image of an element or subgroup `x@phi` (or `phi(x)`, preimage of an element or subgroup `x@@phi`, domain (`Domain`), codomain (`Codomain`), image (`Image`), and kernel (`Kernel`). However, in some cases it will not be possible to compute the kernel of a homomorphism and if the kernel is not available then the computation of preimages of subgroups will not be possible.

Example: Symmetric Group as Image of a Braid Group (ex-1337d3)

For arbitrary $n > 0$, the symmetric group of degree $n+1$ is an epimorphic image of the braid group on n generators. In this example, this relationship for the case $n = 4$ will be exhibited.

The first step is to create the braid group B on 5 strings, that is, 4 Artin generators.

```
> B<x,y,z,t> := BraidFPGroup(5);
> B;
Finitely presented group B on 4 generators
Relations
    x * y * x = y * x * y
    x * z = z * x
    x * t = t * x
    y * z * y = z * y * z
    y * t = t * y
    z * t * z = t * z * t
```

In the symmetric group of degree 5, 4 transpositions are defined which will be the images of the generators of B.

```
> S := Sym(5);
> imgs := [ S!(1,2), S!(2,3), S!(3,4), S!(4,5) ];
```

In order to verify that this assignment actually gives rise to a well-defined homomorphism, a check is made to decide whether the potential images satisfy the defining relations of B.

```
> rels := Relations(B);
> rels;
[ x*y*x = y*x*y, x*z = z*x, x*t = t*x, y*z*y = z*y*z,
  y*t = t*y, z*t*z = t*z*t ]
> IsSatisfied(rels, imgs);
true
```

They do. So now the homomorphism from B to S can be defined:-

```
> f := hom< B->S | imgs >;
```

Note that f is surjective, i.e., S is an epimorphic image of B as claimed above.

```
> f(B) eq S;
true
```

What is the kernel of f?

```
> Kernel(f);
Finitely presented group
Index in group B is 120 = 2^3 * 3 * 5
Subgroup of group B defined by coset table
```

Using the function `GeneratingWords` described elsewhere, a set of generators for $\ker(f)$ as a subgroup of B can be constructed.

```
> GeneratingWords(B, Kernel(f));
{  x^-2, y^-2, z^-2, t^-2, (y*z^-1*y^-1)^2, (z*t^-1*z^-1)^2,
   (x*y^-1*x^-1)^2, (x*y*z^-1*y^-1*x^-1)^2, (y*z*t^-1*z^-1*y^-1)^2,
   (x*y*z*t^-1*z^-1*y^-1*x^-1)^2 }
```

It is easy to see that all generators of $\ker(f)$ are conjugates of words of the form $g^{\pm 2}$, where g is a generator of B. This is checked, using the normal closure intrinsic `NormalClosure`.

```
> N := sub< B | x^2, y^2, z^2, t^2 >;
> Kernel(f) eq NormalClosure(B, N);
true
```

Thus the braid relations together with the relators x^2, y^2, z^2, t^2 are a set of defining relations for S. This is confirmed by running the following code:-

```
> BS := quo< B | x^2, y^2, z^2, t^2 >;
> Order(BS);
120
> P := PermutationGroup(BS);
> P;
Permutation group P acting on a set of cardinality 10
Order = 120 = 2^3 * 3 * 5
> IsIsomorphic(P, Sym(5));
true
```

79.6.2 Searching for Homomorphisms

This section describes functions for computing representatives of the classes of homomorphisms from a finitely presented group G to a finite group H modulo a group A of automorphisms of H. The algorithm and code are due to Derek Holt.

Intrinsics:

`Homomorphisms` (G, H, A) : Given a finitely presented group G and two permutation groups H and A with $H \lhd A$, a sequence containing representatives of the classes of homomorphisms from G to H modulo automorphisms of G induced by elements of A is returned.

`Homomorphisms` (G, H, A) : This version is identical except that H and A are polycyclic groups rather than permutation groups.

Notes:

(i) Two homomorphisms $\phi_1, \phi_2 : G \to H$ are considered equivalent if there exists an element $a \in A$ such that $\phi_1(x) = \phi_2(x)^a$ for all $x \in G$.

(ii) There is a process version of this intrinsic; that is, a restartable version that returns each homomorphism as soon as it is discovered.

(iii) The call `Homomorphisms(G, H, H)` may be abbreviated to `Homomorphisms(G, H)`.

(iv) The permutation group version of the intrinsic has parameters `Surjective`, `Limit`, `TimeLimit`, `CosetEnumeration`, and `CacheCosetAction`. If `Surjective` is set to `true` then the search is restricted to epimorphisms. The final two parameters relate to the runtime and memory usage of the calculation. The polycyclic group version of the intrinsic has only parameters `Surjective` and `Limit`. The default values of `Surjective` and `Limit` are `true` and 1, respectively. If `Limit` is set to 0 this indicates no limit.

Example: Homomorphism onto A5 `(ex-1803a5)`

Consider the finitely presented group

$$G := < a, b, c \mid ac^{-1}bc^{-1}aba^{-1}b, abab^{-1}c^2b^{-1}, a^2b^{-1}(ca)^4cb^{-1} > .$$

```
> G := FPGroup< a,b,c | a*c^-1*b*c^-1*a*b*a^-1*b,
>                      a*b*a*b^-1*c^2*b^-1,
>                      a^2*b^-1*c*a*c*a*c*a*c*a*c*b^-1 >;
```

The intrinsic `Homomorphisms` will be used to prove that there is a homomorphism from G onto A_5.

```
> H := Alt(5);
> homs := Homomorphisms(G, H : Limit := 1);
> homs;
[
    Homomorphism of GrpFP: G into GrpPerm: H, Degree 5, Order 2^2 * 3 * 5
        G.1 |--> (1, 2, 3)
        G.2 |--> (1, 2, 4)
        G.3 |--> (2, 4)(3, 5)
]
```

Hence there exists at least one such homomorphism. Are there any more?

```
> homs := Homomorphisms(G, H);
> #homs;
4
```

So there are four distinct classes of homomorphisms! Finally, the kernel of the first of these homomorphisms is found.

```
> phi :=  homs[1];
> K := Kernel(phi);
> K;
Finitely presented group K
Index in group G is 60 = 2^2 * 3 * 5
Subgroup of group G defined by coset table
```

79.6.3 Searching for Isomorphisms

This section describes an intrinsic that searches for isomorphisms between two finitely presented groups. The algorithm and code are due to Derek Holt.

Intrinsics:

`SearchForIsomorphism` $(G,\ H,\ m)$: Attempt to find an isomorphism from the finitely presented group G to the finitely presented group H. The search will be restricted to those homomorphisms for which the sum of the word-lengths of the images of the generators of G in H is at most m. If an isomorphism ϕ is found, then the values `true`, ϕ, ϕ^{-1} are returned. Otherwise, the values `false`, _, _ are returned; of course, that does not necessarily mean that the groups are not isomorphic.

Notes:

(i) Setting the verbose flag `"IsoSearch"` to 1, will cause information about the progress of the search to be printed.

(ii) There are a number of parameters available for the intrinsic `SearchForIsomorphism`. If the boolean parameter `All` is set to `true` then the search will be continued until all possible isomorphisms that are defined by images of the generators of G in H that satisfy the image length condition are considered. If the boolean parameter `IsomsOnly` is set to `false` then homomorphisms $\phi : F \to G$ will be sought. The reader should consult chapter *Finitely Presented Groups* for a full description of all the parameters.

Example: Isomorphism Search for Hillman's Groups `(ex-cebd96)`

Jonathon Hillman asked whether the following two groups are isomorphic:

```
> G1<s,t,u> := FPGroup <s,t,u | s*u*s^-1=u^-1, t^2=u^2, t*s^2*t^-1=s^-2,
>                              u*(s*t)^2=(s*t)^2*u >;
> G2<x,y,z> := FPGroup<x,y,z | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=z^2*(x*y)^2,
>                             y^2=(z^-1*x)^2, z*(x*y)^2=(x*y)^2*z >;
> isiso, f1, f2 := SearchForIsomorphism(G1,G2,7);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1 into GrpFP: G2 induced by
    s |--> x * z^-1
    t |--> y * z
    u |--> x * y^-1 * z
> f2;
Homomorphism of GrpFP: G2 into GrpFP: G1 induced by
    x |--> s^2 * u * t^-1
    y |--> s^-1 * u^-1
    z |--> s * u * t^-1
```

So the search for an isomorphism succeeded and returned the isomorphism and its inverse.

Example: Isomorphism Search for Neumann's Groups (ex-8e44ee)

Walter Neumann asked whether these two groups are isomorphic:

```
> G1<x,y,z> := FPGroup< x,y,z | x^2*y^5, x^14*z^23, (x^2,y), (x^2,z), x*y*z>;
> G2<a,b> := FPGroup<a,b | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7>;
```

It is a good idea to minimize the number of generators of $G1$ since there is clearly a redundant generator.

```
> G1s := Simplify(G1);
> Ngens(G1s);
2
> G1!G1s.1, G1!G1s.2;
x z
```

Now `SearchForIsomorphism` ($G1$, $G2$, 15) will succeed, but will take many hours, and also use a lot of memory. In contrast to $G1$, it can sometimes help to introduce a new generator in the presentation for $G2$ if there are common substrings in relators.

```
> G2b<a,b,c> := FPGroup< a,b,c | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7, c=b^7>;
> isiso, f1, f2 := SearchForIsomorphism(G1s,G2b,4);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1s into GrpFP: G2b induced by
    G1s.1 |--> a * c^-1
    G1s.2 |--> c
> f2;
Homomorphism of GrpFP: G2b into GrpFP: G1s induced by
    a |--> G1s.1 * G1s.2
    b |--> G1s.1^6 * G1s.2^10
    c |--> G1s.2
```

79.7 Quotient Groups

A powerful technique for investigating an fp-group is to study some of its quotients. For example, the action of a group G on the cosets of a subgroup H (where the index of H in G is finite) as described in the section on subgroups is an epimorphism of G onto a permutation group of degree $[G : H]$. In this section further methods for constructing various types of quotient group are described. Specifically, machinery is provided for constructing abelian quotients, p-quotients, nilpotent quotients, soluble quotients, and $PSL(2, q)/PGL(2, q)$ quotients.

79.7.1 Abelian Quotient

Computing the quotient of an fp-group G by its derived group is usually the first step in analyzing an fp-group. The ability to do this to finite-index subgroups of G is a key step when trying to prove that a group is infinite. The intrinsics in this section compute information about the abelian quotient of an fp-group G.

Intrinsics:

`AbelianQuotient` (G) : The maximal abelian quotient G/G' of the fp-group G is returned as a group in category `GrpAb` (cf. chapter *Abelian Groups*). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`HasFiniteAbelianQuotient` (G) : Return `true` if the maximal abelian quotient of the fp-group G is finite.

`ElementaryAbelianQuotient` $(G,\ p)$: The maximal p-elementary abelian quotient Q is returned as a group G in category `GrpAb` (cf. chapter *Abelian Groups*). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`AbelianQuotientInvariants` (G) : Let G be an fp-group or a subgroup of finite index of an fp-group. The elementary divisors of the derived quotient group G/G' are returned as a sequence of integers.

`AbelianQuotientInvariants` $(G,\ n)$: Let G be an fp-group, n a small integer, and N the normal subgroup of G generated by the derived group G' together with all n-th powers of elements of G. The elementary divisors of the quotient group G/N, are returned as a sequence of integers.

`AbelianQuotientPrimes` (G) : Return a sequence of primes containing all prime divisors in the order of G/G', if this is finite. Note that there may be primes in the return sequence which do not divide this order. If `[ 0 ]` is returned then G/G' is infinite. If `[]` is returned, then G is perfect.

`HasComputableAbelianQuotient` (G) : Given an fp-group G, this intrinsic tests whether the abelian quotient of G can be computed. If so, it returns the value `true`, the abelian quotient A of G, and the natural epimorphism $\pi : G \to A$. If the abelian quotient of G cannot be computed, the value `false` is returned.

`HasInfiniteComputableAbelianQuotient` (G) : Given an fp-group G, this function tests whether the abelian quotient of G can be computed and is infinite. If so, it returns the value `true`, the abelian quotient A of G, and the natural epimorphism $\pi : G \to A$. If the abelian quotient of G cannot be computed or if it is finite, the value `false` is returned.

Notes:

(i) A number of variations on the above intrinsics may be found in chapter *Finitely Presented Groups*.

(ii) The above intrinsics require either a presentation for G, or the coset table for G inside some supergroup of G which has a presentation. If such a coset table cannot be computed, an error will result.

(iii) Some functions may require the computation of a coset table. Experienced users can control the behaviour of an implicit coset enumeration by setting global parameters which can be changed using the function `SetGlobalTCParameters`.

(iv) Intrinsics with names of the form `AbelianQuotientName` can be abbreviated to `AQName`, where name is `Quotient`, `Primes`, or `Invariants`.

(v) Considerable effort has been put into computing the Smith Normal Form for large integer matrices that can arise when executing these intrinsics. An important application where such large matrices arise is when the input is the integer matrix resulting from the abelianisation of the presentation of a subgroup H which has been constructed by rewriting the presentation of its parent group. If the index is large (even 50 or 100) then the presentation of H may be huge.

(vi) The intrinsic `HasInfiniteComputableAbelianQuotient` first checks the modular abelian invariants for a set of small primes. If for one of these primes the modular abelian quotient is trivial, then A must be finite and the function returns without actually computing the abelian quotient. This heuristic may save time if only infinite quotients are of interest.

Example: Structure of the Fibonacci Group F(7) (ex-12a8db)

The Fibonacci group $F(7)$ has the following 2-generator presentation:

$$< a, b \mid a^2b^{-2}a^{-1}b^{-2}(a^{-1}b^{-1})^2, abab^2abab^{-1}(ab^2)^2 > .$$

The goal is to determine the structure of this group.

```
> F<a, b> := FreeGroup(2);
> F7<a, b> := quo< F |  a^2*b^-2*a^-1*b^-2*(a^-1*b^-1)^2,
>        a*b*a*b^2*a*b*a*b^-1*(a*b^2)^2 >;
> F7;
Finitely presented group F7 on 2 generators
Relations
       a^2 * b^-2 * a^-1 * b^-2 * a^-1 * b^-1 * a^-1 * b^-1 =Id(F7)
       a * b * a * b^2 * a * b * a * b^-1 * a * b^2 * a * b^2 = Id()7
```

The first step is to determine the structure of the maximal abelian quotient of $F(7)$.

```
> AbelianQuotientInvariants(F7);
[ 29 ]
```

The maximal abelian quotient of $F(7)$ is cyclic of order 29. At this point there is no obvious way to proceed, so an attempt is made to determine the index of some subgroups.

```
> Index( F7, sub< F7 | a > );
1
```

This demonstrates that $F(7)$ is generated by a and so must be cyclic. This fact coupled with the knowledge that its abelian quotient has order 29 proves that the group is cyclic of order 29.

Example: Abelian Quotient Invariants of a Subgroup (ex-26e129)

The group $G = (8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (ab^{-1})^3 > .$$

The goal here is to determine the abelian quotient invariants of the (rather famous) subgroup H of G, generated by the words a^2 and ab^{-1}:

```
> G<a, b> := FPGroup<a, b| a^8, b^7, (a * b)^2, (a * b^-1)^3>;
> H<x, y> := sub< G | a^2, a * b^-1 >;
```

The fastest way to determine the order of the maximal 2-elementary abelian quotient of H is to use the modular version of `AbelianQuotientInvariants`:

```
> #AbelianQuotientInvariants(H, 2);
1
```

So the maximal 2-elementary abelian quotient of H has order 2^1.

79.7.2 p-Quotient

Let F be a finitely presented group, p a prime, and c a positive integer. A p-quotient algorithm constructs a consistent power-conjugate presentation for the largest p-quotient of F having lower exponent-p class at most c. The p-quotient algorithm used by Magma is part of the ANU p-Quotient program. For details of the algorithm, see [NO96]. The result is returned as a group of type `GrpPC` (cf. chapter Finite Soluble Groups).

Intrinsics:

`pQuotient` $(G,\ p,\ c)$: Given an fp-group G, a prime p, and a positive integer c, construct a pc-presentation for the largest p-quotient Q of G having lower exponent-p class at most c. If c is given as 0 then the limit 127 is placed on the class. The prime p must be less than 65536. The intrinsic also returns the natural homomorphism π from G to Q, a sequence S describing the definitions of the pc-generators of Q and a flag indicating whether Q is the maximal p-quotient of G.

Notes:

(i) If the fp-group is a finite p-group then the p-quotient function can be used to construct a pc-presentation for the group.

(ii) If the parameter `Metabelian` is set to `true` then a consistent pc-presentation is constructed for the largest metabelian p-quotient of G having lower exponent-p class at most c.

(iii) Assume that the p-quotient Q has order p^n, Frattini rank d, and that its generators are $a_1, \ldots, a_n$. Then the power-conjugate presentation constructed has the following additional structure. The set $\{a_1, \ldots, a_d\}$ is a generating set for G. For each a_k in $\{a_{d+1}, \ldots, a_n\}$, there is at least one relation whose right hand side is a_k. One of these relations is taken as the *definition* of a_k.

(iv) The intrinsic has the following parameters in addition to `Metabelian`:

- `Exponent` enables an exponent law, $x^m = 1$, to be imposed on the quotient being computed. It is set to a positive integer value m.
- `Print` enables the user to obtain verbose output during the running of the intrinsic. Legal values are the integers 1, 2, and 3.
- `Workspace` allows the user to specify the size of workspace that will be assigned to the p-quotient job.

(v) A process version of the p-quotient algorithm is provided. This enables the user to create a restartable computation which gives the user complete control over its execution. For a description see chapter *Finitely Presented Groups*.

(vi) The k-th term of the return value sequence S is a sequence of two integers, describing the definition of the k-th pc-generator $Q.k$ of Q as follows:-

- If $S[k] = [0, r]$, then $Q.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $Q.k$ is defined via the power relation for $Q.r$.
- If $S[k] = [r, s]$, then $Q.k$ is defined via the conjugate relation involving $(Q.r)^{Q.s}$.

Example: Largest 2-Quotient of Class 6 `(ex-81ff16)`

In this example the largest 2-quotient of class 6 for a two-generator, two-relator group is constructed.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | (b, a, a) = 1, (a * b * a)^4 = 1 >;
> Q, fQ := pQuotient(G, 2, 6);
> Order(Q);
524288
> fQ;
Mapping from: GrpFP: G to GrpPC: Q
```

Example: 3-Quotient of a Group of Exponent 9 `(ex-94855c)`

The largest 3-quotient of class 6 for a two-generator group of exponent 9 will be constructed.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | a^3 = b^3 = 1 >;
> q := pQuotient(G, 3, 6: Print := 1, Exponent := 9);
Lower exponent-3 central series for G
Group: G to lower exponent-3 central class 1 has order 3^2
Group: G to lower exponent-3 central class 2 has order 3^3
Group: G to lower exponent-3 central class 3 has order 3^5
Group: G to lower exponent-3 central class 4 has order 3^7
Group: G to lower exponent-3 central class 5 has order 3^9
Group: G to lower exponent-3 central class 6 has order 3^11
```

Example: Metabelian 5-Quotient (ex-e48428)

The parameter `Metabelian`: will be used to construct a metabelian 5-quotient of the group

$$< a, b \mid a^{625} = b^{625} = 1, (b, a, b) = 1, (b, a, a, a, a) = (b, a)^5 > .$$

```
> F<a, b> := FreeGroup(2);
> G := quo< F | a^625 = b^625 = 1, (b, a, b) = 1,
>           (b, a, a, a, a) = (b, a)^5 >;
> q := pQuotient(G, 5, 20: Print := 1, Metabelian := true);
Lower exponent-5 central series for G
Group: G to lower exponent-5 central class 1 has order 5^2
Group: G to lower exponent-5 central class 2 has order 5^5
Group: G to lower exponent-5 central class 3 has order 5^8
Group: G to lower exponent-5 central class 4 has order 5^11
Group: G to lower exponent-5 central class 5 has order 5^12
Group: G to lower exponent-5 central class 6 has order 5^13
Group: G to lower exponent-5 central class 7 has order 5^14
Group: G to lower exponent-5 central class 8 has order 5^15
Group: G to lower exponent-5 central class 9 has order 5^16
Group: G to lower exponent-5 central class 10 has order 5^17
Group: G to lower exponent-5 central class 11 has order 5^18
Group: G to lower exponent-5 central class 12 has order 5^19
Group: G to lower exponent-5 central class 13 has order 5^20
Group completed. Lower exponent-5 central class = 13, order = 5^20
```

79.7.3 Nilpotent Quotient

A nilpotent quotient algorithm constructs, from a finite presentation of a group, a polycyclic presentation for a nilpotent quotient of the finitely presented group. The nilpotent quotient algorithm used by Magma is the ANU Nilpotent Quotient program, as described in [Nic96]. The version included in Magma is Version 2.2 of January 2007.

The lower central series $G_0, G_1, \ldots$ of a group G can be defined inductively as $G_0 = G$, $G_i = [G_(i-1), G]$. The group G is said to have nilpotency class c if c is the smallest non-zero integer such that $G_c = 1$. If N is a normal subgroup of G and G/N is nilpotent, then N contains G_i for some non-negative integer i. G has infinite nilpotent quotients if and only if G/G_1 (the maximal abelian quotient of G) is infinite and a prime p divides a finite factor of a nilpotent quotient if and only if p divides a cyclic factor of G/G_1.

Intrinsics:

`NilpotentQuotient` $(G,\ c)$: The class c nilpotent quotient of G is computed as a group in category *GrpGPC*, and is returned together with the epimorphism π from G onto this quotient. When c is set to zero, the function attempts to compute the maximal nilpotent quotient of G.

Notes:

(i) There are a number of parameters associated with this intrinsic and descriptions of these can be found in chapter *Finitely Presented Groups*.

(ii) The verbose identifier for this intrinsic is `NilpotentQuotient` which takes a small positive integer n in the range [0..3]. Setting n to be 0 turns printing off and this is the default value.

Example: Class 2 Nilpotent Quotient (ex-0b5725)

Here is a finitely presented group. The abelian quotient is infinite so the class 2 nilpotent quotient is computed.

```
> G := Group<x,y,z|(x*y*z^-1)^2, (x^-1*y^2*z)^2, (x*y^-2*x^-1)^2 >;
> AbelianQuotient(G);
Abelian Group isomorphic to Z/2 + Z/2 + Z
Defined on 3 generators
Relations:
  2*$.1 = 0
  2*$.2 = 0
> N := NilpotentQuotient(G,2); N;
GrpGPC : N of infinite order on 6 PC-generators
PC-Relations:
  N.1^2 = N.3^2 * N.5,
  N.2^2 = N.4 * N.6,
  N.4^2 = Id(N),
  N.5^2 = Id(N),
  N.6^2 = Id(N),
  N.2^N.1 = N.2 * N.4,
  N.3^N.1 = N.3 * N.5,
  N.3^N.2 = N.3 * N.6
```

Example: Free Nilpotent Groups via Nilpotent Quotient (ex-470660)

The free nilpotent group of rank r and class e is defined as $F/\gamma_{e+1}(F)$, where F is a free group of rank r and $\gamma_{e+1}(F)$ denotes the $(e+1)$ st term of the lower central series of F. The intrinsic `NilpotentQuotient` will be used to construct the free nilpotent group N of rank 2 and class 3 as quotient of the free group F of rank 2 and the natural epimorphism from F onto N.

```
> F<a,b> := FreeGroup(2);
> N<[x]>, pi := NilpotentQuotient(F, 3);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
    x[2]^x[1] = x[2] * x[3],
    x[2]^(x[1]^-1) = x[2] * x[3]^-1 * x[4],
    x[3]^x[1] = x[3] * x[4],
    x[3]^(x[1]^-1) = x[3] * x[4]^-1,
    x[3]^x[2] = x[3] * x[5],
    x[3]^(x[2]^-1) = x[3] * x[5]^-1
```

Next the nilpotency class of the quotient will be confirmed..

```
> NilpotencyClass(N);
3
```

Example: Nilpotent Quotient of a Baumslag-Solitar Group (ex-f4b49e)

The Baumslag–Solitar groups

$$BS(p,q) = \langle\ a,\ b \mid ab^p a^{-1} = b^q \rangle$$

form a fascinating class of 1-relator groups. The nilpotent quotient of class 4 for $q = 4$, will be computed and the preimages of its generators identified.

```
> G<a,b> := FPGroup<a,b|a*b*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
  N.2^3 = N.3^2 * N.4^2 * N.5,
  N.3^3 = N.4^2 * N.5^2,
  N.4^3 = N.5^2,
  N.5^3 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3^2 * N.4^2 * N.5,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4^2 * N.5^2,
  N.4^N.1 = N.4 * N.5,
  N.4^(N.1^-1) = N.4 * N.5^2
> for i := 1 to Ngens(N) do N.i @@ f; end for;
 a
 b
 (b, a)
 (b, a, a)
 (b, a, a, a)
```

79.7.4 Soluble Quotient

A soluble quotient algorithm computes a consistent power-conjugate presentation (pc-presentation) of the largest finite soluble quotient of an fp-group, subject to certain algorithmic and user supplied restrictions. In this section a description of a basic version of the algorithm available within Magma in given. For more information the user is referred to chapter *Finitely Presented Groups*.

Intrinsics:

`SolvableQuotient/SolubleQuotient` (G) : Given an fp-group G, the largest finite soluble quotient Q is returned. The epimorphism $\phi : G \to Q$ is also returned.

`SolvableQuotient/SolubleQuotient` (G, n) : Given an fp-group G and a positive integer n, a soluble quotient Q of order n is returned. The $\phi : G \to Q$ is also returned. When the order of Q is known in advance, this version can avoid work in proving that the final quotient is maximal.

`SolvableQuotient/SolubleQuotient` (G, P) : Given an fp-group G and a sequence P containing primes this intrinsic calculates the largest soluble quotient whose order has prime divisors only in P. The epimorphism $\phi : G \to Q$ is also returned. This version is included since finding the possible primes at each step can be very expensive.

Notes:

(i) A user wishing to get the most out of `SolubleQuotient` should note that there is a complex interface to it and the user should read the full description of this intrinsic, its variations, and the parameters available, in chapter *Finitely Presented Groups*.

(ii) In the case of `SolubleQuotient` $(G,\ n)$, if $n > 0$ is not the order of the maximal finite soluble quotient, then it may happen that no group of order n can be found, since an epimorphic image of size n may not be exhibited by the chosen series.

Example: Soluble Quotient of a Two-Generator Group `(ex-211a7e)`

The soluble quotient of the group

$$\langle a, b \mid a^2, b^4, ab^{-1}ab(abab^{-1})^5ab^2ab^{-2}\rangle$$

will be computed.

```
> G<a,b> := FPGroup< a, b | a^2, b^4,
>                     a*b^-1*a*b*(a*b*a*b^-1)^5*a*b^2*a*b^-2 >;
> Q := SolubleQuotient(G);
> Q;
GrpPC : Q of order 1920 = 2^7 * 3 * 5
PC-Relations:
    Q.1^2 = Q.4,
    Q.2^2 = Id(Q),
    Q.3^2 = Q.6,
    Q.4^2 = Id(Q),
    Q.5^2 = Q.7,
    Q.6^2 = Id(Q),
    Q.7^2 = Id(Q),
    Q.8^3 = Id(Q),
    Q.9^5 = Id(Q),
    Q.2^Q.1 = Q.2 * Q.3,
    Q.3^Q.1 = Q.3 * Q.5,
    Q.3^Q.2 = Q.3 * Q.6,
    Q.4^Q.2 = Q.4 * Q.5 * Q.6 * Q.7,
    Q.4^Q.3 = Q.4 * Q.6 * Q.7,
    Q.5^Q.1 = Q.5 * Q.6,
    Q.5^Q.2 = Q.5 * Q.7,
    Q.5^Q.4 = Q.5 * Q.7,
    Q.6^Q.1 = Q.6 * Q.7,
    Q.8^Q.1 = Q.8^2,
    Q.8^Q.2 = Q.8^2,
    Q.9^Q.1 = Q.9^3,
    Q.9^Q.2 = Q.9^4,
    Q.9^Q.4 = Q.9^4
```

Example: Order of a Large Soluble Quotient (ex-820eb8)

Consider the group G defined by the presentation

$$\langle x, y \mid x^3, y^8, [x, y^4], x^{-1}yx^{-1}y^{-1}xyxy^{-1}, (xy^{-2})^2(x^{-1}y^{-2})^2(xy^2)^2(x^{-1}y^2)^2, (x^{-1}y^{-2})^6(x^{-1}y^2)^6\rangle.$$

```
> G<x, y> := FPGroup< x, y | x^3, y^8, (x,y^4), x^-1*y*x^-1*y^-1*x*y*x*y^-1,
>   (x*y^-2)^2*(x^-1*y^-2)^2*(x*y^2)^2*(x^-1*y^2)^2, (x^-1*y^-2)^6*(x^-1*y^2)^6 >
↪;
```

The soluble quotient algorithm will be applied to G and its order computed.

```
> time Q := SolubleQuotient(G);
Time: 4.620
> Order(Q);
165888
```

Note that $165888 = 2^{11} \cdot 3^4$. Knowing the primes in advance makes the computation faster:

```
> time Q := SolubleQuotient(G, {2, 3});
Time: 1.900
```

Assuming that G is finite the Todd–Coxeter procedure will used to find its order:

```
> Order(G);
165888
```

Hence the group G is finite, soluble, and isomorphic to Q. Since Q is defined by a pc-presentation anything can be computed for it and so it can be used to obtain information about G.

```
> #ConjugacyClasses(Q);
272
```

So G has 272 conjugacy classes of elements.

79.7.5 Simple Group Quotients

The previous four methods for finding epimorphisms of an fp-group G only apply when G has a non-trivial abelian quotient, that is, when G is not perfect. This and the next subsection present methods for the case in which G is perfect. The first method searches through a list of non-abelian simple groups to find simple quotients of G as permutation groups. It is similar to the algorithm used to find homomorphisms of an fp-group onto a permutation group. Currently (2019) the list of target simple groups consists of all non-abelian simple groups having order less than 10^{10}. Such a list is dominated by $\mathrm{PSL}(2, q)$'s with q odd. In the current implementation these $\mathrm{PSL}(2, q)$'s are treated as an infinite family rather than stored individually, and so continue beyond the above limit.

Intrinsics:

`SimpleQuotients` $(G,\ deg1,\ deg2,\ ord1,\ ord2)$:

`SimpleQuotients` $(G,\ ord1, ord2)$:

The homomorphism algorithm is used to find epimorphisms from G onto non-abelian simple groups in a fixed list. The arguments $deg1$ and $deg2$ are respectively lower and upper bounds for the degree of the image group. If the degree arguments are not present then bounds of 5 and 10^{10} are used. The arguments $ord1$ and $ord2$ are, respectively, lower and upper bounds for the orders of the image group. (Setting $ord2$ low enough is particularly important if a quick search is required. If $ord1$ is not given then it defaults to 1. The return value is a list of sequences containing the epimorphisms found. Each sequence contains epimorphisms onto one simple group.

The parameter `Limit` limits the number of successful searches to be performed by `Homomorphisms`. The default value is 1, so by default the search terminates with the first simple group found to be a homomorphic image of G.

Notes:

(i) The parameter `Limit` limits the number of successful searches that are to be performed by `Homomorphisms`. The default value is 1, so by default the search terminates with the first simple group found to be a homomorphic image of G.

(ii) The parameter `HomLimit` limits the number of homomorphisms that will be searched for by any particular call to `Homomorphisms`. It defaults to zero, so that all homomorphisms for any group found will be returned in that case.

(iii) The parameter `Family` selects sublists of the main list to search. Possible values of this parameter are `"All"`, `"PSL"`, `"PSL2"`, `"Mathieu"`, `"Alt"`, `"PSp"`, `"PSU"`, `"Other"`, and `"notPSL2"`; sets of these strings are also allowed, which searches on the union of the appropriate sublists.

(iii) A process version of `SimpleQuotients` creates a restartable version of the computation so that it returns a result to the user each time a new epimorphism is found. It involves three intrinsics:

- `SimpleQuotientProcess` has the same arguments as `SimpleQuotient`. It initialises the process and returns a name P for future reference to the process.
- `NextSimpleQuotient` (P), where P is the process name continues the search until the next epimorphism is found whereupon it is returned and the process is suspended until the next call.
- `IsEmptySimpleQuotientProcess` (P), where P is the process name, returns `false` when the search has been completed and `true` otherwise.

Full details can be found in chapter *Finitely Presented Groups*.

Example: Simple Quotients of a Perfect Group `(ex-af2186)`

The following finitely presented group is perfect so a search is made for simple quotients. The first search is for simple quotients of order not exceeding 10^5 and degree not exceeding 100.

```
> G := FPGroup<a, b, c | a^13, b^3, c^2,a = b*c>;
> IsPerfect(G);
true
> L := SimpleQuotients(G, 1, 100, 2, 10^5 : Limit := 2);
> #L;
2
> for x in L do CompositionFactors(Image(x[1])); end for;
    G
    |  A(1, 13)               = L(2, 13)
    1
    G
    |  A(2, 3)                = L(3, 3)
    1
> L[2,1];
Homomorphism of GrpFP: F into GrpPerm: $, Degree 13, Order
2^4 * 3^3 * 13 induced by
F.1 |--> (1, 10, 4, 5, 11, 8, 3, 6, 7, 12, 9, 13, 2)
F.2 |--> (2, 10, 4)(3, 6, 7)(5, 11, 13)(8, 12, 9)
F.3 |--> (1, 10)(2, 5)(3, 12)(8, 13)
> #L[2];
2
```

The simple groups L(2,13) and L(3,3) are quotients, with two inequivalent homomorphisms onto the second group. The process mechanism will be used to search for projective unitary groups of order up to 10^6 and having permutation representations of degree not greater then 100.

```
> P := SimpleQuotientProcess(G, 1, 100, 2, 10^6 : Family:="PSU");
> IsEmptySimpleQuotientProcess(P);
false
> eps, info := SimpleEpimorphisms(P);
> info;
<65, 62400, PSU(3, 4)>
```

The group PSU(3,4) of order 62400 and degree 65 is a quotient of G. The process is restarted.

```
> NextSimpleQuotient(~P);
> IsEmptySimpleQuotientProcess(P);
true
```

There are no more unitary group quotients within the given degree and order limits.

79.7.6 L_2-Quotients

Given an fp-group G, the L_2-quotient algorithm of Plesken, Fabianska, and Jambor computes all quotients of G which are isomorphic to the 2-dimensional linear groups $PSL(2,q)$ or $PGL(2,q)$ simultaneously for any prime power q. The algorithm can handle the case of infinitely many quotients, and also works for very large prime powers. In practice it is fast when the number of generators is small and can be used to show that many fp-groups are infinite. While the simple groups $PSL(2,q)$ represent just one dimension in one family out of 16 families of simple groups, in a given range of orders, they are very numerous compared to other simple groups. See [Jam12] or [Jam15] for more information about the L_2-quotient algorithm.

In this chapter only part of the L_2-quotient machinery is discussed while computing L_3- and U_3-quotients is not discussed at all. The reader is referred to chapter *Finitely Presented Groups* for a complete treatment of the $L_2/L_3/U_3$-quotient machinery.

The term L_2-quotient will include both of the groups $PSL(2,q)$ and $PGL(2,q)$. L_2-quotients have their own Magma type: `L2Quotient`. Some fp-groups G have infinitely many L_2-quotients and this is indicated by one of the L_2-quotient names `L_2(infty^k)`, `L_2(p^(infty^d))`, or `L_2(infty^(infty^d))`.

Intrinsics:

`L2Quotients` (G) : Given a fp group G, a sequence containing all L_2-quotients is returned.

`GetMatrices` (Q) : For a finite L_2-quotient Q of G, (that is, a quotient $L_2(p^k)$ or $\mathrm{PGL}(2,p^k)$), this intrinsic returns a matrix group H and a sequence A of 2×2 matrices in H, where $A[i]$ corresponds to the i-th generator of G. Note that $G \to H$, $G.i \to A[i]$ does not in general define a homomorphism, but the induced map $G \to H/Z(H)$ does.

Notes:

(i) At the moment the algorithm does not return images onto the groups $PSL(2,2)$, $PSL(2,3)$, and $PSL(2,4) = PSL(2,5)$.

(ii) Infinite quotients should be interpreted as follows:

- **Quotients of type** $\mathrm{L}_2(\infty^k)$: If G has a quotient $\mathrm{L}_2(\infty^k)$, then for almost all (all but finitely many) primes p, G has finitely many quotients of type $\mathrm{PSL}(2,p^r)$ or $\mathrm{PGL}(2,p^{r/2})$ with $r \leq k$. So $\mathrm{L}_2(\infty^k)$ is a mnemonic, where ∞ in the base stands for infinitely many primes, and k stands for the highest possible exponent.
- **Quotients of type** $\mathrm{L}_2(p^{\infty^d})$: If G has a quotient $\mathrm{L}_2(p^{\infty^d})$, then there are infinitely many positive integers k such that G has a quotient of type $\mathrm{PSL}(2,p^k)$ or $\mathrm{PGL}(2,p^k)$. So $\mathrm{L}_2(p^{\infty^d})$ is a mnemonic, where ∞ in the exponent stands for infinitely many possible exponents. The parameter d describes the degree of infinity, and is omitted if $d = 1$.
- **Quotients of type** $\mathrm{L}_2(\infty^{\infty^d})$: If G has a quotient $\mathrm{L}_2(\infty^{\infty^d})$, then for almost all primes p and infinitely many positive integers k, G has a quotient of type $\mathrm{PSL}(2,p^k)$ or $\mathrm{PGL}(2,p^k)$. So $\mathrm{L}_2(\infty^{\infty^d})$ is a mnemonic, where ∞ in the base stands for infinitely many primes, and ∞ in the exponent stands for infinitely many possible exponents. The parameter d describes the degree of infinity, and is omitted if $d = 1$. These quotients can be further investigated using the methods `AddGroupRelations`, `AddRingRelations`, and `SpecifyCharacteristic`.

Example: L2-Quotients of FP-Groups `(ex-5313fb)`

The L_2-quotients of a certain finitely-presented group will be computed.

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^7, (a,b)^11 >;
> L2Quotients(G);
[
    L_2(43)
]
> H := FPGroup< a,b,c | a^3, b^7, c^19, (a*b)^2, (a*c)^2, (b*c)^2, (a*b*c)^2 >;
> L2Quotients(H);
[
    L_2(113)
]
```

This means that G has $\mathrm{PSL}(2,43)$ as quotient, but no other $\mathrm{PSL}(2,q)$ or $\mathrm{PGL}(2,q)$ is a quotient of G. Similarly, the only L_2-quotient of H is $\mathrm{PSL}(2,113)$.

The next example has more quotients:

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^16, (a,b)^11 >;
> L2Quotients(G);
[
    PGL(2,23),
    PGL(2,23),
    PGL(2,463)
]
```

Here $\mathrm{PGL}(2,23)$ occurs twice. This means that there are two epimorphisms of G onto $\mathrm{PGL}(2,23)$

which do not differ by an automorphism of $\mathrm{PGL}(2, 23)$. In other words, the kernels of the epimorphisms are distinct.

Example: Groups with Infinitely Many L2-Quotients (ex-0eaade)

Some groups have infinitely many L_2-quotients. This is indicated by one of the L_2-quotients `L_2(infty^k)`, `L_2(p^(infty^d))`, or `L_2(infty^(infty^d))`, as the follow examples illustrate.

```
> G := FPGroup< a,b,c | a^3, b^7, (a*b)^2, (a*c)^2, (b*c)^2, (a*b*c)^2 >;
> L2Quotients(G);
[
    L_2(infty^6)
]
> H := FPGroup<a,b,c | a^3, (a,c)=(c,a^-1), a*b*a=b*a*b, a*b*a*c^-1=c*a*b*a>;
> L2Quotients(H);
[
    L_2(3^infty)
]
> K := FPGroup< a,b | a^3*b^3 >;
> L2Quotients(K);
[
    L_2(infty^infty)
]
```

Example: Matrix Groups of an L2-Quotient (ex-ba85d0)

The following example illustrates how the quotient matrix groups can be accessed.

```
> H := FPGroup< a,b,c | a^3, b^7, c^19, (a*b)^2, (a*c)^2, (b*c)^2, (a*b*c)^2 >;
> quot := L2Quotients(H); quot;
[
    L_2(113)
]
> H, A := GetMatrices(quot[1]);
> H;
MatrixGroup(2, GF(113))
Generators:
    [  0   1]
    [112 112]

    [  0  85]
    [109  24]

    [102 104]
    [ 63  72]
> A;
[
    [  0   1]
    [112 112],

    [  0  85]
    [109  24],

    [102 104]
    [ 63  72]
]
```

79.8 Bibliography

- [AR84] D. G. Arrell and E. F. Robertson. A modified Todd-Coxeter algorithm. In *Computational group theory (Durham, 1982)*, pages 27–32. Academic Press, London, 1984.
- [HKRR84] George Havas, P. E. Kenne, J. S. Richardson, and E. F. Robertson. A Tietze transformation program. In *Computational group theory (Durham, 1982)*, pages 69–73. Academic Press, London, 1984.
- [HEO05] D. F. Holt, B. Eick, and E. O`Brien. *Handbook of Computational Group Theory*. Chapman and Hall/CRC, 2005.
- [Jam12] Sebastian Jambor. *An L3-U3-quotient algorithm for finitely presented groups*. PhD Thesis, RWTH Aachen University, 2012.
- [Jam15] Sebastian Jambor. An L_2-quotient algorithm for finitely presented groups on arbitrarily many generators. *J. Algebra*, 423:1109–1142, 2015. URL: https://doi.org/10.1016/j.jalgebra.2014.08.058, doi:10.1016/j.jalgebra.2014.08.058.
- [MKS76] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial group theory*. Dover Publications Inc., New York, revised edition, 1976. Presentations of groups in terms of generators and relations.
- [NO96] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.
- [Nic96] Werner Nickel. Computing nilpotent quotients of finitely presented groups. In *Geometric and computational perspectives on infinite groups (Minneapolis, MN and New Brunswick, NJ, 1994)*, pages 175–191. Amer. Math. Soc., Providence, RI, 1996.
- [Sim94] Charles C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

Chapter 80

Finitely Presented Groups

80.1 Introduction

This Chapter documents the functions designed for computing with finitely-presented groups (fp-groups for short). The name of the corresponding Magma category is `GrpFP`. The intrinsics considered here are designed for computations in the area that is sometimes referred to as *combinatorial group theory*.

80.1.1 Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings:

- The construction of fp-groups in terms of generators and relations
- The simplification of presentations using Tietze transformations
- The Todd-Coxeter procedure for constructing the coset table of a subgroup. This is the basis for many fp-group algorithms
- A collection of tools for computing with subgroups having (modest) finite index in a group, where the subgroups are represented by coset tables
- The construction of all subgroups having index less than some (small) specified bound
- The Reidemeister-Schreier algorithm for constructing presentations of subgroups
- The determination of various properties of fp-groups such as *perfect*, *finite/infinite*, *small cancellation*, and *large*
- A search for homomorphisms of an fp-group onto a permutation group or a pc-group
- A test for the isomorphism of two fp-groups
- The construction of particular types of quotient groups: abelian quotient, p-quotient, nilpotent quotient, soluble quotient, simple group quotient, and $L_2/L_3/U_3$ quotients
- The construction of representations of an fp-group corresponding to actions on coset spaces and elementary abelian sections

This chapter contains detailed information about the intrinsics provided for general fp-groups. The much shorter chapter *Introduction to Fp-Groups*, "Introduction of Finitely Presented Groups", gives a basic description of the more important intrinsics for fp-groups and should serve to get users started. That chapter typically does not list the parameters of an intrinsic and omits the more specialised intrinsics which are mainly useful when generic intrinsics fail.

Free groups are treated in Chapter *Free Groups* and it should be read in conjunction with this chapter. Automatic groups and hyperbolic groups are discussed in Chapter *Automatic and Hyperbolic Groups*.

For a description of the fundamental algorithms for finitely presented groups, the reader should consult Sims [Sim94] or Holt [HEO05].

80.2 Construction of an FP-Group

An fp-group is normally constructed either by giving a presentation in terms of generators and relations or it is defined to be a subgroup or quotient group of an existing fp-group. However, fp-groups may be also created from finite permutation and matrix groups. Finally, Magma has separate types for a number of families of fp-groups that satisfy a condition such as being polycyclic. Again functions are provided to convert a member of one of these families into a general fp-group.

80.2.1 Introduction

The construction of fp-groups utilises the fact that every group is a quotient of some free group. Thus, two general fp-group constructors are provided: *FreeGroup*(n) which constructs a free group of rank n, and *quo< F | R >*< F | R > which constructs the quotient of group F by the normal subgroup defined by the relations R.

The naming of generators presents special difficulties since they are not always used in a consistent manner in the mathematical literature. A generator name is used in two distinct ways. Firstly, it plays the role of a *variable* having as its value a designated generator of G. Secondly, it appears as the *symbol* designating the specified generator whenever elements of the group are output. These two uses are separated in the Magma semantics. In Magma, a standard indexing notation is provided for referencing the generators of any fp-group G. Thus, G.i denotes the i-th generator of G. However, users may give individual names to the generators by means of the *generator*-assignment. Suppose that the group G is defined on r generators. Then if the right hand side of the following statement creates a group, the special assignment

```
> G< v_1, ..., v_r> := \construction;
```

is equivalent to the statements

```
> G := \construction;
>    v_1 := G.1;
>    ...
>    v_r := G.r;
```

It should be noted that when the fp-group G is created as the quotient of the group F, any names that the user may have associated with the generators of F will not be associated with the corresponding generators of G. If this were allowed, then it would violate the fundamental principle that every object is viewed as belonging to a *unique* structure.

80.2.2 Quotient Group Constructor

A group with non-trivial relations is constructed as a quotient of an existing group, usually a free group. For convenience, the necessary free group may be constructed in-line.

quo< F | R >: GrpFP, List -> GrpFP, Hom(Grp)

Given an fp-group F, and a set of relations R in the generators of F, construct the quotient G of F by the normal subgroup of F defined by R. The group G is defined by means of a presentation which consists of the relations for F (if any), together with the additional relations defined by the list R. The expression defining F may be either simply the name of a previously constructed group, or an expression defining an fp-group. If R is a list then each term of the list is either a *word*, a *relation*, a *relation list*, or a *subgroup* of F.

A *word* is interpreted as a relator.

A *relation* consists of a pair of words, separated by '='. (See above.)

A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=': $w_1 = w_2 = \cdots = w_r$. This is interpreted as the set of relations $w_1 = w_r, \ldots, w_{r-1} = w_r$. Note that the relation list construct is only meaningful in the context of the quo-constructor.

A *subgroup* H appearing in the list R contributes its generators to the relation set for G, i.e. each generator of H is interpreted as a relator for G. The group F may be referred to by the special

symbol $ in any word appearing to the right of the '|' symbol in the quo-constructor. Also, in the context of the quo-constructor, the identity element may be represented by the digit 1.

The function returns:

(a) The quotient group G;

(b) The natural homomorphism $\phi : F \to G$.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*.

`G / H: GrpFP, GrpFP -> GrpFP`

Given a subgroup H of the group G, construct the quotient of G by the normal closure N of H. The quotient is formed by taking the presentation for G and including the generating words of H as additional relators.

Example: Sym(4) as a Quotient of a Free Group (ex-94ed98)

The symmetric group of degree 4 may be represented as a two generator group with presentation $< a, b \mid a^2, b^3, (ab)^4 >$. Giving the relations as a list of relators, the presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi := quo< F | a^2, b^3, (a*b)^4 >;
```

Alternatively, giving the relations as a relations list, the presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi :=  quo< F | a^2 = b^3 = (a*b)^4 = 1>;
```

Finally, giving the relations in the form of a set of relations, this presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> rels := { a^2, b^3, (a*b)^4 };
> G<x, y>, phi :=  quo< F | rels >;
```

Example: Quotient Constructor Without a Free Group (ex-803aa2)

A group may be defined using the *quo< F | R >*-constructor without first assigning a free group. The $ symbol is used to reference the group whose quotient is being formed.

```
> S4<x, y> := quo< FreeGroup(2) | $.1^2, $.2^3, ($.1*$.2)^4 >;
> S4;
Finitely presented group S4 on 2 generators
Relations
      x^2 = Id(S4)
      y^3 = Id(S4)
      (x * y)^4 = Id(S4)
```

Example: Quotient of a Triangle Group (ex-8457bf)

We illustrate the use of the *quo< F | R >*-constructor in defining the quotient of a group other than a free group.

```
> F<x, y> := FPGroup< x, y | x^2 = y^3 = (x*y)^7 = 1 >;
> F;
Finitely presented group F on 2 generators
Relations
```

(continues on next page)

(continued from previous page)

```
      x^2 = Id(F)
      y^3 = Id(F)
      (x * y)^7 = Id(F)
> G<a, b> := quo< F | (x, y)^8 >;
> G;
Finitely presented group G on 2 generators
Relations
      a^2 = Id(G)
      b^3 = Id(G)
      (a * b)^7 = Id(G)
      (a, b)^8 = Id(G)
> Order(G);
10752
```

80.2.3 The FP-Group Constructor

For convenience, a constructor is provided which allows the user to define an fp-group in a single step.

FPGroup< X | R >: List(Var), List(GrpFPRel) -> GrpFP, Hom(Grp)

Given a list X of variables $x_1, \ldots, x_r$, and a list of relations R over these generators, first construct the free group F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the normal subgroup of F defined by the relations R. The syntax for the relations R is the same as for the *quo< F | R >*-constructor. The function returns:

(a) The quotient group G;

(b) The natural homomorphism $\phi : F \to G$.

Example: Binary Tetrahedral Group (ex-388271)

We illustrate the *FPGroup< X | R >*-constructor by defining the binary tetrahedral group in terms of the presentation $< r, s \mid r^3 = s^3 = (rs)^2 >$:

```
> G<r, s> := FPGroup< r, s | r^3 = s^3 = (r*s)^2 >;
```

Example: Group Generated by Three Involutions (ex-cb4daf)

Again, using the *FPGroup< X | R >*-constructor, the group $< r, s, t \mid r^2, s^2, t^2, rst = str = trs >$ would be specified as:

```
> G<r, s, t> := FPGroup<r, s, t | r^2, s^2, t^2, r*s*t = s*t*r = t*r*s>;
```

Example: Coxeter's Parametrised Families of Groups (ex-4c068c)

In our final example we illustrate the use of functions to represent parametrised families of groups. In the notation of Coxeter, the symbol $(l, m|n, k)$ denotes the family of groups having presentation

$$< a, b \mid a^l, b^m, (a * b)^n, (a * b^{-1})^k > .$$

```
> Glmnk := func< l, m, n, k | FPGroup< a, b | a^l, b^m, (a*b)^n, (a*b^-1)^k > >;
> G<a, b> := Glmnk(3, 3, 4, 4);
> G;
Finitely presented group G on 2 generators
Relations
      a^3 = Id(G)
      b^3 = Id(G)
      (a * b)^4 = Id(G)
      (a * b^-1)^4 = Id(G)
```

(continues on next page)

(continued from previous page)

```
> Order(G);
168
> G<a, b> := Glmnk(2, 3, 4, 5);
> G;
Finitely presented group G on 2 generators
Relations
    a^2 = Id(G)
    b^3 = Id(G)
    (a * b)^4 = Id(G)
    (a * b^-1)^5 = Id(G)
> Order(G);
1
```

Thus $(2, 3 \mid 4, 5)$ is the trivial group.

80.2.4 Accessing the Defining Generators and Relations

The functions in this group provide access to basic information stored for a finitely-presented group G.

`G . i: GrpFP, RngIntElt -> GrpFPElt`

The i-th defining generator for G. A negative subscript indicates that the inverse of the generator is to be created. The generator `G.0` is `Identity(G)`, the empty word in G.

`Generators(G): GrpFP -> { GrpFPElt }`

A set containing the generators for the group G.

`NumberOfGenerators(G): GrpFP -> RngIntElt`

`Ngens(G): GrpFP -> RngIntElt`

The number of generators for the group G.

`PresentationLength(G): GrpFP -> RngIntElt`

The total length of the relators for G.

`Relations(G): GrpFP -> [ GrpFPRel ]`

A sequence containing the defining relations for G.

80.2.5 Operations on Words

The functions described in this section perform low level string operations like substitution, elimination, or substring matching on elements of fp-groups.

`Eliminate(u, x, v): GrpFPElt, GrpFPElt, GrpFPElt -> GrpFPElt`

Given words u and v, and a generator x, all belonging to a group G, return the word obtained from u by replacing each occurrence of x by v and each occurrence of x^{-1} by v^{-1}.

`Eliminate(U, x, v): { GrpFPElt }, GrpFPElt, GrpFPElt -> { GrpFPElt }`

Given a set of words U, a word v, and a generator x, all belonging to a group G, return the set of words obtained by taking each element u of U in turn, and replacing each occurrence of x in u by v and each occurrence of x^{-1} by v^{-1}.

`Match(u, v, f): GrpFPElt, GrpFPElt, RngIntElt -> BoolElt, RngIntElt`

Suppose u and v are words belonging to the same group G, and that f is an integer such that $1 \leq f \leq \#u$. The function seeks the least integer l such that:

(a) $l \geq f$; and

(b) v appears as a subword of u, starting at the l-th letter of u.

If such an integer l is found `Match` returns the value `true` and l. If no such l is found, `Match` returns the value `false`.

RotateWord(u, n): GrpFPElt, RngIntElt -> GrpFPElt

The word obtained by cyclically permuting the word u by n places. If n is positive, the rotation is from left to right, while if n is negative the rotation is from right to left. In the case where n is zero, the function returns u.

Substitute(u, f, n, v): GrpFPElt, RngIntElt, RngIntElt, GrpFPElt -> GrpFPElt

Given words u and v belonging to a group G, and non-negative integers f and n, this function replaces the substring of u of length n, starting at position f, by the word v. Thus, if $u = x_{i_1}^{e_1} \cdots x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}} \cdots x_{i_m}^{e_m}$ then the substring $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$ is replaced by v. If the function is invoked with $v = \mathrm{Id(G)}$, then the substring $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$ of u is deleted.

Subword(u, f, n): GrpFPElt, RngIntElt, RngIntElt -> GrpFPElt

The subword of the word u comprising the n consecutive letters commencing at the f-th letter of u.

Example: Operations on Words in a Free Group (ex-1c9bc2)

We demonstrate some of these operations in the context of the free group on generators x, y, and z.

```
> F<x, y, z> := FreeGroup(3);
> u := (x, y*z);
> w := u^(x^2*y);
> #w;
12
> w;
y^-1 * x^-3 * z^-1 * y^-1 * x * y * z * x^2 * y
```

We replace each occurrence of the generator x in w by the word $y * z^-1$.

```
> Eliminate(w, x, y*z^-1);
y^-1 * z * y^-1 * z * y^-1 * z * y^-1 * z^-2 * y * z * y * z^-1 * y * z^-1 * y
```

We count the number of occurrences of each generator in w.

```
> [ ExponentSum(w, F.i) : i in [1..Ngens(F)] ];
[ 0, 0, 0 ]
> GeneratorNumber(w);
-2
```

We locate the start of the word u in the word w.

```
> b, p := Match(w, u, 1);
> b, p;
true 4
```

We now replace the subword u in w by the word $y * x$.

```
> t := Substitute(w, p, #u, y*x);
> t;
y^-1 * x^-2 * y * x^3 * y
```

We create the set of all distinct cyclic permutations of the word u.

```
> rots := { RotateWord(u, i) : i in [1 ..#u] };
> rots;
{ y^-1 * x * y * z * x^-1 * z^-1,  x * y * z * x^-1 * z^-1 * y^-1,
x^-1 * z^-1 * y^-1 * x * y * z,  z * x^-1 * z^-1 * y^-1 * x * y,
z^-1 * y^-1 * x * y * z * x^-1,  y * z * x^-1 * z^-1 * y^-1 * x }
```

80.2.6 Operations on Presentations

The functions described in this section construct a new fp-group from an existing one by adding or deleting a generator or by adding, deleting, or changing a relation. The new group is created without any relationship to the existing group.

`AddGenerator(G): GrpFP -> GrpFP`

Given an fp-group G with presentation $\langle X \mid R\rangle$, create a new fp-group with presentation $\langle X \cup \{z\} \mid R\rangle$, where z is a symbol not in X.

`AddGenerator(G, w): GrpFP, GrpFPElt -> GrpFP`

Given an fp-group G with presentation $\langle X \mid R\rangle$, and given also a word w in the generators X, create a new fp-group having the presentation $\langle X \ \cup \{z\} \mid R \cup \{z = w\}\rangle$, where z is a symbol not in X.

`AddRelation(G, r): GrpFP, RelElt -> GrpFP`

Given an fp-group G, and a relation r on the generators of G, create a new fp-group whose presentation consists of the relations of G together with the relation r.

`AddRelation(G, g): GrpFP, GrpFPElt -> GrpFP`

Given an fp-group G, and an element g of G, create a new fp-group whose presentation consists of the relations of G together with the relation $g = \mathrm{Id}(G)$.

`AddRelation(G, r, i): GrpFP, RelElt, RngIntElt -> GrpFP`

Given an fp-group G, and a relation r on the generators of G, create a new fp-group which has as its presentation the relations of G together with the relation r inserted after the i-th existing relation of G.

`AddRelation(G, g, i): GrpFP, GrpFPElt, RngIntElt -> GrpFP`

Given an fp-group G, and an element g of G, create a new fp-group which has as its presentation the relations of G together with the relation $g = \mathrm{Id}(G)$ inserted after the i-th existing relation of G.

`DeleteGenerator(G, x): GrpFP, GrpFPElt -> GrpFP`

Given an fp-group G with presentation $\langle X \mid R\rangle$, and given also an element z in X, create a new fp-group with presentation $\langle X \setminus \{z\} \mid R'\rangle$, where the relations R' are obtained from R by deleting all relations containing an occurrence of z.

`DeleteRelation(G, r): GrpFP, RelElt -> GrpFP`

Given an fp-group G, which includes the relation r amongst its relations, create a new fp-group which has as its presentation the relations of G with relation r omitted.

`DeleteRelation(G, g): GrpFP, GrpFPElt -> GrpFP`

Given an fp-group G, which includes the relation $g = \mathrm{Id}(G)$ amongst its relations, create a new fp-group which has as its presentation the relations of G with this relation omitted.

`DeleteRelation(G, i): GrpFP, RngIntElt -> GrpFP`

Given an fp-group G, create a new fp-group which has as its presentation the relations for G with the i-th relation deleted.

`ReplaceRelation(G, s, r): GrpFP, RelElt, RelElt -> GrpFP`

`ReplaceRelation(G, h, r): GrpFP, GrpFPElt, RelElt -> GrpFP`

`ReplaceRelation(G, s, g): GrpFP, RelElt, GrpFPElt -> GrpFP`

`ReplaceRelation(G, h, g): GrpFP, GrpFPElt, GrpFPElt -> GrpFP`

Given an fp-group G, which includes the relation s or $h = \mathrm{Id}(G)$ amongst its relations, create a new fp-group which has as its presentation the relations for G with the relation s replaced by the relation r or $g = \mathrm{Id}(G)$.

`ReplaceRelation(G, i, r): GrpFP, RngIntElt, RelElt -> GrpFP`

Given an fp-group G and a relation r in the generators of G, create a new fp-group which has as its presentation the relations for G with relation number i replaced by the relation r.

`ReplaceRelation(G, i, g): GrpFP, RngIntElt, GrpFPElt -> GrpFP`

Given an fp-group G and an element g of G, create a new fp-group which has as its presentation the relations for G with relation number i replaced by the relation $g = \mathrm{Id}(G)$.

Example: Replacing a Relation (ex-c80efd)

We use the function *ReplaceRelation* to vary a particular relation in a presentation. The order of the resulting group together with the index of a particular subgroup is determined.

```
> G<x,y,z,h,k,a> := FPGroup< x, y, z, h, k, a |
>    x^2, y^2, z^2, (x,y), (y,z), (x,z), h^3, k^3, (h,k),
>    (x,k), (y,k), (z,k), x^h*y, y^h*z, z^h*x, a^2, a*x*a*y,
>    a*y*a*x, (a,z), (a,k), (a*h)^2 >;
> for i := 0 to 1 do
>     for j := 0 to 1 do
>         for k := 0 to 1 do
>              for l := 0 to 2 do
>                 rel := G.1^i*G.2^j*G.3^k*G.5^l*(G.6*G.4)^2 = Id(G);
>                 K := ReplaceRelation(G, 21, rel);
>                 print Order(K), Index(K, sub< K | K.6, K.4>);
>             end for;
>         end for;
>     end for;
> end for;

<0, 0, 0, 0> 144 24
<0, 0, 0, 1> 144 8
<0, 0, 0, 2> 144 8
<0, 0, 1, 0> 18 3
<0, 0, 1, 1> 18 1
<0, 0, 1, 2> 18 1
<0, 1, 0, 0> 72 3
<0, 1, 0, 1> 72 1
<0, 1, 0, 2> 72 1
<0, 1, 1, 0> 36 6
<0, 1, 1, 1> 36 2
<0, 1, 1, 2> 36 2
<1, 0, 0, 0> 18 3
<1, 0, 0, 1> 18 1
<1, 0, 0, 2> 18 1
<1, 0, 1, 0> 144 6
<1, 0, 1, 1> 144 2
<1, 0, 1, 2> 144 2
<1, 1, 0, 0> 36 6
<1, 1, 0, 1> 36 2
<1, 1, 0, 2> 36 2
<1, 1, 1, 0> 72 12
<1, 1, 1, 1> 72 4
<1, 1, 1, 2> 72 4
```

80.3 Simplification

Simplification may be achieved by setting some parameters and trusting that the built-in strategy will produce the result desired. Alternatively the user may create a Tietze process allowing the user to interactively perform the simplification.

80.3.1 Automatic Simplification

Subgroups of finitely presented groups constructed in certain ways may be have a generating set containing redundant generators. The most important case in which this situation may occur is a subgroup of the domain of a homomorphism f of groups, defined as the preimage under f of some given subgroup of the codomain of f. In this case, the generating set of the preimage will contain generators of the kernel of f and is likely to be not minimal.

Since reducing the generating set may be expensive and is not necessary in all situations, a reduction is not done automatically. Instead, Magma provides a function for reducing generating sets of finitely presented groups.

`ReduceGenerators(G): GrpFP -> GrpFP, Map`

Given a finitely presented group G, attempt to construct a presentation H on fewer generators. H is returned as a subgroup of G (which of course is equal to G), so that element coerce is possible. The isomorphism from G to H is returned as second return value.

If a presentation for G is known, this function attempts to simplify this presentation (cf. section *Interactive Simplification*). Otherwise, it tries to rewrite G with respect to a suitable supergroup to obtain a presentation on fewer generators.

For a sample application of this function, see Example *KG-Modules of a Finitely Presented Group*.

Given a finitely presented group G, the user can attempt to simplify its presentation using Tietze transformations and substring searching. The choice of simplification strategy can either be left to Magma or selected by the user.

`Simplify(G: parameters): GrpFP -> GrpFP, Map`

```
Preserve          : [RngIntElt]                 Default: []
Iterations        : RngIntElt                   Default: 10000
EliminationLimit  : RngIntElt                   Default: 100
LengthLimit       : RngIntElt                   Default: Infinity()
ExpandLimit       : RngIntElt                   Default: 150
GeneratorsLimit   : RngIntElt                   Default: 0
SaveLimit         : RngIntElt                   Default: 10
SearchSimultaneous: RngIntElt                   Default: 20
Print             : RngIntElt                   Default: 0
```

Given a finitely presented group G, attempt to simplify the presentation of G by repeatedly eliminating generators and subsequently shortening relators by substitution of substrings that correspond to the left or right hand side of a relation. The order in which transformations are applied is determined by a set of heuristics. The transformation process terminates when no more eliminations of generators and no more length reducing substring replacements are possible. A new group K isomorphic to G is returned as a subgroup of G which is (hopefully) defined by a simpler presentation than G. The isomorphism $f : G \to K$ is returned as second return value.

The simplification process can be controlled by a set of parameters described below.

The strategy employed by the function *Simplify* can be controlled using the following set of parameters.

This parameter can be used to indicate that certain generators of G should not be eliminated (*default*: no restrictions). `Preserve` is assigned a sequence of integers in the range $[1, \ldots, n]$, where n is the number of generators of G, containing the numbers of those generators of G which should be preserved. See Example *Reducing a Generating Set* for a sample application.

This parameter sets the maximal number of iterations of the main elimination / simplification cycle which will be performed.

This parameter sets the maximal number of generators which may be eliminated in each elimination phase.

If `LengthLimit` is set to n, any eliminations which would make the total length of the relators grow beyond n will not be performed (*default*: no limit).

If `ExpandLimit` is set to n, the total length of the relators is not permitted to grow by more than a factor of $n\%$ in any elimination phase (*default*: 150%). If this limit is reached, the elimination phase is aborted.

Any eliminations which would reduce the number of generators below the value of `GeneratorsLimit` will not be performed (*default*: 0).

If `SaveLimit` is set to n, any simplification phase is repeated, if the reduction in the total length of the relators achieved during this phase exceeds $n\%$ (*default*: 10%).

This parameter sets the number of relators processed simultaneously in a simplification phase.

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose(s): MonStgElt → RngIntElt("Tietze")`, which is 0 unless it has been changed through use of `SetVerbose(s, i): MonStgElt, RngIntElt`.

Example: Simplifying the Fibonacci Group F(8) (ex-6d0638)

Consider the Fibonacci group $F(8)$.

```
> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>     quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>              x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;
```

We use the function *Simplify* in order to obtain a presentation of $F(8)$ on two generators.

```
> H<[y]>, f := Simplify(F8);
> H;
Finitely presented group H on 2 generators
Generators as words in group F8
    y[1] = x3
    y[2] = x4
Relations
    y[2] * y[1]^-2 * y[2] * y[1]^-1 * y[2]^2 * y[1] * y[2]^2 *
       y[1]^-1 = Id(H)
    y[1] * y[2] * y[1] * y[2]^2 * y[1] * y[2] * y[1]^2 * y[2]^-1
       * y[1] = Id(H)
```

The isomorphism f can be used to express the old generators in terms of the new ones.

```
> f;
Mapping from: GrpFP: F8 to GrpFP: H
> f(x1);
y[1]^2 * y[2]^-1
```

`SimplifyLength(G: parameters): GrpFP -> GrpFP, Map`

Given a finitely presented group G, attempt to eliminate generators and shorten relators by locating substrings that correspond to the left or right hand side of a relation. The order in which transformations are applied is determined by a set of heuristics. As opposed to the function *Simplify*, this function terminates the transformation process when the total length of the presentation starts to increase with the elimination of further generators.

A new group K isomorphic to G is returned which is (hopefully) defined by a simpler presentation than G. K is returned as a subgroup of G. The isomorphism $f : G \to K$ is returned as second return value. This function accepts the same set of parameters as the function *Simplify*.

80.3.2 Interactive Simplification

The user may exercise fine control over the simplification of a presentation by creating a Tietze process which allows the user to look at the result of a simplification operation before specifying the next step.

TietzeProcess(G: parameters): GrpFP -> Process(Tietze)

```
Preserve          : [RngIntElt]                 Default: []
Iterations        : RngIntElt                   Default: 10000
EliminationLimit  : RngIntElt                   Default: 100
LengthLimit       : RngIntElt                   Default: Infinity()
ExpandLimit       : RngIntElt                   Default: 150
GeneratorsLimit   : RngIntElt                   Default: 0
SaveLimit         : RngIntElt                   Default: 10
SearchSimultaneous: RngIntElt                   Default: 20
Print             : RngIntElt                   Default: 0
```

Create a Tietze process that takes the presentation for the fp-group G as its starting point. This process may now be manipulated by various procedures that will be described below.

These parameters define the defaults used for the Tietze operations. Each of the various procedures described below allows some or all of these control parameters to be overridden.

For the meanings of the parameters, see the description under *Simplify* above.

ShowOptions(~P : parameters): GrpFPTietzeProc

Display the defaults associated with the Tietze process P. The current status of all the control parameters may be viewed by using this function.

SetOptions(~P : parameters): GrpFPTietzeProc

Change the defaults associated with the Tietze process P. All of the control parameters may be overridden permanently by using this function.

Simplify(~P : parameters): GrpFPTietzeProc

SimplifyPresentation(~P : parameters): GrpFPTietzeProc

Use the default strategy to simplify the presentation as much as possible. The transformation process is terminated when no more eliminations of generators and no more length reducing substring replacements are possible. All the control parameters may be overridden for this function.

SimplifyLength(~P : parameters): GrpFPTietzeProc

Use the default strategy to simplify the presentation as much as possible. The transformation process is terminated when the total length of the presentation starts to increase with the elimination of further generators. All the control parameters may be overridden for this function.

Eliminate(~P: parameters): GrpFPTietzeProc

EliminateGenerators(~P: parameters): GrpFPTietzeProc

```
Relator  : RngIntElt                    Default: 0
Generator: RngIntElt                    Default: 0
```

Eliminate generators in the presentation defined by the Tietze process P under the control of the parameters. First any relators of length one are used to eliminate trivial generators. Then, if there are any non-involutory relators of length two, the generator with the higher number is eliminated. Of the control parameters, only `EliminationLimit, ExpandLimit, GeneratorsLimit`, and `LengthLimit` may be overridden by this function.

If $n > 0$, try to eliminate a generator using the n-th relator. If no generator is specified by the parameter `Generator` below, then the generator which is eliminated will be the one occurring once in the relator that produced the smallest total relator length.

If $n > 0$, try to eliminate the n-th generator. If no relation is specified by the parameter `Relator` above, then the shortest relator in which the n-th generator occurs exactly once (if any) is used.

Search(~P: parameters): GrpFPTietzeProc

Simplifies the presentation by repeatedly searching for common substrings in pairs of relators where the length of the substring is greater than half the length of the shorter relator and making the corresponding transformations. Relators of length 1 or 2 are also used to generate simplifications. The control parameters **SaveLimit** and **SearchSimultaneous** can be overridden.

SearchEqual(~P: parameters): GrpFPTietzeProc

Modifies the presentation by repeatedly searching for common substrings in pairs of relators where the length of the substring is exactly half the length of the shorter relator and making the corresponding transformations. The control parameter **SearchSimultaneous** can be overridden.

Group(P): GrpFPTietzeProc -> GrpFP, Map

Extract the group G defined by the current presentation associated with the Tietze process P, together with the isomorphism between the original group and G. G is returned as a subgroup of the original group underlying P.

NumberOfGenerators(P): GrpFPTietzeProc -> RngIntElt

Ngens(P): GrpFPTietzeProc -> RngIntElt

The number of generators for the presentation currently stored by the Tietze process P.

NumberOfRelations(P): GrpFPTietzeProc -> RngIntElt

Nrels(P): GrpFPTietzeProc -> RngIntElt

The number of relations in the presentation currently stored by the Tietze process P.

PresentationLength(P): GrpFPTietzeProc -> RngIntElt

The sum of the lengths of the relators in the presentation currently stored by the Tietze process P.

Example: Interactive Simplification of the Fibonacci Group F(7) (ex-1eeac7)

The Fibonacci group $F(n)$ is generated by $\{x_1, \ldots, x_n\}$ with defining relations

$$x_i x_{i+1} = x_{i+2}, i \in \{1, \ldots, n\},$$

where the subscripts are taken modulo n. Consider the Fibonacci group $F(7)$, which is defined in terms of the presentation

$$< x_1, x_2, x_3, x_4, x_5, x_6, x_7 \mid x_1x_2 = x_3, x_2x_3 = x_4, x_3x_4 = x_5, \\ x_4x_5 = x_6, x_5x_6 = x_7, x_6x_7 = x_1, x_7x_1 = x_2 > .$$

The following code will produce a 2-generator, 2-relator presentation for $F(7)$:

```
> F<x1, x2, x3, x4, x5, x6, x7> := FreeGroup(7);
> F7<x1, x2, x3, x4, x5, x6, x7> :=
>    quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>             x5*x6=x7, x6*x7=x1, x7*x1=x2 >;
> P := TietzeProcess(F7);
> for i := 7 to 3 by -1 do
>    Eliminate(~P: Generator := i);
> end for;
> Search(~P);
> H<x, y>, f := Group(P);
> H;
Finitely presented group H on 2 generators
Generators as words in group F7
    x = x1
    y = x2
Relations
    x^-1 * y^-1 * x^-1 * y^-2 * x^-1 * y * x^-1 * y * x^-1 = Id(H)
    x * y^3 * x^-1 * y * x^-1 * y^2 * x * y * x * y^-1 = Id(H)
```

The resulting presentation is

$$< a, b \mid a^{-1}b^{-1}a^{-1}b^{-2}a^{-1}ba^{-1}ba^{-1}, ab^3a^{-1}ba^{-1}b^2abab^{-1} > .$$

We can use the isomorphism f returned by the function *Group* to express arbitrary words in the original generators of $F(7)$ in terms of the new generators x and y:

```
> f;
Mapping from: GrpFP: F7 to GrpFP: H
> f(x7);
x * y^2 * x * y^2 * x * y * x * y^2 * x * y
```

Alternatively, a similar effect may be obtained using the *Simplify* function:

```
> F<x1, x2, x3, x4, x5, x6, x7> := FreeGroup(7);
> F7<x1, x2, x3, x4, x5, x6, x7> :=
>    quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>       x5*x6=x7, x6*x7=x1, x7*x1=x2>;
> H<x, y>, f := Simplify(F7: Iterations := 5);
> H;
Finitely presented group H on 2 generators
Generators as words in group F7
    x = x2
    y = x3
Relations
    x * y^-1 * x * y^2 * x * y * x^2 * y^-1 = Id(H)
    y * x * y^2 * x^-1 * y * x^-2 * y * x^-2 = Id(H)
```

Again, we can use the isomorphism f returned by the function *Simplify* to express arbitrary words in the original generators of $F(7)$ in terms of the new generators x and y:

```
> f;
Mapping from: GrpFP: F7 to GrpFP: H
> f(x7);
y * x * y * x * y^2 * x * y
```

Example: Reducing a Generating Set (ex-df6881)

In a situation where some proper subset S of the original generating set of a finitely group G is sufficient to generate G, the function *Simplify* can also be used to rewrite words in the original generators in terms of the elements of S. Consider again one of the Fibonacci groups, say $F(8)$.

```
> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>    quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>       x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;
```

Obviously, $F(8)$ is generated by x_1 and x_2. We utilise the function *Simplify* to obtain a presentation H of $F(8)$ on x_1 and x_2, using the parameter `Preserve` to indicate that x_1 and x_2 – i.e. the first and the second generator – are to be retained in the new presentation. We also compute the isomorphism $f : F(8) \to H$.

```
> H<x, y>, f := Simplify(F8: Preserve := [1,2]);
```

Mapping elements of $F(8)$ to H using the map f basically means to rewrite these elements in terms of the generators $x = x_1$ and $y = x_2$. Since H is returned as a subgroup of $F(8)$, the resulting words can be coerced from H back into $F(8)$, yielding words explicitly in x_1 and x_2.

```
> F8 ! f(x5*x6);
x1 * x2^2 * x1 * x2 * x1^2 * x2^-1 * x1 * x2^-1
```

80.4 Standard Constructions

In this section various intrinsics for constructing fp-groups are described. These are familiar groups constructed as fp-groups, extensions of existing fp-groups, and the construction of fp-groups from some other type of group.

80.4.1 Familiar Groups as FP-Groups

A number of functions are provided which construct presentations for various standard groups.

AbelianFPGroup([n_1,...,n_r]): [RngIntElt] -> GrpFP

Construct the abelian group defined by the sequence $[n_1, \ldots, n_r]$ of non-negative integers as an fp-group. The function returns the direct product of cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$, where C_0 is interpreted as an infinite cyclic group.

AlternatingFPGroup(n): RngIntElt -> GrpFP

Alt(GrpFP, n): Cat, RngIntElt -> GrpFP

Construct the alternating group of degree n as an fp-group, where the generators correspond to the permutations $(3, 4, \ldots, n)$ and $(1, 2, 3)$, for n odd, or $(1, 2)(3, 4, \ldots, n)$ and $(1, 2, 3)$, for n even.

BraidFPGroup(n): RngIntElt -> GrpFP

Construct the braid group on n strings ($n - 1$ Artin generators) as an fp-group.

CoxeterFPGroup(t): MonStgElt -> GrpFP

Construct the Coxeter group of Cartan type t as a finitely presented group, given by the standard Coxeter presentation. The Cartan type t is passed to this function as a string; we refer to Chapter Root Data for details.

CyclicFPGroup(n): RngIntElt -> GrpFP

Construct the cyclic group of order n as an fp-group.

DihedralFPGroup(n): RngIntElt -> GrpFP

For $n > 2$, return the dihedral group of order $2n$ as an fp-group. For $n = 0$, return the infinite dihedral group as an fp-group.

ExtraSpecialFPGroup(p, n : parameters): RngIntElt, RngIntElt -> GrpFP

Type: MonStgElt	Default: "+"

Given a small prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} in the category GrpFP. The isomorphism type of G can be selected using the parameter Type.

Possible values for this parameter are "+" (default) and "−".

If Type is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If Type is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

SymmetricFPGroup(n): RngIntElt -> GrpFP

Sym(GrpFP, n): Cat, RngIntElt -> GrpFP

Construct the symmetric group of degree n as an fp-group, where the generators correspond to the permutations $(1, 2, \ldots, n)$ and $(1, 2)$.

Example: Standard Groups as FP-Groups (ex-936a87)

We create the symmetric group Sym(8) as an fp-group:

```
> S8 := SymmetricFPGroup(8);
> S8;
Finitely presented group S8 on 2 generators
Relations
      S8.1^8 = Id(S8)
      S8.2^2 = Id(S8)
      (S8.1 * S8.2)^7 = Id(S8)
      (S8.1^-1 * S8.2 * S8.1 * S8.2)^3 = Id(S8)
      (S8.2 * S8.1^-2 * S8.2 * S8.1^2)^2 = Id(S8)
      (S8.2 * S8.1^-3 * S8.2 * S8.1^3)^2 = Id(S8)
      (S8.2 * S8.1^-4 * S8.2 * S8.1^4)^2 = Id(S8)
```

We create the Coxeter group of Cartan type $F4$ as an fp-group:

```
> F := CoxeterFPGroup("F4");
> F;
Finitely presented group F on 4 generators
Relations
    (F.2 * F.3)^2 = (F.3 * F.2)^2
    F.1^2 = Id(F)
    F.1 * F.3 = F.3 * F.1
    F.2 * F.4 = F.4 * F.2
    F.1 * F.2 * F.1 = F.2 * F.1 * F.2
    F.2^2 = Id(F)
    F.3^2 = Id(F)
    F.3 * F.4 * F.3 = F.4 * F.3 * F.4
    F.4^2 = Id(F)
    F.1 * F.4 = F.4 * F.1
```

80.4.2 Construction of Extensions

`Darstellungsgruppe(G): GrpFP -> GrpFP`

Given an fp-group G, construct a maximal central extension $\tilde{G}$ of G. The group $\tilde{G}$ is created as an fp-group.

`DirectProduct(G, H): GrpFP, GrpFP -> GrpFP`

Given two fp-groups G and H, construct the direct product of G and H.

`DirectProduct(Q): [ GrpFP ] -> GrpFP`

Given a sequence Q of r fp-groups, construct the direct product $Q[1] \times \cdots \times Q[r]$.

`FreeProduct(G, H): GrpFP, GrpFP -> GrpFP`

Given two fp-groups G and H, construct the free product of G and H.

`FreeProduct(Q): [ GrpFP ] -> GrpFP`

Given a sequence Q of r fp-groups, construct the free product of the groups $Q[1], \ldots, Q[r]$.

Example: Maximal Central Extension of a Group `(ex-dc16bb)`

We construct a maximal central extension of the following group of order 36.

```
> G<x1, x2> := FPGroup<x1, x2 | x1^4,(x1*x2^-1)^2,x2^4,(x1*x2)^3>;
> G;

Finitely presented group G on 2 generators
Relations
       x1^4 = Id(G)
       (x1 * x2^-1)^2 = Id(G)
```

(continues on next page)

(continued from previous page)

```
       x2^4 = Id(G)
       (x1 * x2)^3 = Id(G)

> D := Darstellungsgruppe(G);
> D;

Finitely presented group D on 4 generators
Relations
       D.1^4 * D.3^-1 * D.4^2 = Id(D)
       D.1 * D.2^-1 * D.1 * D.2^-1 * D.4 = Id(D)
       D.2^4 = Id(D)
       D.1 * D.2 * D.1 * D.2 * D.1 * D.2 * D.4 = Id(D)
       (D.1, D.3) = Id(D)
       (D.2, D.3) = Id(D)
       (D.1, D.4) = Id(D)
       (D.2, D.4) = Id(D)
       (D.3, D.4) = Id(D)

> Index(D, sub< D | >);
108
```

Thus, a maximal central extension of G has order 108

Example: Direct Product of FP-Groups (ex-bdaee4)

We create the direct product of the alternating group of degree 5 and the cyclic group of order 2.

```
> A5 := FPGroup<a, b | a^2, b^3, (a*b)^5 >;
> Z2 := quo< FreeGroup(1) | $.1^2 >;
> G := DirectProduct(A5, Z2);
> G;

Finitely presented group G on 3 generators
Relations
      G.1^2 = Id(G)
      G.2^3 = Id(G)
      (G.1 * G.2)^5 = Id(G)
      G.3^2 = Id(G)
      G.1 * G.3 = G.3 * G.1
      G.2 * G.3 = G.3 * G.2
```

80.4.3 Conversion to FP-Groups

In this section we describe intrinsics which given a group in some category other than that of fp-group, return an isomorphic fp-group.

`FPGroup(G): GrpPerm -> GrpFP, Hom(Grp)`

`FPGroup(G): GrpMat -> GrpFP, Hom(Grp)`

Given a finite group G in category `GrpPerm` or `GrpMat`, this function returns a finitely presented group F, isomorphic to G, together with the isomorphism $\phi : F \to G$. The generators of F correspond to the generators of G, so this function can be used to obtain a set of defining relations for the given generating set of G.

It should be noted that this function is only practical for groups of order at most a few million. In the case of much larger permutation groups, an isomorphic fp-group can be constructed using the function *FPGroupStrong*.

Example: FP-Group from a Permutation Group (ex-185d54)

We define the alternating group $G \simeq A_{10}$ as a permutation group.

```
> G := Alt(10);
> G;
Permutation group G acting on a set of cardinality 5
Order = 1814400 = 2^7 * 3^4 * 5^2 * 7
    (1, 2)(3, 4, 5, 6, 7, 8, 9, 10)
    (1, 2, 3)
```

Now we create an fp-group F isomorphic to G, using the function *FPGroup*. The presentation is constructed by computing a set of defining relations for the generators of G, i.e. the generators of the returned fp-group correspond to the generators of G. This defines a homomorphism from F to G, which the function *FPGroup* returns as second return value.

```
> F<x,y>, f := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
    x^8 = Id(F)
    y^3 = Id(F)
    (x^-1, y^-1)^2 = Id(F)
    (x * y^-1 * x^-2 * y^-1 * x)^2 = Id(F)
    (x * y * x^-3 * y^-1 * x^2)^2 = Id(F)
    x^-1 * y^-1 * x^-4 * y^-1 * x^4 * y^-1 * x^4 * y^-1 * x^-3 = Id(F)
    (y^-1 * x^-1)^9 = Id(F)
> f;
Mapping from: GrpFP: F to GrpPerm: G
> f(x);
(1, 2)(3, 4, 5, 6, 7, 8, 9, 10)
> f(y);
(1, 2, 3)
```

Constructing a nice presentation for a large finite permutation group or matrix group is a difficult problem. The next two intrinsics obtain a presentation as a side effect of the Todd-Coxeter-Schreier-Sims algorithm.

FPGroupStrong(G): GrpPerm -> GrpFP, Hom(Grp)

FPGroupStrong(G): GrpMat -> GrpFP, Hom(Grp)

Given a finite group G in category GrpPerm or GrpMat, this function returns a finitely presented group F, isomorphic to G, together with the isomorphism $\phi : F \to G$. The generators of F correspond to a set of strong generators of G. If no strong generating set is known for G, one will be constructed.

For a detailed description of this function, in particular for a list of available parameters, we refer to Chapter Permutation Groups and Chapter Matrix Groups over General Rings, respectively.

FPGroupStrong(G, N): GrpPerm, GrpPerm -> GrpFP, Hom(Grp)

Given a permutation group G and a normal subgroup N of G, this function returns a finitely presented group F, isomorphic to G/N, together with a homomorphism $\phi : G \to F$.

For a detailed description of this function, we refer to Chapter Permutation Groups.

Example: FP-Group on Strong Generators (ex-f1a62b)

We take the alternating group $G \simeq A_5$ as a permutation group.

```
> G := Alt(5);
> G;
Permutation group G acting on a set of cardinality 5
Order = 60 = 2^2 * 3 * 5
    (1, 2)(3, 4, 5, 6, 7, 8, 9, 10)
    (1, 2, 3)
```

Using the function *FPGroupStrong*, we now create an fp-group Fs, isomorphic to G, whose generators correspond to a set of strong generators of G.

```
> Fs<[z]>, fs := FPGroupStrong(G);
> Fs;
Finitely presented group Fs on 3 generators
Relations
    z[1]^-3 = Id(Fs)
    (z[1]^-1 * z[3]^-1)^2 = Id(Fs)
    z[3]^-3 = Id(Fs)
    z[1]^-1 * z[2]^-1 * z[1] * z[3]^-1 * z[2] = Id(Fs)
    z[2]^-3 = Id(Fs)
    (z[3]^-1 * z[2]^-1)^2 = Id(Fs)
> fs;
Mapping from: GrpFP: Fs to GrpPerm: G
```

Applying the isomorphism fs, we have a look at the strong generating set constructed for G.

```
> [ fs(z[i]) : i in [1..#z] ];
[
    (3, 4, 5),
    (1, 2, 3),
    (2, 3, 4)
]
```

It should be noted that `FPGroupStrong` constructs a presentation on what is often a much larger generating set and consequently the presentation can be much larger. However, this intrinsic will produce presentations for very large groups as compared to the intrinsic `FPGroup`. It is recommended that the user apply the simplification algorithm (intrinsic `Simplify`) to the presentation returned by `FPGroupStrong`).

Groups that satisfy certain properties, such as being abelian or polycyclic, are known to possess presentations with respect to which the word problem is soluble. Specialised categories have been constructed in Magma for several of these, e.g. the categories *GrpGPC*, GrpPC and *GrpAb*. The functions described in this section allow a group created in one of the special presentation categories to be recast as an fp-group.

`FPGroup(G): GrpPC -> GrpFP, Hom(Grp)`

`FPGroup(G): GrpGPC -> GrpFP, Hom(Grp)`

`FPGroup(G): GrpAb -> GrpFP, Hom(Grp)`

Given a group G, defined either by a polycyclic group presentation (types `GrpPC` and `GrpGPC`) or an abelian group presentation (type `GrpAb`), return a group H isomorphic to G, together with the isomorphism $\phi : G \to H$. The generators for H will correspond to the generators of G. The effect of this function is to convert a presentation in a special form into a general fp-group presentation.

Example: Converting a Polycyclic Group to an FP-Group (ex-a91a69)

We illustrate the cast from special forms of fp-groups to the category `GrpFP` by converting a polycyclic group.

```
> G := DihedralGroup(GrpGPC, 0);
> G;
GrpGPC : G of infinite order on 2 PC-generators
PC-Relations:
    G.1^2 = Id(G),
    G.2^G.1 = G.-2
> F := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
    F.1^2 = Id(F)
```

(continues on next page)

(continued from previous page)

```
    F.2^F.1 = F.2^-1
    F.1^-1 * F.2^-1 * F.1 = F.2
```

There is a special Magma category GrpPermCox, a subcategory of GrpPerm, for finite Coxeter groups. Here, we describe a function to create from a Coxeter group W a finitely presented group F, isomorphic to W, which is given by the standard Coxeter group presentation. We refer to Chapter Coxeter Groups for the details.

CoxeterFPGroup(W): GrpFPCox -> GrpFP, Map

CoxeterFPGroup(W): GrpPermCox -> GrpFP, Map

```
Local: BoolElt                    Default: false
```

Given a finite Coxeter group W in the category GrpFPCox or GrpPermCox, construct a finitely presented group F isomorphic to W, given by a standard Coxeter presentation. The isomorphism from W to F is returned as second return value. The first argument to this function must be the category GrpFP.

If the parameter Local is set to true, F is the appropriate subgroup of the FP version of the overgroup of W.

Example: Coxeter Group as an FP-Group (ex-1b08d9)

We construct a Coxeter group W of Cartan type $C5$ and create an isomorphic fp-group F. We can use the isomorphism from W to F to map words in the generators of F to permutation group elements and vice versa.

```
> W := CoxeterGroup("C5");
> F<[s]>, h := CoxeterFPGroup(W);
> F;
Finitely presented group F on 5 generators
Relations
    s[3]^2 = Id(F)
    s[2] * s[4] = s[4] * s[2]
    s[1]^2 = Id(F)
    s[1] * s[2] * s[1] = s[2] * s[1] * s[2]
    s[2] * s[5] = s[5] * s[2]
    s[4]^2 = Id(F)
    s[1] * s[3] = s[3] * s[1]
    s[3] * s[4] * s[3] = s[4] * s[3] * s[4]
    s[2]^2 = Id(F)
    s[1] * s[4] = s[4] * s[1]
    s[5]^2 = Id(F)
    (s[4] * s[5])^2 = (s[5] * s[4])^2
    s[3] * s[5] = s[5] * s[3]
    s[1] * s[5] = s[5] * s[1]
    s[2] * s[3] * s[2] = s[3] * s[2] * s[3]
> h;
Mapping from: GrpCox: W to GrpFP: F given by a rule
> h(W.1*W.2);
s[1] * s[2]
> (s[1]*s[2]*s[3]*s[4]) @@ h;
(1, 39, 4, 3, 2)(5, 13, 19, 23, 25)(6, 36, 35, 8, 7)(9, 16, 21,
    24, 17)(10, 33, 32, 31, 11)(12, 18, 22, 15, 20)(14, 29, 28,
    27, 26)(30, 38, 44, 48, 50)(34, 41, 46, 49, 42)(37, 43, 47,
    40, 45)
```

80.5 Properties of an FP-Group

In this section intrinsics for providing information about the global properties of an fp-group are described.

80.5.1 Cardinality

A very basic requirement is to determine whether a given fp-group G is finite or infinite. Unfortunately, this is a problem for which there is no general solution. Instead, an intrinsic `Order` is provided which attempts to prove that the group is finite and a second intrinsic `IsInfiniteFPGroup` is provided which attempts to prove that the group is infinite. If both fail to produce a result then no conclusion can be drawn.

We assume that the group is finite and try to prove that this assumption is true using coset enumeration and other techniques.

`Order(G: parameters): GrpFP -> RngIntElt`

`FactoredOrder(G: parameters): GrpFP -> [ <RngIntElt, RngIntElt> ]`

Given an fp-group G, this function attempts to determine the order of G or to prove that G is infinite. If a finite order can be computed, the function `Order` returns the order as a positive integer, whereas the function `FactoredOrder` returns a sequence of prime power factors. The function `FactoredOrder` reports an error in all other cases, whereas the function `Order` returns the object `Infinity`, if G can be shown to be infinite and returns a value of 0 if neither a finite value for the group order nor a proof for the infinity of G can be obtained. No conclusions can be drawn from a return value 0 of `Order`.

In addition to the parameters controlling possibly invoked coset enumerations, there exist some other parameters controlling the strategy used by the functions `Order` and `FactoredOrder`. These parameters are described below.

Example: Order of a Finitely Presented Group (ex-134948)

We use the function *Order* without any parameters to compute the order of the group

$$G = < a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 > .$$

```
> G<x, y> := FPGroup<x,y | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> G;
Finitely presented group G on 2 generators
Relations
    x^8 = Id(G)
    y^7 = Id(G)
    (x * y)^2 = Id(G)
    (x^-1 * y)^3 = Id(G)
> Order(G);
10752
```

The strategy employed by the functions `Order` and `FactoredOrder` may involve trying to obtain information on certain subgroups of G. Whether or not an attempt is made to construct a presentation for a subgroup arising in the course of the computation by means of Reidemeister-Schreier rewriting, is controlled by three parameters:

```
UseRewrite: BoolElt                    Default: true
MinIndex  : RngIntElt                  Default: 10
MaxIndex  : RngIntElt                  Default: 1000
```

If `UseRewrite` is set to `false`, attempts to construct presentations for subgroups are not made. Otherwise, `MinIndex` and `MaxIndex` specify for subgroups of which index range Reidemeister-Schreier rewriting is done.

The following strategy is used for trying to determine the order of G.

(1) Check whether G is free. If so, G is either trivial or infinite.

(2) Check whether the presentation for G is deficient (i.e. whether the number of relations is smaller than the number of generators). If it is, G is infinite.

(3) Check the subgroups of G with known order. If such a subgroup is known to be infinite or if we can compute its index in G, we're done.

(4) Try to compute the index of G in a supergroup of known order. (An infinite supergroup in which G has finite index proves G to be infinite.)

(5) Try to enumerate the cosets of the trivial subgroup in G.

(6) Check the subgroups of known or easily computable index in G. If we can compute the order of such a subgroup or prove that it is infinite, we're done.

(7) Try to enumerate the cosets of some subgroups occurring "naturally" in the presentation of G.

(8) Check the supergroups in which G has known or easily computable index. If we can compute the order of a supergroup or prove that it is infinite, we're done.

(9) Try to rewrite G w.r.t. some supergroup and to enumerate the cosets of the trivial subgroup using the resulting presentation.

Steps requiring coset enumeration in G or a supergroup of G are skipped, if no relations are known for this group. Steps involving Reidemeister-Schreier rewriting may be skipped according to the values of the parameters mentioned above.

Experienced users can control the behaviour of coset enumerations which may be invoked by the functions `Order` and `FactoredOrder` with a wide range of parameters. Both functions – in addition to the parameters mentioned above – accept the same parameters as the function *`CosetEnumerationProcess`* in Subsec. *Interactive Coset Enumeration*.

The intrinsic `IsInfiniteFPGroup` attempts to prove that an fp-group is infinite by finding a subgroup that has an infinite abelian section.

`IsInfiniteFPGroup(G : parameters): GrpFP -> BoolElt`

Given a finitely-presented group G, the function attempts to establish that G is infinite. If it succeeds it returns `true`. Otherwise it returns `false` meaning that the tests it has made don't prove the group to be infinite. This does not imply that the group is likely to be finite. The code searches for epimorphisms ϕ of G and uses the Holt-Plesken cohomological criterion, as implemented in the intrinsic `HasPositiveH1Dimension`, to test for non-finiteness of the kernel of ϕ. The version in this release works best on groups G which are perfect or which have small abelian quotients. It will be extended to handle groups which have larger soluble quotients in a subsequent release.

At present epimorphisms are sought using four techniques that are described elsewhere in this chapter: L2-quotient, Simple Quotient, Low Index Subgroups, and Low Index Normal Subgroups. The intrinsic has corresponding parameters `L2Quot`, `SimQuot`, `LISub`, and `LINSub` that allow the user to control the types of epimorphisms to be used. The default is for all four methods to be employed – however, the user can turn any of them off by setting the appropriate parameter to `false`. A verbose flag `InfGrp` will provide some information about progress.

The application of the Holt-Plesken theorem to an epimorphism ϕ requires a knowledge of the QH modules, where H is the image of ϕ. A database containing such modules for ATLAS groups of degree up to 1000 is available as an optional download. The serious user of `IsInfiniteFPGroup` is strongly advised to have this database available as it can save a great deal of runtime.

Example: Proving a Group Infinite `(ex-fea44f)`

Consider the following one-relator quotient group of the triangle group $\langle 2, 3, 7\rangle$:

$$\langle x, y | x^2, y^3, (xy)^7, (x, y)^{12}\rangle$$

```
> SetVerbose("InfGrp", 2);
> G<x,y> := FPGroup<x,y|x^2, y^3, (x*y)^7, (x,y)^12>;
> inf := IsInfiniteFPGroup(G);
>
 L2Q SEARCH commencing:
   L2Q SEARCH Considering L2Qs with degrees in the interval [1, 1000]
 SIMPLE QUOTIENT Commence searching interval [60, 10000]:
   Found epi onto PSL(2, 7) of order 168 and degree 8: 0.000 secs.
   Found epi onto PSL(2, 13) of order 1092 and degree 14: 0.060 secs.
 SIMPLE QUOTIENT Search of interval [60, 10000] unsuccessful: 2 epis, 0.610
secs.

 NORMAL SUBGROUPS Search of indices [1, 1000] commencing:
 NORMAL SUBGROUPS Search of [1, 1000] unsuccessful: 1 + 0 subgroups, 0.200
secs.

 SUBGRP SEARCH of indices [1, 25] commencing:
 SUBGRP SEARCH of [1, 25] was unsuccessful: 8 + 8 subgroups, 0.990 secs.

 SIMPLE QUOTIENT Commence searching interval [20001, 100000]:
 SIMPLE QUOTIENT Search of interval [20001, 100000] unsuccessful: 0 epis,
0.620 secs.

 NORMAL SUBGROUPS Search of indices [1001, 10000] commencing:
 NORMAL SUBGROUPS Search of [1001, 10000] unsuccessful: 0 + 0 subgroups,
0.960 secs.

 SUBGRP SEARCH of indices [26, 50] commencing:
 SUBGRP SEARCH of [26, 50] was unsuccessful: 8 + 8 subgroups, 1.460 secs.

 SIMPLE QUOTIENT Commence searching interval [100001, 1000000]:
   Found epi onto J_2 of order 604800 and degree 100: 1.300 secs.
     For 4950-dim ex_sq of perm module, dim of H^1/Z is 1: 1.050 secs.
 SIMPLE QUOTIENT Search of interval [100001, 1000000] successful: 1 epis,
2.430 secs.

> inf;
true
```

So the group G has the simple group J_2 as quotient and the rational representation corresponding to the exterior square of J_2's degree 100 permutation module lifted to G has first cohomology group of dimension 1. So G is infinite by an application of the Holt-Plesken theorem.

`HasPositiveH1Dimension(G, phi : parameters): GrpFP, HomGrp -> BoolElt`

Given a finitely-presented group G and an epimorphism ϕ of G onto a permutation group H, this intrinsic returns `true` if some $\mathbb{Q}H$-module, where $\mathbb{Q}$ is the field of the rationals, lifts to a $\mathbb{Q}G$-module whose first cohomology group is non-trivial. By a result of Holt and Plesken this shows that G is infinite. The $\mathbb{Q}H$-modules are obtained in three ways. The first consists of $\mathbb{Q}H$-modules that are easily obtainable from the permutation representation of H. The second method is to systematically construct the $\mathbb{Q}H$-modules of H while the third source is the Magma database of rational representations of small simple groups. Parameters allow the user to specify which of these methods are used (see below).

The parameter `ESLimit` allows the user to place a limit on the dimension of exterior squares for which the Holt-Plesken test will be applied. If the parameter `CmpIrrs` is set to `true` then Magma will construct rational irreducible representations for H if they are not present in the database. If the parameter `DBIrrs` is set to `true` then Magma will use those irreducible modules for H that are available in the Rational Representations Database. Setting both `CmpIrrs` and `DBIrrs` to `false`

specifies that the test should be applied only to the permutation module and its exterior square.

There are two signatures for this intrinsic where the second has a third argument needed by the intrinsic `IsInfiniteFPGroup`. Note that this version is only intended to be called internally. For all other uses the two-argument version should be used.

80.5.2 Small Cancellation Conditions

`SmallCancellationConditions(G): GrpFP -> RngIntElt, RnIntElt, FldRatElt)`

`Print`	: `RngIntElt`	Default: 0
`involgens`	: `BoolElt`	Default: false

This functions calculates the values for which the small cancellation conditions $T(i)$, $C(j)$, and $C'(k)$ are satisfied by the presentation G. It returns i, j, k', where $T(i)$, $C(j)$, and $C'(k)$ are satisfied for all $k > k'$.

If the parameter `involgens` is set `true` then edges labelled by involutory generators x in a van Kampen diagram for G are regarded as undirected, and are not regarded as pieces of the relator x^2.

Example: Small Cancellation Conditions `(ex-2f799a)`

```
> G := FPGroup< x,y | x^6, y^6, (x*y)^5, (x,y)^3 >;
> SmallCancellationConditions(G);
This presentation satisfies T( 3 ).
This presentation satisfies C( 5 ).
This presentation satisfies C'(k) for all k > 1/5 .
3 5 1/5
```

80.5.3 Largeness

`IsLarge(G, L, U:parameters): GrpFP, RngIntElt, RngIntElt -> BoolElt, GrpFP`

`Simplify`	: `BoolElt`	Default: true
`MaxExponent`	: `RngIntElt`	Default: 100000
`TimeLimit`	: `RngIntElt`	Default: 10

Attempt to show that G has a subgroup with homomorphic image the free group of rank 2. If successful return `true`, plus a subgroup witness to this property. If not successful, return `false`. Note that `false` does not imply that G doesn't have the stated property, just that no such subgroup has been found. The witness group returned has a finite index subgroup with the free group of rank 2 as a homomorphic image. The algorithm will consider subgroups of G with index h having $L \leq h \leq U$.

We attempt to decide if G is large by calculating the (multivariable) Alexander polynomial of those subgroups H of G having index h in the given range, and also subgroups of H having index up to $2h$ in G.

Controls simplification of subgroup presentations.

If the abelian torsion of subgroup H is greater than `MaxExponent` we do not rewrite to obtain a presentation of H.

If, in computing low index subgroups of a finite index subgroup H of G, the low-index subgroup calculation takes more than `TimeLimit` in seconds, then abort without trying to find more subgroups of H.

80.6 Subgroups

80.6.1 Specification of a Subgroup

`sub< G | L >: GrpFP, List -> GrpFP`

Construct the subgroup H of the fp-group G generated by the words specified by the terms of the *generator list* $L = L_1, \ldots, L_r$.

A term L_i of the generator list may consist of any of the following objects:

(a) A word;

(b) A set or sequence of words;

(c) A sequence of integers representing a word;

(d) A set or sequence of sequences of integers representing words;

(e) A subgroup of an fp-group;

(f) A set or sequence of subgroups.

The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

The generators of H consist of the words specified directly by terms L_i together with the stored generating words for any groups specified by terms of L_i. Repetitions of an element and occurrences of the identity element are removed (unless H is trivial).

If the `sub`-constructor is invoked with an empty list L, the trivial subgroup will be constructed.

`sub< G | f >: GrpFP, Hom(Grp) -> GrpFP`

Given a homomorphism f from G onto a transitive subgroup of $\mathrm{Sym}(n)$, construct the subgroup of G which affords this permutation representation.

`ncl< G | L >: GrpFP, List -> GrpFP`

Construct the subgroup N of the fp-group G as the normal closure of the subgroup H generated by the words specified by the terms of the *generator list* L.

The possible forms of a term of the generator list are the same as for the *`sub< G | L >`*-constructor.

This constructor may be applied even when H has infinite index in G, provided that its normal closure N has finite index. The subgroup N is obtained by computing the coset table of the trivial subgroup in the group defined by the relations of G together with relators corresponding to the words generating H. For a sample application of this function, see Example *Braid Group onto Sym(5)*.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *`SetGlobalTCParameters`*. For a detailed description of the available parameters and their meanings, see Subsec. *Interactive Coset Enumeration*.

`ncl<G | f>: GrpFP, Hom(Grp) -> GrpFP`

Given a homomorphism f from G onto a transitive subgroup of $\mathrm{Sym}(n)$, construct the subgroup of G that is the normal closure of the subgroup K of G which affords this permutation representation.

`CommutatorSubgroup(G): GrpFP -> GrpFP`

`DerivedSubgroup(G): GrpFP -> GrpFP`

`DerivedGroup(G): GrpFP -> GrpFP`

Given an fp-group G, try to construct the derived subgroup G' of G as finite index subgroup of G. The construction fails if no presentation for G is known or can be constructed, or if the index of G' in G is too large or infinite.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global

parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, see Subsec. *Interactive Coset Enumeration*.

Example: Subgroup of Index 448 (ex-6af151)

The group $(8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 >,$$

and has a subgroup of index 448 generated by the words a^2 and $a^{-1}b$:

```
> G<a, b> := FPGroup<a, b| a^8, b^7, (a * b)^2, (a^-1 * b)^3>;
> G;
Finitely presented group G on 2 generators
Relations
     a^8 = Id(G)
       b^7 = Id(G)
       (a * b)^2 = Id(G)
> H<x, y> := sub< G | a^2, a^-1 * b >;
> H;
Finitely presented group H on 2 generators
Generators as words in group G
      x = a^2
      y = a^-1 * b
```

Example: Subgroup Defined by a Homomorphism (ex-4fd961)

Given the group G defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (a, b)^9 >,$$

there is a homomorphism into Sym(9) defined by

$$a \to (2, 4)(3, 5)(6, 7)(8, 9)$$
$$b \to (1, 2, 3)(4, 6, 7)(5, 8, 9)$$

We construct the subgroup H of G that is the preimage of the stabiliser of the point 1 in G.

```
> G<a, b> := FPGroup< a, b | a^2, b^3, (a*b)^7, (a, b)^9>;
> T := PermutationGroup< 9 | (2, 4)(3, 5)(6, 7)(8, 9),
>    (1, 2, 3)(4, 6, 7)(5, 8, 9) >;
> f := hom< G -> T | a -> T.1, b ->T.2 >;
> H := sub< G | f >;
> H;
Finitely presented group H
Subgroup of group G defined by coset table
> Index(G, H);
9
```

Using the function *GeneratingWords*, we obtain a set of generators for H.

```
> print GeneratingWords(G, H);
     { a, b^-1 * a * b^3 * a * b, b * a * b * a * b * a * b^-1,
       b^3, b^-1 * a * b * a * b * a * b, b * a * b^3 * a * b^-1 }
```

80.6.2 The Todd-Coxeter Algorithm

This section describes the simplest use of coset enumeration techniques in Magma. Magma also provides interactive facilities for coset enumeration. For information on these more advanced uses, the user is referred to Subsec. *Interactive Coset Enumeration*.

The Todd-Coxeter implementation installed in Magma is based on the stand-alone coset enumeration programme ACE3 developed by George Havas and Colin Ramsay at the University of Queensland. The reader should consult [CDHW73] and [Hav91] for an explanation of the terminology and a general description of the algorithm. A manual for ACE3 as well as the sources of ACE3 can be found online [Ram].

Experienced users can control the Todd-Coxeter procedures invoked by the functions described in this section with a wide range of parameters. For a complete description of these parameters and their meanings we refer to the manual entry for the function *CosetEnumerationProcess*. We just mention briefly the most important ones:

```
CosetLimit: RngIntElt                    Default: 0
```

If CosetLimit is set to n, where n is a positive integer, then the coset table may have at most n rows. In other words, a maximum of n cosets can be defined at any instant during the enumeration. It is ensured in this case, that enough memory is allocated to store the requested number of cosets, regardless of the value of the parameter *Workspace*. If CosetLimit is set to 0 (default), the maximal number of active cosets is determined by the size of the coset table (cf. parameter *Workspace*) and the number of columns of the coset table (i.e. the number of group generators).

```
Workspace: RngIntElt                     Default: 4000000
```

The number of words allocated for the coset table. Note that if *CosetLimit* is set, at least as much memory is allocated as is necessary to store the requested number of cosets.

```
Strategy: MonStgElt                      Default:
```

Using this parameter one of several predefined strategies can be selected. (See the manual entry for the function *CosetEnumerationProcess* in Subsec. *Interactive Coset Enumeration* for a complete list.) The most important values of this parameter are:

- "Easy": This selects a combination of parameters, which in situations where an overflow is not expected is likely to produce a result more quickly and using less memory than the default strategy. This strategy will fail for more complicated enumerations.
- "Hard": This selects a combination of parameters, which is more likely to produce a finite result for difficult coset enumerations than the default strategy. This strategy will use more memory and enumerations usually will take longer.

ToddCoxeter(G, H: parameters): GrpFP, GrpFP -> RngIntElt, Map, RngIntElt, RngIntElt

Given a subgroup H of the fp-group G, this function attempts to build up a coset table of H in G using the Todd-Coxeter procedure.

The first return value is the index of H in G (or 0, if the enumeration fails to complete). The other values returned are the coset table map, the maximum number of simultaneously active cosets, and the total number of cosets defined during the enumeration.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function *CosetEnumerationProcess* in Subsec. *Interactive Coset Enumeration*.

Index(G, H: parameters): GrpFP, GrpFP -> RngIntElt

FactoredIndex(G, H: parameters): GrpFP, GrpFP -> [<RngIntElt, RngIntElt>]

Given a subgroup H of the fp-group G, these functions attempt to determine the index of H in G by enumerating the cosets of H using the Todd-Coxeter procedure. The index is returned as a positive integer (Index) or as a sequence of factors (FactoredIndex), respectively. If the coset enumeration fails to complete with a closed coset table, Index returns a value of 0, whereas FactoredIndex reports an error. No conclusion can be drawn in this case.

Experienced users can control the Todd-Coxeter procedure invoked by these functions with a wide range of parameters. Both functions accept the same parameters as the function *CosetEnumerationProcess* in Subsec. *Interactive Coset Enumeration*.

Example: Todd-Coxeter Enumeration of 448 Cosets (ex-0e7ed5)

The classical test example for Todd-Coxeter programmes is the enumeration of the 448 cosets of the subgroup $H = < a^2, a^{-1}b >$ in the group $G = < a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 >$.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a^2,a^-1*b>;
> Index(G, H);
448
```

Example: Todd-Coxeter for the Harada-Norton Group (ex-c75263)

The Harada-Norton simple group has the presentation

$$\begin{aligned} < x, a, b, c, d, e, f, g \mid\ & x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, (x,a), (x,g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, (ef)^3, (eg)^2, (fg)^3, (b, xbx), (a, edcb), \\ & (a,f)dcbdcd, (ag)^5, (cdef, xbx), (b, xcdefx), (cdef, xcdefx) > \end{aligned}$$

The subgroup generated by x, b, c, d, e, f, g has index 1,140,000. We use the parameter *CosetLimit* to request a sufficiently large coset table. For the enumeration we choose the predefined strategy *Hard* with the modification of a complete C-style lookahead (*Lookahead* := 2).

```
> HN<x, a, b, c, d, e, f, g> :=
>     FPGroup< x, a, b, c, d, e, f, g |
>               x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>               (x, a), (x, g),
>               (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>               (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>               (d*e)^3, (d*f)^2, (d*g)^2,
>               (e*f)^3, (e*g)^2,
>               (f*g)^3,
>               (b, x*b*x),
>               (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>               (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>               (c*d*e*f, x*c*d*e*f*x)
>            >;
> H := sub<HN | x,b,c,d,e,f,g >;
> idx := Index(HN, H: Print := true, CosetLimit := 1200000,
>                     Strategy := "Hard", Lookahead := 2);
INDEX = 1140000
 (a=1140000 r=1471 h=1168483 n=1168483;
  l=2945 c=201.17;
  m=1142416 t=1470356)
> idx;
1140000
```

Example: Orders of a Parametrised Family of Groups (ex-4e7bab)

We use a function representing a parametrised presentation to determine the order of a collection of groups obtained by systematically varying one relation. We select the predefined coset enumeration strategy *Easy* for the order computations.

```
> Grp := func< p, q, r, s |
>
>   FPGroup<
>     x, y, z, h, k, a |
>     x^2, y^2, z^2, (x,y), (y,z), (x,z), h^3, k^3, (h,k),
>     (x,k), (y,k), (z,k), x^h*y, y^h*z, z^h*x, a^2, a*x*a*y,
>     a*y*a*x, (a,z), (a,k), x^p*y^q*z^r*k^s*(a*h)^2 >
>        >;
> [ < <i,j,k,l>, Order(Grp(i,j,k,l) : Strategy := "Easy") >
>   : i, j, k in [0..1], l in [0..2] ];
[ <<0, 0, 0, 0>, 144>, <<0, 0, 1, 0>, 18>, <<0, 1, 0, 0>, 72>,
  <<0, 1, 1, 0>, 36>, <<1, 0, 0, 0>, 18>, <<1, 0, 1, 0>, 144>,
  <<1, 1, 0, 0>, 36>, <<1, 1, 1, 0>, 72>, <<0, 0, 0, 1>, 144>,
  <<0, 0, 1, 1>, 18>, <<0, 1, 0, 1>, 72>, <<0, 1, 1, 1>, 36>,
  <<1, 0, 0, 1>, 18>, <<1, 0, 1, 1>, 144>, <<1, 1, 0, 1>, 36>,
  <<1, 1, 1, 1>, 72>, <<0, 0, 0, 2>, 144>, <<0, 0, 1, 2>, 18>,
  <<0, 1, 0, 2>, 72>, <<0, 1, 1, 2>, 36>, <<1, 0, 0, 2>, 18>,
  <<1, 0, 1, 2>, 144>, <<1,1, 0, 2>, 36>, <<1, 1, 1, 2>, 72> ]
```

80.6.3 Interactive Coset Enumeration

Introduction

This section presents the interactive coset enumeration facility of Magma. This concept makes it possible to restart an enumeration after changing enumeration parameters or adding relators or subgroup generators, while making use of information obtained up to that point as much as possible. It is thus particularly suitable for very hard enumerations, requiring a careful and interactive choice of enumeration parameters or for series of similar enumerations.

The Todd-Coxeter implementation installed in Magma is based on the stand alone coset enumeration programme ACE3 developed by George Havas and Colin Ramsay at the University of Queensland. The reader should consult [CDHW73] and [Hav91] for an explanation of the terminology and a general description of the algorithm. A manual for ACE3 as well as the sources of ACE3 can be found online [Ram].

In Magma an interactive coset enumeration is realised as an object of the category GrpFPCosetEnumProc which can be created and modified, allows starting and restarting of coset enumerations and provides access to internal data like the coset table and various status information.

Constructing and Modifying a Coset Enumeration Process

CosetEnumerationProcess(G, H: parameters): GrpFP, GrpFP -> GrpFPCosetEnumProc

```
Compact             : RngIntElt                 Default: 10
CosetLimit          : RngIntElt                 Default: 0
Workspace           : RngIntElt                 Default: 4000000
FillFactor          : RngIntElt                 Default: 0
CTFactor            : RngIntElt                 Default: 1000
RTFactor            : RngIntElt                 Default: 2000/1
Style               : MonStgElt                 Default: "R_CR"
Lookahead           : RngIntElt                 Default: 0
Mendelsohn          : BoolElt                   Default: false
RelationsInSubgroup: RngIntElt                  Default: -1
RowFilling          : BoolElt                   Default: true
PrefDefMode         : RngIntElt                 Default: 3
PrefDefSize         : RngIntElt                 Default: 8
DeductionMode       : RngIntElt                 Default: 4
DeductionSize       : RngIntElt                 Default: 1000
```

(continues on next page)

(continued from previous page)

```
PathCompression    : BoolElt                    Default: false
TimeLimit          : RngIntElt                  Default: -1
LoopLimit          : RngIntElt                  Default: 0
LowerBound         : RngIntElt                  Default: 1
Print              : BoolElt                    Default: false
Messages           : RngIntElt                  Default: 0
Strategy           : MonStgElt                  Default:
```

This function creates a coset enumeration process for enumerating the cosets of the subgroup H of the finitely presented group G. Note that no actual coset enumeration is started for the created coset enumeration process. This can be done with the function *StartEnumeration*.

The user can control in detail the way a subsequent enumeration will be performed with the help of the following parameters. For a more thorough explanation of the parameters and of how they affect the coset enumeration, we refer to the ACE3 manual.

This parameter controls the compaction of the coset table during an enumeration. A compaction will be done if a new coset definition is required, there is no space for a new coset available in the coset table and the percentage of dead cosets exceeds the value of the parameter Compact.

If CosetLimit is set to n, where n is a positive integer, then the coset table may have at most n rows. In other words, a maximum of n cosets can be defined at any instant during the enumeration. It is ensured in this case, that enough memory is allocated to store the requested number of cosets, regardless of the value of the parameter *Workspace*. If CosetLimit is set to 0 (default), the maximal number of active cosets is determined by the size of the coset table (cf. parameter *Workspace*) and the number of columns of the coset table (i.e. the number of group generators).

The number of words allocated for the coset table. Note that if *CosetLimit* is set, at least as much memory is allocated as is necessary to store the requested number of cosets.

In certain situations it is necessary to ensure that a certain proportion, the *fill fraction* (see Havas (1991)), of the coset table is always kept filled, even if *preferred definitions* are made. The parameter FillFactor allows the user to specify the fill fraction which is the reciprocal of FillFactor. The default value of 0 selects a fill factor of $\lfloor (5 * (c + 2))/4 \rfloor$, where c is the number of columns in the coset table.

These parameters control the enumeration style and the balance between coset table style definitions (C-style, Felsch style) and relator table style definitions (R-style, HLT style).

The parameters CTFactor and RTFactor set the number of C-style coset definitions and R-style coset applications, respectively, which are performed in every step of the appropriate type. The default value for RTFactor is $2000/l$, where l is the total length of the relators.

The possible arguments to the parameter Style are "R", "C", "R_CR", "Rc", "Cr", "Rt", and "CR".

In style R all definitions are made via relator scans, i.e. this is HLT mode.

In style C all definitions are made by filling the next empty position in the coset table and testing the new coset in all essentially different positions in the relators, i.e. this is Felsch mode.

In style R_CR we run in style R until an overflow occurs, perform a lookahead on the entire table and then switch to style CR (see below).

The styles Rc and Cr are like the styles R and C, except that a single style C or style R pass is done after the initial R or style C pass, respectively.

In style Rt definitions are made as in style R, but the new cosets are tested in all essentially different positions in the relators as in style C.

In style CR alternate passes of style C and style R are performed, with all definitions tested in all essentially different positions in the relators as in style C.

In HLT style enumerations, possible implications of new definitions or deductions made while tracing relators may not be detected until much later. One possible solution to this problem

is to perform lookaheads occasionally, i.e. to process the coset table, looking for deductions or coincidences. A lookahead can be done using the entire table (*complete*) or only that part of the table above the current coset (*partial*) and it can be done in R-style (scanning cosets from the beginning of relators) or in C-style (testing all definitions in all essentially different relator positions). The following lookahead modes are supported.

For `Lookahead := 0` (default) no lookahead is done.

For `Lookahead := 1` a partial R-style lookahead is done.

For `Lookahead := 2` a complete C-style lookahead is done.

For `Lookahead := 3` a complete R-style lookahead is done.

For `Lookahead := 4` a partial C-style lookahead is done.

If `Mendelsohn` is set, coset applications are done at all cyclic permutations of the relators. This is usually very expensive but may nevertheless be helpful in certain cases.

This parameter controls, whether relators are treated as additional subgroup generators, i.e. whether they are applied to coset 1 at the start of an enumeration. An argument of -1 (default) includes all relators, an argument of 0 turns this feature off and a positive argument includes the appropriate number of relators, in order.

In R-style it is usual to scan each row of the table after its coset has been applied to all relators, and to make definitions to fill any holes encountered. Failure to do so can cause even simple enumerations to overflow. To switch row filling off, set the parameter `RowFilling` to false.

If the argument is 0, then Felsch style definitions are made using the next empty position in the coset table. Otherwise, gaps of length one found during relator scans are preferentially filled. If the argument is 1, they are filled immediately, and if it is 2, the consequent deduction is also made immediately. If the argument is 3, then the gaps are noted in the preferred definition queue and the next coset definition will be made to fill the oldest gap of length one.

This parameter controls the size of the preferred definition queue, which is implemented as a ring buffer, dropping earliest entries. Setting `PrefDefSize` to n allocates a buffer of size 2^n.

A completed table is only valid if every table entry has been tested in all essentially different relator positions. Untested deductions are stored on a stack. This parameter allows the user to specify how deductions should be handled. The possible actions are:

For `DeductionMode := 0` : discard deductions if there is no stack space left.

For `DeductionMode := 1` : as 0, but redundant cosets are purged off the top of the stack whenever a coincidence is found.

For `DeductionMode := 2` : as 0, but all redundant cosets are purged from the stack whenever a coincidence is found.

For `DeductionMode := 3` : discard the entire stack if it overflows.

For `DeductionMode := 4` : if the stack overflows, then double the stack size and purge all redundant cosets from the stack.

If deductions are discarded for any reason during an enumeration, then a final relator application pass will be done at the end of the enumeration automatically to check the result.

Sets the (initial) size of the deduction stack in words, with one deduction taking two words. A value of 0 selects the default size of 1000 words.

Switching this option on reduces the amount of data movement during coincidence processing at the expense of tracing and compressing coincidence paths, which involves many coset table accesses. The value of this parameter has no effect on the result but may influence the running time.

This parameter puts a time limit in seconds on the length of an enumeration. A value of -1 (default) means no limit. A value of 0 performs exactly one pass through the main enumeration loop.

The core enumerator is organised as a state machine, with each step performing an action (e.g. lookahead, compaction) or a block of actions (e.g. *CTFactor* coset definitions or *RTFactor* coset applications). Using this parameter, a limit on the total number of steps can be imposed. A value of 0 (default) turns this feature off.

This may be set to any known lower bound for the index. Should it happen that the coset table has no holes, and the number of active cosets is equal to the given bound, the enumeration will terminate. When the given bound is equal to the index, this will save tracing relators at many cosets when there is no possibility of finding coincidences.

If this parameter is set to true, the enumerator prints a single message at the end of an enumeration and possibly some progress messages during an enumeration (cf. parameter *Messages*).

If the argument n of this parameter is non-zero (and *Print* is set to true), then a progress message is printed after every $|n|$ "actions" (i.e. coset definitions, deductions, coincidences, etc.) A negative value of n turns hole monitoring on.

Using this parameter one of several predefined strategies can be selected. The effect is equivalent to selecting an appropriate combination of arguments for some of the parameters explained above. Any predefined strategy can be modified by explicitly overriding parameter values.

The following predefined strategies exist: "Default", "Easy", "Hard", "Felsch", "HLT", "CT", "RT", "Sims1", "Sims3", "Sims5", "Sims7", "Sims9".

All predefined strategies set *PrefDefSize* := 8, *DeductionSize* := 1000, *PathCompression* := false and *LoopLimit* := 0. For the remaining parameter settings see the tables below.

	Default	Easy	Hard	Felsch	HLT	CT
Compact	10	100	10	10	10	100
Workspace	4	1	10	4	4	4
FillFactor	0	1	0	0	1	1
CTFactor	1000	0	1000	1000	0	1000
RTFactor	2000/1	1000	1	0	1000	0
Style	R_CR	R	CR	C	R	C
Lookahead	0	0	0	0	1	0
Mendelsohn	false	false	false	false	false	false
RelationsInSubgroup	-1	0	-1	-1	0	0
RowFilling	true	true	true	false	true	false
PrefDefMode	3	0	3	3	0	0
DeductionMode	4	0	4	4	0	4

	RT	Sims1	Sims3	Sims5	Sims7	Sims9
Compact	100	10	10	10	10	10
Workspace	4	4	4	4	4	4
FillFactor	1	1	1	1	1	1
CTFactor	0	0	0	0	0	1000
RTFactor	1000	1000	1000	1000	1000	0
Style	R	R	Rt	R	Rt	C
Lookahead	0	0	0	0	0	0
Mendelsohn	false	false	false	true	true	false
RelationsInSubgroup	0	0	0	0	0	0
RowFilling	false	true	true	true	true	false
PrefDefMode	0	0	0	0	0	0
DeductionMode	0	0	4	0	4	4

AddRelator(~P, w): GrpFPCosetEnumProc, GrpFPElt

Add a word w in the generators of the group G underlying the coset enumeration process P to the defining relations of G. This means that a coset enumeration process P for the cosets of H in G is transformed into a coset enumeration process for the cosets of $\pi(H)$ in $\pi(G)$, where $\pi : G \to \pi(G)$ is an epimorphism with kernel $\langle w \rangle^G$. (Here $\langle w \rangle^G$ denotes the normal closure in G of the subgroup of G generated by w.)

`AddSubgroupGenerator(~P, w): GrpFPCosetEnumProc, GrpFPElt`

Add an element w of the group G underlying the coset enumeration process P to the generators of the subgroup. This means that a coset enumeration process P for the cosets of H in G is transformed into a coset enumeration process for the cosets of $\langle H, w\rangle$ in G, where $\langle H, w\rangle$ denotes the subgroup of G generated by H and w.

`SetProcessParameters(~P: parameters): GrpFPCosetEnumProc`

Change enumeration parameters of the coset enumeration process P. The set of parameters accepted by this function is the same as for the function *CosetEnumerationProcess*; see there for a description.

All parameters which are not explicitly changed or modified by selecting one of the predefined strategies retain their old values.

It should be noted that it is not possible to decrease the workspace allocated by a coset enumeration process, once an enumeration has been started. However the workspace can be extended without invalidating any information contained in the process.

Starting and Restarting an Enumeration

There are several ways of starting and restarting an enumeration for a coset enumeration process, which retain information from previous enumerations to a varying extent.

`StartEnumeration(~P: parameters): GrpFPCosetEnumProc`

Start a new enumeration for P. All information in P is discarded. This function can be called at any time for an existing coset enumeration process. The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function *SetProcessParameters* before calling `StartEnumeration`.) The set of parameters accepted by this function is the same as for the function *CosetEnumerationProcess*; see there for a description.

`RedoEnumeration(~P: parameters): GrpFPCosetEnumProc`

Restart an enumeration for P. All information in P is retained and the enumeration is restarted at coset number 1. This function can be called for any coset enumeration process, which contains a valid coset table. (If P does not contain a valid coset table, a call to `RedoEnumeration` causes a runtime error. Use the function *CanRedoEnumeration* to check whether a call to `RedoEnumeration` is legal for a certain coset enumeration process.) Note that the coset table of P need not be complete to use this function.

This function is intended for the case where additional relators and/or subgroup generators have been introduced. The coset table contained in P is still valid. However, the additional data may allow the enumeration to compete, or cause a collapse to a smaller index.

The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function *SetProcessParameters* before calling `RedoEnumeration`.) The set of parameters accepted by this function is the same as for the function *CosetEnumerationProcess*; see there for a description.

`CanRedoEnumeration(P): GrpFPCosetEnumProc -> BoolElt`

Returns `true`, if a call to *RedoEnumeration* is legal for the coset enumeration process P.

`ContinueEnumeration(~P: parameters): GrpFPCosetEnumProc`

Continue an enumeration for P, which has been interrupted because some limit was exceeded. All information in P is retained and the enumeration is restarted at the coset where the previous enumeration was stopped. This function can only be called for a coset enumeration process, if the previous enumeration produced a valid coset table and the subgroup underlying P has not been changed since then. (Otherwise, a call to `ContinueEnumeration` causes a runtime error. Use the function *CanContinueEnumeration* to check whether a call to `ContinueEnumeration` is legal for a certain coset enumeration process.) Note that the coset table of P need not be complete to use this function.

This function is intended for the case where an enumeration stopped without producing a finite index. This function allows to continue the enumeration with modified enumeration parameters

with the minimal possible overhead.

The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function *SetProcessParameters* before calling ContinueEnumeration.) The set of parameters accepted by this function is the same as for the function *CosetEnumerationProcess*; see there for a description.

CanContinueEnumeration(P): GrpFPCosetEnumProc -> BoolElt

Returns true, if a call to *ContinueEnumeration* is legal for the coset enumeration process P.

ResumeEnumeration(~P: parameters): GrpFPCosetEnumProc

Resume or start an enumeration for P in the "cheapest" way permitted by the state of the coset enumeration process P. The call

```
ResumeEnumeration(~P : parameters);
```

is equivalent to

```
if CanContinueEnumeration(P) then
            ContinueEnumeration(~P : parameters);
         elif CanRedoEnumeration(P) then
            RedoEnumeration(~P : parameters);
         else
            StartEnumeration(~P : parameters);
         end if;
```

The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function *SetProcessParameters* before calling ResumeEnumeration.) The set of parameters accepted by this function is the same as for the function *CosetEnumerationProcess*; see there for a description.

Accessing Information

CosetsSatisfying(P : parameters): GrpFPCosetEnumProc -> { GrpFPElt }

CosetSatisfying(P : parameters): GrpFPCosetEnumProc -> { GrpFPElt }

```
First      : RngIntElt                    Default: 1
Last       : RngIntElt                    Default: 0
Limit      : RngIntElt                    Default: 0
Normalizing: BoolElt                      Default: false
Order      : RngIntElt                    Default: 0
Print      : RngIntElt                    Default: 0
```

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, these functions return a set of words representing cosets which satisfy the conditions defined in the parameters.

A call to CosetSatisfying is equivalent to calling CosetsSatisfying with *Limit* set to 1 and *First* set to 2 (unless a higher value for *First* is specified explicitly), i.e. the function returns when the first coset, other than H, satisfying the specified conditions has been found.

This parameter determines at which coset the search for coset representatives satisfying the designated conditions is started. The coset H always is coset number 1, i.e. setting First to 2 restricts the search to non-trivial cosets.

For the function CosetSatisfying, the minimal possible value for this parameter is 2; setting it to 1 does not cause an error but will be ignored.

This parameter determines at which coset the search for coset representatives satisfying the designated conditions is stopped. If it is set to 0 (default), all active cosets (starting from *First*) are searched.

If this parameter is set to $l > 0$, the search for coset representatives is aborted as soon as l cosets satisfying the designated condition have been found. If it is set to 0 (default for `CosetsSatisfying`), no limit is in force.

This parameter is not available for the function `CosetSatisfying`.

If true, select coset representatives x such that $x^{-1}h_i x$ is known to be contained in H from the information in the coset table of P for every generator h_i of H. (i.e. x is recognised as an element of the normaliser of H in G.)

Select coset representatives x such that x^n is known to be contained in H from the information in the coset table of P.

If the value of this parameter is positive, print the coset representatives found to satisfy the designated conditions.

These functions can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to any of these functions causes a runtime error. You can use the function *HasValidCosetTable* to check whether a call to these functions is legal for a certain coset enumeration process.

`CosetTable(P): GrpFPCosetEnumProc -> Map`

The current coset table of P as a map $f : \{1, \ldots, r\} \times G \to \{0, \ldots, r\}$, where G and H are the finitely presented group and the subgroup underlying P, respectively, and r is number of active cosets. $f(i, x)$ is the coset to which coset i is mapped under the action of $x \in G$. The value 0 is only included in the codomain if the coset table is not complete, and it denotes that the image of i under x is not known.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `CosetTable` causes a runtime error. Use the function *HasValidCosetTable* to check whether a call to `CosetTable` is legal for a certain coset enumeration process. Note that the coset table of P need not be complete to use this function.

`HasValidCosetTable(P): GrpFPCosetEnumProc -> BoolElt`

Returns `true`, if P contains a valid (but not necessarily closed) coset table, i.e. if a call to *CosetTable* is legal for the coset enumeration process P.

`HasClosedCosetTable(P): GrpFPCosetEnumProc -> BoolElt`

`HasCompleteCosetTable(P): GrpFPCosetEnumProc -> BoolElt`

Returns `true`, if P contains a closed, valid coset table.

Note that if `HasClosedCosetTable` returns `true` for a coset enumeration process P, then in particular a call to *CosetTable* is legal for P.

`ExcludedConjugate(P): GrpFPCosetEnumProc -> GrpFPElt`

`ExcludedConjugates(P): GrpFPCosetEnumProc -> { GrpFPElt }`

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, these functions return a set E containing either at most one word (`ExcludedConjugate`) or all words (`ExcludedConjugates`) of the form $g_i^{-1}h_j g_i$, where g_i is a generator of G and h_j is a generator of H, such that $g_i^{-1}h_j g_i$ is not known to lie in H from the information contained in the coset table of P.

If E is empty, then H is a normal subgroup of G. Otherwise the addition of elements of E to the generators of H may yield a larger subgroup of the normal closure of H in G. In the case of a non-complete coset table it may happen, however, that excluded conjugates are found which actually lie in H.

These functions can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `ExcludedConjugate` or `ExcludedConjugates` causes a runtime error. You can use the function *HasValidCosetTable* to check whether a call to these functions is legal for a certain coset enumeration process.

ExistsCosetSatisfying(P : parameters): GrpFPCosetEnumProc -> BoolElt, GrpFPElt

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, return whether or not there exists a coset other than H, which satisfies the conditions defined in the parameters. If such a coset exists, a representing word is returned as second return value. This function accepts the same set of parameters as the function *CosetSatisfying*; see there for a description.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to ExistsCosetSatisfying causes a runtime error. You can use the function *HasValidCosetTable* to check whether a call to ExistsCosetSatisfying is legal for a certain coset enumeration process.

ExistsExcludedConjugate(P): GrpFPCosetEnumProc -> BoolElt, GrpFPElt

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, return whether or not there exists a word of the form $g_i^{-1}h_jg_i$, where g_i is a generator of G and h_j is a generator of H, such that $g_i^{-1}h_jg_i$ is not known to lie in H from the information contained in the coset table of P. If the answer is positive, such a word is returned as second return value.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to ExistsExcludedConjugate causes a runtime error. You can use the function *HasValidCosetTable* to check whether a call to ExistsExcludedConjugate is legal for a certain coset enumeration process.

Note that the coset table of P need not be complete to call this function. A negative result of ExistsExcludedConjugate always implies that H is a normal subgroup of G, even if the coset table of P is not complete. In the case of a non-complete coset table it may happen, however, that excluded conjugates are found which actually lie in H.

ExistsNormalisingCoset(P): GrpFPCosetEnumProc -> BoolElt, GrpFPElt

ExistsNormalizingCoset(P): GrpFPCosetEnumProc -> BoolElt, GrpFPElt

Returns true, if an element of $G \setminus H$ which normalises H can be found from the coset table contained in P. (Here, G and H are the finitely presented group and the subgroup underlying P, respectively.) If the answer is positive, such an element is returned as second return value. This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to ExistsNormalisingCoset causes a runtime error. You can use the function *HasValidCosetTable* to check whether a call to ExistsNormalisingCoset is legal for a certain coset enumeration process.

Note that the coset table of P need not be complete to call this function. However, no conclusion can be drawn from a negative result in the case of a non-complete coset table.

Group(P): GrpFPCosetEnumProc -> GrpFP

Returns the group underlying P as a finitely presented group.

Index(P): GrpFPCosetEnumProc -> RngIntElt

Returns the index of H in G. (Here, G and H denote the finitely presented group and the subgroup underlying P, respectively.)

This function can only be called, if the last enumeration done for P has completed successfully with a finite index. Otherwise, a call to Index will cause a runtime error. Use the function *HasValidIndex* to check whether a call to Index is legal for a certain coset enumeration process.

HasValidIndex(P): GrpFPCosetEnumProc -> BoolElt

Returns true, if the last enumeration done for P has completed successfully with a finite index, i.e., if a call to *Index* is legal for the coset enumeration process P.

MaximalNumberOfCosets(P): GrpFPCosetEnumProc -> RngIntElt

Returns the maximal number of cosets which were simultaneously active during the last enumeration done for P, or 1 if no enumeration has been done for P.

This function may be useful for assessing the performance of a certain set of enumeration parameters.

Subgroup(P): GrpFPCosetEnumProc -> GrpFP

Returns the subgroup H underlying the coset enumeration process P. H is returned as a subgroup of G, where G is the finitely presented group underlying P.

TotalNumberOfCosets(P): GrpFPCosetEnumProc -> RngIntElt

Returns the total number of cosets defined during the last enumeration done for P, or 1 if no enumeration has been done for P.

This function may be useful for assessing the performance of a certain set of enumeration parameters.

Example: Interactive Coset Table for Harada-Norton (ex-bfc054)

In the Harada-Norton sporadic simple group

$$\begin{aligned} < x, a, b, c, d, e, f, g \mid\ & x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, \\ & (x, a), (x, g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, \\ & (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, \\ & (ef)^3, (eg)^2, \\ & (fg)^3, \\ & (b, xbx), \\ & (a, edcb), (a, f)dcbdcd, (ag)^5, \\ & (cdef, xbx), (b, xcdefx), (cdef, xcdefx) > \end{aligned}$$

we want to construct the coset table for the subgroup generated by x, b, c, d, e, f, g interactively. First, we create a coset enumeration process. Since the index of the chosen subgroup is 1 140 000, we request a coset limit of 1 200 000. Then we start the enumeration.

```
> HN<x, a, b, c, d, e, f, g> :=
>     FPGroup< x, a, b, c, d, e, f, g |
>              x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>              (x, a), (x, g),
>              (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>              (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>              (d*e)^3, (d*f)^2, (d*g)^2,
>              (e*f)^3, (e*g)^2,
>              (f*g)^3,
>              (b, x*b*x), (a, e*d*c*b), (a, f)*d*c*b*d*c*d,
>              (a*g)^5, (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>              (c*d*e*f, x*c*d*e*f*x)
>           >;
> H := sub<HN | x,b,c,d,e,f,g >;
> P := CosetEnumerationProcess(HN, H : CosetLimit := 1200000, Print := true);
> StartEnumeration(~P);
Overflow
 (a=1110331 r=58605 h=101193 n=1200001;
  l=5444 c=105.36;
  m=1110331 t=2001537)
```

The enumeration could not be completed successfully. Though P does contain a valid coset table, but this coset table fails to be closed.

```
> HasValidCosetTable(P);
true
> HasClosedCosetTable(P);
false
```

We extract the coset table map, check its domain and its codomain, and determine the position of the first "hole" in the coset table.

```
> ct := CosetTable(P);
> Domain(ct) : Minimal;
Cartesian Product<{ 1 .. 1110331 }, GrpFP: HN>
> Codomain(ct);
{ 0 .. 1110331 }
> row := 1;
> while forall(col){ gen : gen in {x, a, b, c, d, e, f, g}
>                        | ct(<row, gen>) ne 0 } do
>    row +:= 1;
> end while;
> row;
41881
> col;
x
```

We change the enumeration parameters for the process P, selecting the predefined strategy *Hard*. Since this predefined strategy sets a workspace of 10 000 000, we must expressly override this, if we want the old value to be retained. Because the workspace size never is decreased once an enumeration has been started, we can retain the old value simply by setting *Workspace* to 0; this overrides the setting done by selecting the predefined strategy, but actually doesn't change the process' workspace. We then continue the enumeration with the new set of parameters, after checking that this is legal.

```
> SetProcessParameters(~P : Strategy := "Hard",
>                           Workspace := 0);
> CanContinueEnumeration(P);
true
> ContinueEnumeration(~P);
INDEX = 1140000
 (a=1140000 r=58512 h=1144534 n=1144534;
  l=70 c=6.50;
  m=1140000 t=2035739)
```

We have obtained a closed coset table. Note that the codomain of the new coset table map does not contain 0.

```
> HasClosedCosetTable(P);
true
> ct := CosetTable(P);
> Domain(ct) : Minimal;
Cartesian Product<{ 1 .. 1140000 }, GrpFP: HN>
> Codomain(ct);
{ 1 .. 1140000 }
```

Example: Copying a Coset Enumeration Process (ex-e1cc15)

First, we create a coset enumeration process for the trivial subgroup of the finite group $G := \langle a, b \mid a^8, b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ and start the enumeration.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | >;
> P := CosetEnumerationProcess(G, H : Print := true);
> StartEnumeration(~P);
INDEX = 10752
```

(continues on next page)

(continued from previous page)

```
 (a=10752 r=57263 h=1 n=57263;
  l=292 c=0.11;
  m=47825t=57262)
```

We want to enumerate the cosets of the two non-trivial subgroups of G generated by $a^{-1} * b$ and a^2, respectively. To do this, we create two copies of the coset enumeration process `P` and use the function *AddSubgroupGenerator* for each of them. Since the copies inherit all information contained in `P`, we then can call the function *RedoEnumeration* to enumerate the cosets of the two non-trivial subgroups, making use of the existing coset table.

```
> P1 := P;
> AddSubgroupGenerator(~P1, a^-1*b);
> Subgroup(P1);
Finitely presented group on 2 generators
Generators as words in group G
    $.1 = Id(G)
    $.2 = a^-1 * b
> CanRedoEnumeration(P1);
true
> RedoEnumeration(~P1);
INDEX = 3584
 (a=3584 r=57263 h=1 n=57263;
  l=49 c=0.02;
  m=47825 t=57262)
>
> P2 := P;
> AddSubgroupGenerator(~P2, a^2);
> Subgroup(P2);
Finitely presented group on 2 generators
Generators as words in group G
    $.1 = Id(G)
    $.2 = a^2
> CanRedoEnumeration(P2);
true
> RedoEnumeration(~P2);
INDEX = 2688
 (a=2688 r=57263 h=1 n=57263;
  l=37 c=0.02;
  m=47825 t=57262)
```

Finally, we are interested in the quotient of G by the normal closure of the subgroup generated by a^4 and want to enumerate the cosets of the image of the subgroup generated by a^2 in this quotient. Since this is the subgroup used in `P2`, we create a copy of `P2` and add the relation a^4. Again, we are able to make use of information obtained earlier, by continuing the inherited enumeration.

```
> P3 := P2;
> AddRelator(~P3, a^4);
> CanContinueEnumeration(P3);
true
> ContinueEnumeration(~P3);
INDEX = 84
 (a=84 r=57263 h=1 n=57263;
 l=2 c=0.00;
 m=47825 t=57262)
```

We extract the quotient and its subgroup from the process `P3`, using the appropriate access functions.

```
> G3<a3,b3> := Group(P3);
> G3;
Finitely presented group G3 on 2 generators
Relations
    a3^8 = Id(G3)
    b3^7 = Id(G3)
    (a3 * b3)^2 = Id(G3)
    (a3^-1 * b3)^3 = Id(G3)
    a3^4 = Id(G3)
> H3<u3, v3> := Subgroup(P3);
> H3;
Finitely presented group H3 on 2 generators
Generators as words in group G3
    u3 = Id(G3)
    v3 = a3^2
```

Example: Normaliser Element from a Partial Coset Table (ex-ef1fe7)

Consider the subgroup H of the (infinite) group $G := \langle a, b \mid b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ generated by a. We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
 (a=957026 r=415230 h=415230 n=999999;
  l=3553 c=2.38;
  m=960050 t=999998)
```

The enumeration produces a valid (albeit not complete) coset table.

```
> HasValidCosetTable(P);
true
> HasCompleteCosetTable(P);
false
```

Even the partial coset table is sufficient to find an element in the normaliser of H in G.

```
> found, elt := ExistsNormalisingCoset(P);
> found;
true
> elt;
b^-4 * a * b^-2
```

Example: Excluded Conjugates in a Coset Table (ex-0f03e7)

Consider again the subgroup H of the (infinite) group $G := \langle a, b \mid b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ generated by a. We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
 (a=957026 r=415230 h=415230 n=999999;
```

(continues on next page)

(continued from previous page)

```
  l=3553 c=2.38;
  m=960050 t=999998)
```

We check, whether the coset table exhibits excluded conjugates.

```
> ExistsExcludedConjugate(P);
true a^b
```

It does. This means in particular, that H is not normal in G. We create a copy `P1` of the coset enumeration process `P`, extend the subgroup of `P1` by the excluded conjugates found in the previous step and restart the enumeration for `P1`.

```
> P1 := P;
> for c in ExcludedConjugates(P) do
>   AddSubgroupGenerator(~P1, c);
> end for;
> RedoEnumeration(~P1);
INDEX = 1 (a=1 r=2 h=2 n=2; l=2 c=0.94; m=960050 t=999998)
```

The new subgroup is equal to G. In particular, the normal closure of H in G is the whole of G.

We return to the coset enumeration process `P` and check whether we can find a non-trivial element $x \in \mathtt{N}_G(H)$ such that $x^2 \in H$.

```
> ExistsCosetSatisfying(P : Order := 2, Normalizing := true);
true b^-4 * a * b^-2
```

We can. In fact, $b^{-4} \cdot a \cdot b^{-2}$ is the only non-trivial coset which is known to satisfy this condition…

```
> CosetsSatisfying(P : Order := 2, Normalizing := true);
{ Id(G), b^-4 * a * b^-2 }
```

…and we can't find in a similar way a non-trivial element $x \in \mathtt{N}_G(H)$ such that $x^3 \in H$.

```
> ExistsCosetSatisfying(P : Order := 3, Normalizing := true);
false
```

Note the difference in the output of *CosetSatisfying* and *CosetsSatisfying*: The former takes into account only cosets other than H, whereas the latter (unless we set the parameter *First*) includes the coset H, which obviously satisfies the specified conditions.

```
> CosetSatisfying(P : Order := 3, Normalizing := true);
{}
> CosetsSatisfying(P : Order := 3, Normalizing := true);
{ Id(G) }
```

80.6.4 Implicit Invocation of the Todd-Coxeter Algorithm

Several functions working with finitely presented groups at some point require a coset table of a subgroup and may invoke a coset enumeration indirectly, e.g. the function `H meet K` or the function *Normaliser*. The default behaviour for such implicitly called coset enumerations is the same as the one for coset enumerations invoked explicitly, e.g. using the function *ToddCoxeter*.

If such an implicitly called coset enumeration fails to produce a closed coset table, the calling function may terminate with a runtime error.

Experienced users can control the behaviour of indirectly invoked coset enumerations with a set of global parameters. These global parameters are valid for all implicitly called coset enumerations. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive*

Coset Enumeration. Note that coset enumerations which are *explicitly* invoked, e.g. by a call to the function *Index*, are not affected by this global set of parameters. Parameters for these functions have to be specified in the function call.

SetGlobalTCParameters(: parameters)

This function sets the parameter values used for indirect invocations of the Todd-Coxeter coset enumeration procedure. The parameters accepted and their default values are the same as for the function *CosetEnumerationProcess* described in Subsec. *Interactive Coset Enumeration*.

UnsetGlobalTCParameters()

This function restores the default values for the parameters used for indirect invocations of the Todd-Coxeter coset enumeration procedure. For a description of the meanings of the parameters and their default values, see *CosetEnumerationProcess* in Subsec. *Interactive Coset Enumeration*.

Example: Implicit Coset Enumeration (ex-de08eb)

We consider again the Harada-Norton simple group with the presentation

$$\begin{aligned} < x, a, b, c, d, e, f, g \mid\ & x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, (x,a), (x,g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, (ef)^3, (eg)^2, (fg)^3, (b, xbx), (a, edcb), \\ & (a,f)dcbdcd, (ag)^5, (cdef, xbx), (b, xcdefx), (cdef, xcdefx) > \end{aligned}$$

and the subgroup H generated by x, b, c, d, e, f, g.

```
> HN<x, a, b, c, d, e, f, g> :=
>     FPGroup< x, a, b, c, d, e, f, g |
>              x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>              (x, a), (x, g),
>              (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>              (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>              (d*e)^3, (d*f)^2, (d*g)^2,
>              (e*f)^3, (e*g)^2,
>              (f*g)^3,
>              (b, x*b*x),
>              (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>              (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>              (c*d*e*f, x*c*d*e*f*x) >;
> H := sub<HN | x,b,c,d,e,f,g >;
```

H has index 1,140,000 in HN. Using the default settings, the normaliser of H in HN cannot be computed.

```
> N := Normaliser(HN, H);

>> N := Normaliser(HN, H);
                  ^
Runtime error in 'Normaliser': Coset table is not closed
```

We change the global parameters for implicitly called coset enumerations and try again.

```
> SetGlobalTCParameters( : Strategy := "Hard");
> N := Normaliser(HN, H);
```

With these parameters, the computation works. We see that H is self-normalising in HN.

```
> Index(HN, N);
1140000
> IsSelfNormalising(HN, H);
true
```

80.7 Coset Spaces and Tables

Let $G =< X|R >$ be a finitely presented group. Suppose that H is a subgroup of G having finite index m. Let $V = \{c_1(= H), c_2, \ldots, c_m\}$ denote the set of distinct right cosets of H in G. This set admits a natural G-action $f : V \times G \to V$ where

$$f :< c_i, x >= c_k \quad \Longleftrightarrow \quad c_i * x = c_k,$$

for $c_i \in V$ and $x \in G$. The set V together with this action is a G-set called a *right coset space* for H in G. The action may also be represented using a coset table T. If certain of the products $c_i * x$ are unknown, the corresponding images under f are undefined. In this case, T is not closed, and V is called an *incomplete coset space* for H in G.

80.7.1 Coset Tables

A coset table is represented in Magma as a mapping. Given a finitely-presented group G and a subgroup H, the corresponding (right) coset table is a mapping $f : \{1, \ldots, r\} \times G \to \{0, \ldots, r\}$, where r is the index of H in G. $f(i, x)$ is the coset to which coset i is mapped under the action of $x \in G$. The value 0 is only included in the codomain if the coset table is not closed, and it denotes that the coset is not known.

CosetTable(G, H: parameters): GrpFP, GrpFP -> Map

The (right) coset table for G over subgroup H, constructed by means of the Todd-Coxeter procedure. If the coset table does not close then the codomain will include the value 0.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function *CosetEnumerationProcess* described in Subsec. *Interactive Coset Enumeration*.

CosetTableToRepresentation(G, T): GrpFP, Map -> Map, GrpPerm, Grp

Given a coset table T for a subgroup H of G, construct the permutation representation of G given by its action on the cosets of H, using the columns of T. The function returns:

(a) The homomorphism $f : G \to P$;

(b) The permutation group image P;

(c) The kernel K of the action (a subgroup of G).

CosetTableToPermutationGroup(G, T): GrpFP, Map -> GrpPerm

Given a coset table T for a subgroup H of G, construct the permutation group image of G given by its action on the cosets of H, using the columns of T. This is the second return value of *CosetTableToRepresentation*(G, T).

Example: Coset Table in the Infinite Dihedral Group (ex-ae13a4)

Consider the infinite dihedral group.

```
> G<a,b> := DihedralFPGroup(0);
> G;
Finitely presented group G on 2 generators
Relations
    b^2 = Id(G)
    (a * b)^2 = Id(G)
```

We define a subgroup S of G and compute the coset table map for S in G.

```
> S := sub<G|a*b, a^10>;
> ct := CosetTable(G, S);
> ct;
```

(continues on next page)

(continued from previous page)

```
Mapping from: Cartesian Product<{ 1 .. 10 }, GrpFP: G>
        to { 1 .. 10 }
     $1  $2 -$1
 1.   2   2   3
 2.   4   1   1
 3.   1   4   5
 4.   6   3   2
 5.   3   6   7
 6.   8   5   4
 7.   5   8   9
 8.  10   7   6
 9.   7  10  10
10.   9   9   8
```

When printing the coset table, the action of the generators and of the non-trivial inverses of generators on the enumerated transversal is shown in table form.

Using the coset table, we now construct the permutation representation of G on the cosets of S in G. We assign the representation (a homomorphism), the image (a permutation group of degree $[G : S] = 10$) and the kernel of the permutation representation (a subgroup of G).

```
> fP, P, K := CosetTableToRepresentation(G, ct);
> fP;
Homomorphism of GrpFP: G into GrpPerm: P, Degree 10,
Order 2^2 * 5 induced by
    a |--> (1, 2, 4, 6, 8, 10, 9, 7, 5, 3)
    b |--> (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)
```

Note that the images of a and b correspond to the first two columns of the printed coset table above.

```
> P;
Permutation group P acting on a set of cardinality 10
Order = 20 = 2^2 * 5
    (1, 2, 4, 6, 8, 10, 9, 7, 5, 3)
    (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)
> K;
Finitely presented group K
Index in group G is 20 = 2^2 * 5
Subgroup of group G defined by coset table
```

Now, we define a subgroup of infinite index in G and compute a coset table for it.

```
> H := sub<G|b>;
> ct := CosetTable(G, H);
```

Of course, the coset table cannot be complete; note that 0 is in its codomain, indicating unknown images of cosets.

```
> ct;
Mapping from: Cartesian Product<{ 1 .. 1333331 }, GrpFP: G>
        to { 0 .. 1333331 }
```

80.7.2 Coset Spaces: Induced Homomorphism

`CosetAction(G, H): Grp, Grp -> Hom(Grp), GrpPerm, Grp`

Given a subgroup H of the group G, this function constructs the permutation representation ϕ of G given by the action of G on the cosets of H. It returns:

(a) The homomorphism ϕ;

(b) The image group $\phi(G)$.

(c) (if possible) the kernel of ϕ.

The permutation representation is obtained by using the Todd-Coxeter procedure to construct the coset table for H in G. Note that G may be an infinite group: it is only necessary that the index of H in G be finite.

`CosetAction(V): GrpFPCos -> Hom(Grp), GrpPerm`

Construct the permutation representation L of G given by the action of G on the coset space V. It returns the permutation representation ϕ (as a map) and its image.

`CosetImage(G, H): Grp, Grp -> GrpPerm`

Construct the image of G given by its action on the (right) coset space of H in G.

`CosetImage(V): GrpFPCos -> GrpPerm`

Construct the image of G as defined by its action on the coset space V.

`CosetKernel(G, H): GrpFP, GrpFP -> GrpFP`

The kernel of G in its action on the (right) coset space of H in G. (Only available when the index of H in G is very small).

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

`CosetKernel(V): GrpFPCos -> GrpFP`

The kernel of G in its action on the (right) coset space V. (Only available when the index of the subgroup H of G defining the coset space is very small).

Example: Induced Homomorphism for Co1 `(ex-a27ede)`

The first Conway group has a representation as the image of the group

$$\begin{aligned} G = < a,b,c,d,e,f,g,h \mid\ & a^2, b^2, c^2, d^2, e^2, f^2, g^2, h^2, \\ & (ab)^3, (ac)^2, (ad)^2, (ae)^4, (af)^2, (ag)^2, (ah)^3, \\ & (bc)^5, (bd)^2, (be)^2, (bf)^2, (bg)^4, (bh)^4, \\ & (cd)^3, (ce)^3, (cf)^4, (cg)^2, (ch)^2, \\ & (de)^2, (df)^3, (dg)^2, (dh)^2, \\ & (ef)^6, (eg)^2, (eh)^2, \\ & (fg)^4, (fh)^6, \\ & (gh)^2, \\ & a(cf)^2 = (adfh)^3 = b(ef)^3 = (baefg)^3 = 1, \\ & (cef)^7 = d(bh)^2 = d(aeh)^3 = e(bg)^2 = 1 > \end{aligned}$$

under the homomorphism defined by the action of G on the cosets of the subgroup $H = < a,b,c,d,e,f,g,(adefcefgh)^{39} >$. The permutation group can be constructed as follows:

```
> F<s, t, u, v, w, x, y, z> := FreeGroup(8);
> G<a, b, c, d, e, f, g, h> := quo<F | s^2, t^2, u^2, v^2, w^2, x^2, y^2, z^2,
> (s*t)^3, (s*u)^2, (s*v)^2, (s*w)^4, (s*x)^2, (s*y)^2, (s*z)^3,
> (t*u)^5, (t*v)^2, (t*w)^2, (t*x)^2, (t*y)^4, (t*z)^4,
> (u*v)^3, (u*w)^3, (u*x)^4, (u*y)^2, (u*z)^2,
> (v*w)^2, (v*x)^3, (v*y)^2, (v*z)^2,
> (w*x)^6, (w*y)^2, (w*z)^2,
> (x*y)^4, (x*z)^6,
> (y*z)^2,
> s*(u*x)^2 = (s*v*x*z)^3 = t*(w*x)^3 = (t*s*w*x*y)^3 = 1,
> (u*w*x)^7 = v*(t*z)^2 = v*(s*w*z)^3 = w*(t*y)^2 = 1>;
> H := sub< G | a, b, c, d, e, f, g, (a*d*e*f*c*e*f*g*h)^39 >;
> V := CosetSpace(G, H: FillFactor := 100000);
> Co1 := CosetImage(V);
> Degree(Co1);
    98280
```

Example: Permutation Representation of G2(3) (ex-6d4ee4)

The group $G_2(3)$ is a homomorph of the fp-group G defined below. We construct a permutation representation $G1$ for $G_2(3)$ on 351 points, and then compute the subgroup generated by the images of the first four generators of G in $G1$. (The functions applied to permutation groups are described in Chapter Permutation Groups.)

```
> F<a,b,c,d,y,x,w> := FreeGroup(7);
> z := y*c*a^2*b;
> u := x*d;
> t := w*c*a*d*b^2*c;
> G<a,b,c,d,y,x,w>, g :=
>            quo< F | a^4, b^4, c^2, d^2, (a,b), (a*c)^2, (b*c)^2,
>                       (c*d)^2, d*a*d*b^-1, y^3, (a^-1*b)^y*d*a^-1*b^-1,
>                       (c*d*a^-1*b)^y*b^-1*a*d*c, a*d*y*d*a^-1*y, x^3,
>                       a^x*b^-1, b^x*a*b, c^x*c, (x*d)^2, (u*z)^6, w^3,
>                       (w,y), (a*b)^w*b^-1*a*d*c, (c*d*a^-1*b)^w*d*c*b^2,
>                       (b^2*c*d)^w*a^-1*b^-1, (c*a^2*b*w)^2,
>                      (a^-1*b)^w*d*a^-1*b^-1, (t*u)^3 >;
> z1 := g(z);
> u1 := g(u);
> t1 := g(t);

> H := sub< G | z1*a^2*b^2, u1*c, t1*a^2*b^2 >;
> f, G1, K := CosetAction(G, H);
> Degree(G1);
351

> print Order(G1), FactoredOrder(G1);
4245696 [ <2, 6>, <3, 6>, <7, 1>, <13, 1> ]

> CompositionFactors(G1);
    G
    |  G(2, 3)
    1

> S := sub< G1 | f(a), f(b), f(c), f(d) >;
> BSGS(S);
> S;
Permutation group S of degree 351
Order = 64 = 2^6
```

Thus, the images of a, b, c, and d in G1 generate the Sylow 2-subgroup.

80.7.3 Coset Spaces: Construction

The indexed (right) coset space V of the subgroup H of the group G is a G-set consisting of the set of integers $\{1, ..., m\}$, where i represents the right coset c_i of H in G. The action of G on this G-set is that induced by the natural G-action

$$f : V \times G \to V$$

where

$$f :< c_i, x >= c_k \quad \Longleftrightarrow \quad c_i * x = c_k,$$

for $c_i \in V$ and $x \in G$. If certain of the products $c_i * x$ are unknown, the corresponding images under f are undefined, and V is called an *incomplete coset space* for H in G.

`CosetSpace(G, H: parameters): GrpFP, GrpFP -> GrpFPCos`

This function attempts to construct a coset space for the subgroup H in the group G by means of the Todd-Coxeter procedure. If the enumeration fails to complete, the function returns an incomplete coset space. The coset space is returned as an indexed right coset space. For a sample application of this function see Example *Working with a Coset Space*.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function *CosetEnumerationProcess* described in Subsec. *Interactive Coset Enumeration*.

`RightCosetSpace(G, H: parameters): GrpFP, GrpFP -> GrpFPCos`

`LeftCosetSpace(G, H: parameters): GrpFP, GrpFP -> GrpFPCos`

The explicit right coset space of the subgroup H of the group G is a G-set containing the set of right cosets of H in G. The elements of this G-set are the pairs $< H, x >$, where x runs through a transversal for H in G. Similarly, the explicit left coset space of H is a G-set containing the set of left cosets of H in G, represented as the pairs $< x, H >$. These functions use the Todd-Coxeter procedure to construct the explicit right (left) coset space of the subgroup H of the group G. For a sample application see Example *Working with a Coset Space*.

Experienced users can control the Todd-Coxeter procedure invoked by these functions with a wide range of parameters. Both functions accept the same parameters as the function *CosetEnumerationProcess* described in Subsec. *Interactive Coset Enumeration*.

80.7.4 Coset Spaces: Elementary Operations

`H * g: GrpFP, GrpFPElt -> GrpFPCosElt`

Right coset of the subgroup H of the group G, where g is an element of G (as an element of the right coset of H).

`C * g: GrpFPCosElt, GrpFPElt -> GrpFPCosElt`

Coset to which the right coset C of the group G is mapped by the (right) action of g, where g is an element of G.

`C * D: GrpFPCosElt, GrpFPCosElt -> GrpFPCosElt`

Given two right cosets of the same normal subgroup H of the group G, return the right coset that is the product of C and D.

`g in C: GrpFPElt, GrpFPCosElt -> BoolElt`

Return `true` if element g of group G lies in the coset C.

`g notin C: GrpFPElt, GrpFPCosElt -> BoolElt`

Return `true` if element g of group G does not lie in the coset C.

`C1 eq C2: GrpFPCosElt, GrpFPCosElt -> BoolElt`

Returns `true` if the coset $C1$ is equal to the coset $C2$.

`C1 ne C2: GrpFPCosElt, GrpFPCosElt -> BoolElt`

Returns `true` if the coset $C1$ is not equal to the coset $C2$.

80.7.5 Accessing Information

`# V: GrpFPCos -> RngIntElt`

The cardinality of the coset space V.

`Action(V): GrpFPCos -> Map`

The mapping $V \times G \to V$ giving the action of G on the coset space V. This mapping is a coset table.

`<i, w> @ T: GrpFPCosElt, GrpFPElt, Map -> GrpFPElt`

`T(i, w): Map, GrpFPCosElt, GrpFPElt -> GrpFPElt`

The image of coset i as defined in the coset table T, under the action of word w.

`ExplicitCoset(V, i): GrpFPCos, RngIntElt -> GrpFPCosElt`

The element of the explicit coset space that corresponds to indexed coset i.

`IndexedCoset(V, w): GrpFPCos, GrpFPElt -> GrpFPCosElt`

The element of the indexed coset space V to which the element w of G corresponds.

`IndexedCoset(V, C): GrpFPCos, GrpFPCosElt -> GrpFPCosElt`

The element of the indexed coset space V to which the explicit coset C of G corresponds.

`Group(V): GrpFPCos -> GrpFP`

The group G for which V is a coset space.

`Subgroup(V): GrpFPCos -> GrpFP`

The subgroup H of G such that V is a coset space for G over H.

`IsComplete(V): GrpFPCos -> BoolElt`

Returns `true` if the coset space V is complete.

`ExcludedConjugates(V): GrpFPCos -> { GrpFPElt }`

`ExcludedConjugates(T): Map -> { GrpFPElt }`

Given a partial or complete coset space V for the group G over the subgroup H, or a coset table T corresponding to this coset space, this function returns the set of words $E = \{g_i^{-1} h_j g_i \mid g_i$ a generator of G, h_j a generator of H, and, modulo V, $g_i^{-1} h_j g_i$ does not lie in $H\}$. If E is empty, then H is a normal subgroup of G, while if E is non-empty, the addition of the elements of E to the generators of H will usually be a larger subgroup of the normal closure of H. This function may be used with incomplete coset spaces for G over H; it may then happen that some of the elements of E actually lie in H but there is insufficient information for this to be detected. This function is normally used in conjunction with the Todd-Coxeter algorithm when seeking some subgroup having index sufficiently small so that the Todd-Coxeter procedure completes. The conjugates are returned as a set of words.

`Transversal(G, H): GrpFP, GrpFP -> {@ GrpFPElt @}, Map`

`RightTransversal(G, H): GrpFP, GrpFP -> {@ GrpFPElt @}, Map`

Given a finitely presented group G and a subgroup H of G, this function returns

(a) A set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

These functions may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

Example: Accessing a Coset Table (ex-e36cb8)

Consider the infinite dihedral group.

```
> G<a,b> := DihedralFPGroup(0);
```

We define a subgroup H of index 10 in G...

```
> H := sub< G | a*b, a^10 >;
> Index(G, H);
10
```

...and construct a right transversal for H in G and the associate transversal map.

```
> RT, transmap := Transversal(G, H);
> RT;
{@ Id(G), a, a^-1, a^2, a^-2, a^3, a^-3, a^4, a^-4, a^5 @}
> transmap;
Mapping from: GrpFP: G to SetIndx: RT
```

From this a left transversal is easily obtained:

```
> LT := {@ x^-1 : x in RT @};
> LT;
{@ Id(G), a^-1, a, a^-2, a^2, a^-3, a^3, a^-4, a^4, a^-5 @}
```

We construct the coset table and check whether the enumeration of the cosets is compatible to the enumeration of the right transversal RT.

```
> ct := CosetTable(G, H);
> forall(culprit){ i : i in [1..Index(G, H)]
>                    | ct(1, RT[i]) eq i};
true
```

It is. Thus, we can very easily define a function $RT \times G \to RT$, describing the action of G on the set of right cosets of H in G.

```
> action := func< r, g | RT[ct(Index(RT, r), g)] >;
```

Note that the definition of the function `action` relies on the fact that the computed right transversal and its enumeration match the ones internally used for the coset table ct.

```
> action(Id(G), b);
a
```

i.e. $H * b = H * a$.

```
> action(a^-4, a*b);
a^4
```

i.e. $Ha^{-4} * (ab) = Ha^4$.

Example: Working with a Coset Space (ex-886b74)

Consider the group $G =< a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 >$ and the subgroup H of G, generated by a^2 and $a^{-1}b$.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo< F | x^8, y^7, (x*y)^2, (x^-1*y)^3 >;
> H := sub< G | a^2,a^-1*b >;
```

We construct an indexed right coset space V and an explicit right coset space Vr for H in G.

```
> V := CosetSpace(G, H);
> Vr := RightCosetSpace(G, H);
```

The coset H always has index 1.

```
> trivial_coset := ExplicitCoset(Vr, 1);
> trivial_coset;
<GrpFP: H, Id(G)>
> IndexedCoset(V, trivial_coset);
1
```

We now pick a coset...

```
> coset := ExplicitCoset(Vr, 42);
> coset;
<GrpFP: H, a^-1 * b^-1 * a^3 * b^-1>
```

...multiply from the right with b...

```
> coset * b;
<GrpFP: H, a^-1 * b^-1 * a^3>
```

...and check where this gets us in the indexed coset space V.

```
> IndexedCoset(V, coset * b);
23
```

Example: Derived Subgroup via Coset Enumeration (ex-13317b)

We present a function which computes the derived subgroup G' for the finitely presented group G. It assumes that the Todd-Coxeter procedure can enumerate the coset space of G' in G.

```
> function DerSub(G)
>
> /* Initially define S to contain the commutator of each pair of distinct
>  generators of G */
>
>     S := { (x,y) : x, y in Generators(G) | (x,y) ne Id(G) };
>
> /* successively extend S until it is closed under conjugation by the
>   generators of G */
>
>         repeat
>             V := CosetSpace(G, sub<G | S>);
>             E := ExcludedConjugates(V);
>             S := S join E;
>         until # E eq 0;
>     return sub<G | S>;
> end function;
```

Example: Adding Excluded Conjugates (ex-8d212e)

Given a subgroup H of the finitely presented group G, for which the Todd-Coxeter procedure does not complete, add excluded conjugates one at a time to the generators of G until a subgroup K is

reached such that either K is normal in G, or K has sufficiently small index for the Todd-Coxeter method to complete. The set *hgens* contains a set of generating words for H.

```
> function NormClosure(G, hgens)
> xgens := hgens;
> kgens := hgens;
> indx := 0;
> while # xgens ne 0 do
>       Include(~kgens, Representative(xgens));
>       V := CosetSpace(G, sub<G | kgens>);
>       if IsComplete(V) then break; end if;
>       xgens := ExcludedConjugates(V);
> end while;
> if IsComplete(V) then
>           print "The subgroup generated by", kgens, "has index", #V;
>           return kgens;
> else
>           print "The construction was unsuccessful";
>           return {};
> end if;
> end function;
```

80.7.6 Double Coset Spaces: Construction

DoubleCoset(G, H, g, K): GrpFP, GrpFP, GrpFPElt, GrpFP -> GrpFPDcosElt

The double coset $H * g * K$ of the subgroups H and K of the group G, where g is an element of G.

DoubleCosets(G, H, K): GrpFP, GrpFP, GrpFP -> { GrpFPDcosElt }

Set of double cosets $H * g * K$ of the group G.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

Example: Double Cosets in the Infinite Dihedral Group (ex-b7c1be)

Consider again the infinite dihedral group G...

```
> G<a,b> := DihedralFPGroup(0);
```

...and the subgroup H of index 10 in G.

```
> H := sub< G | a*b, a^10 >;
```

The set of H-H double cosets in G can be obtained with the statement

```
> DoubleCosets(G, H, H);
{ <GrpFP: H, Id(G), GrpFP: H>, <GrpFP: H, a^5, GrpFP: H>,
  <GrpFP: H, a^4, GrpFP: H>, <GrpFP: H, a^2, GrpFP: H>,
  <GrpFP: H, a, GrpFP: H>, <GrpFP: H, a^3, GrpFP: H> }
```

80.7.7 Coset Spaces: Selection of Cosets

CosetsSatisfying(T, S: parameters): Map, { GrpFPElt } -> { GrpFPCosElt }

CosetSatisfying(T, S: parameters): Map, { GrpFPElt } -> { GrpFPCosElt }

CosetsSatisfying(V, S: parameters): GrpFPCos, { GrpFPElt } -> { GrpFPCosElt }

CosetSatisfying(V, S: parameters): GrpFPCos, { GrpFPElt } -> { GrpFPCosElt }

```
First      : RngIntElt                    Default: 1
Last       : RngIntElt                    Default: #V
Limit      : RngIntElt                    Default: Infinity()
Normalizing: BoolElt                      Default: false
Order      : RngIntElt                    Default: 0
Print      : RngIntElt                    Default: 0
```

Given a fp-group G, and a partial or complete coset space V or coset table T for G over the subgroup H generated by the set of words S, these functions return representatives for the cosets of V which satisfy the conditions defined in the parameters. In the description of the parameters below, the symbol l will denote a Boolean value, while the symbol n will denote a positive integer in the range $[1, \#V]$.

The functions are not identical. **CosetsSatisfying** returns a set of coset representatives for V as defined in the parameters. **CosetSatisfying** is the same as **CosetsSatisfying** with the **Limit** parameter equal to 1; thus it returns a set containing a single coset representative, or an empty set if no cosets satisfy the conditions.

Start looking for coset representatives satisfying the designated conditions beginning with coset i of V.

Stop looking for coset representatives after examining coset j of V.

Terminate the search for coset representatives as soon as l have been found which satisfy the designated conditions. This parameter is not available for **CosetSatisfying**, since **Limit** is set to 1 for this function.

If **true**, select coset representatives x such that, modulo V, the word $x^{-1}h_1x$, ..., x_1h_sx is contained in H.

Select coset representatives x such that, modulo V, the word x^n is contained in H.

If $t > 0$, print the coset representatives found to satisfy the designated conditions.

Example: Selecting Cosets by a Condition (ex-c086e8)

Consider the braid group G on 4 strings with Artin generators a, b and c and the subgroup H of G generated by a and b.

```
> G<a,b,c> := BraidFPGroup(4);
> H := sub< G | a,b >;
```

We construct an – obviously incomplete – explicit right coset space for H in G.

```
> V := RightCosetSpace(G, H);
```

Using the function *CosetSatisfying*, we compute an element of G, not contained in H, which normalises H.

```
> cs := CosetSatisfying(V, Generators(Subgroup(V))
>                       : Normalizing := true, First := 2);
> cs;
{
    <GrpFP: H, c * b * a^2 * b * c>
}
> rep := c * b * a^2 * b * c;
```

The conjugates of a and b by this element had better be in H...

```
> rep^-1 * a * rep in H;
true
> rep^-1 * b * rep in H;
true
```

...OK.

80.7.8 Constructing a Presentation for a Subgroup

Let H be a subgroup of finite index in the finitely presented group G. It frequently happens that it is desirable to construct a set of defining relations for H from those of G. Such a presentation can be obtained either on a set of Schreier generators for H or on the given generators of H using the Reidemeister-Schreier rewriting technique [MKS76], if necessary together with extended coset enumeration [AR84, HKRR84]. We emphasise that if the user wishes only to determine the structure of the maximal abelian quotient of H, then the function *AbelianQuotientInvariants* should be used. In this case there is no need to first construct a presentation for H using the *Rewrite* function described below, since *AbelianQuotientInvariants* employs a special form of the Reidemeister-Schreier rewriting process which abelianises each relator as soon as it is constructed. Thus, compared to the function `Rewrite`, the function `AbelianQuotientInvariants` can be applied to subgroups of much larger index.

`Rewrite(G, H : parameters): GrpFP, GrpFP -> GrpFP, Map`

```
Simplify          : BoolElt                    Default: true
EliminationLimit  : RngIntElt                  Default: 100
ExpandLimit       : RngIntElt                  Default: 150
GeneratorsLimit   : RngIntElt                  Default: 0
LengthLimit       : RngIntElt                  Default: Infinity()
SaveLimit         : RngIntElt                  Default: 10
SearchSimultaneous: RngIntElt                  Default: 20
Iterations        : RngIntElt                  Default: 10000
Print             : RngIntElt                  Default: 0
```

Given a finitely presented group G and a subgroup H having finite index in G, return a group R isomorphic to H with a presentation on (some of) the Schreier generators of H in G. The group R will be created as a subgroup of G and defining relations of R on its generators will be available. Note that the generators of R will, in general, not correspond to the generators of H. The isomorphism from H onto R is returned as second return value.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

If this Boolean-valued parameter is given the value `true`, then the resulting presentation for H will be simplified (default). The function `Rewrite` returns a finitely presented group that is isomorphic to H. If simplification is requested (by setting `Simplify := true`) then the simplification procedures are invoked (see next section). These procedures perform a sequence of Tietze transformations which typically result in a considerable simplification of the presentation produced by the rewriting process. Alternatively, the user can set `Simplify := false` and then perform the simplification directly if desired. (See next section). If simplification is not requested as part of `Rewrite`, a small amount of simplification is performed on the presentation before it is returned.

These parameters control the simplification. See the description of *Simplify* for an explanation of these parameters.

`Rewrite(G, ~H : parameters): GrpFP, GrpFP`

```
Simplify: BoolElt                    Default: true
```

Given a finitely presented group G and a subgroup H having finite index in G, compute a defining set of relations for H on the existing generators, using extended coset enumeration and Reidemeister-Schreier rewriting, and change the presentation of H accordingly.

If the computation is successful, defining relations for H on its generators will be available at the end; any previously computed relations of H will be discarded. If the computation is unsuccessful, H is not changed. In any case, both the isomorphism type of H and its embedding into G as a subgroup is preserved by this function.

If this parameter is given the value `true` (default), then an attempt will be made to simplify the constructed set of relations by substring searches, that is, Tietze transformations not changing any generators. The generating set of H is not modified by this process.

Moreover, the extended coset enumeration can be controlled by a wide range of parameters. The function `Rewrite` – in addition to the parameter `Simplify` – accepts the same parameters as the function *`CosetEnumerationProcess`* described in Subsec. *Interactive Coset Enumeration*.

Example: Presentation for an Index 3 Subgroup `(ex-e234d8)`

Starting with the group G defined by

$$< x, y \mid x^2, y^3, (xy)^{12}, (xy)^6(xy^{-1})^6 >,$$

we construct a subgroup K of index 3 generated by the words x, yxy^{-1}, and $yxy^{-1}xy^{-1}xy$. We present the subgroup K, compute its abelian quotient structure, and then show that the class 30 2-quotient of K has order 2^{62}.

```
> G<x, y> := FPGroup< x, y | x^2, y^3, (x*y)^12, (x*y)^6*(x*y^-1)^6 >;
> G;

Finitely presented group G on 2 generators
Relations
       x^2 = Id(G)
       y^3 = Id(G)
       (x * y)^12 = Id(G)
       x * y * x * y * x * y * x * y * x * y * x * y * x * y^-1 * x *
          y^-1 * x * y^-1 * x * y^-1 * x * y^-1 * x * y^-1 = Id(G)

> K := sub< G | x, y*x*y^-1, y*x*y^-1*x*y^-1*x*y >;
> K;
Finitely presented group K on 3 generators
Generators as words in group G
       K.1 = x
       K.2 = y * x * y^-1
       K.3 = y * x * y^-1 * x * y^-1 * x * y
> Index(G, K);
3
> T := Rewrite(G, K);
> T;
Finitely presented group T on 3 generators
Generators as words in group G
    T.1 = x
    T.2 = y * x * y^-1
    T.3 = x^y
Relations
    T.1^2 = Id(T)
    T.2^2 = Id(T)
    T.3^2 = Id(T)
    (T.3 * T.2 * T.1 * T.3 * T.2)^2 = Id(T)
    (T.1 * T.3 * T.2 * T.1 * T.3)^2 = Id(T)
```

(continues on next page)

(continued from previous page)

```
    (T.1 * T.2 * T.1 * T.3 * T.2)^2 = Id(T)
> AbelianQuotientInvariants(T);
[ 2, 2, 2 ]

> Q2 := pQuotient(T, 2, 30);
> FactoredOrder(Q2);
[ <2, 62> ]
```

Example: Presentation on New Generators (ex-490fe5)

In this example we illustrate how the function *Rewrite* can be used to obtain a presentation of a finitely presented group on a different set of generators.

We start with a presentation of $L_2(7)$ on two generators x and y.

```
> F<x,y> := FPGroup< x, y | x^3 = 1, y^3 = 1, (x*y)^4 = 1,
>                          (y*y^x)^2 = y^x*y >;
```

The group is also generated by the elements $a = (xy)^2$ and $b = y$.

```
> H<a,b> := sub<F | (x*y)^2, y >;
> Index(F,H);
1
```

At the moment, no defining relations of $H \cong F$ on the generators a and b are known.

```
> H;
Finitely presented group H on 2 generators
Index in group F is 1
Generators as words in group F
    a = (x * y)^2
    b = y
```

We apply the function *Rewrite* to H as a subgroup of F in order to compute defining relations on the generators a and b.

```
> Rewrite(F, ~H);
> H;
Finitely presented group H on 2 generators
Index in group F is 1
Generators as words in group F
    a = (x * y)^2
    b = y
Relations
    a^2 = Id(H)
    b^3 = Id(H)
    (a * b)^7 = Id(H)
    (a * b^-1 * a * b)^4 = Id(H)
    (b * a * b^-1 * a * b * a)^4 = Id(H)
```

The last relation turns out to be redundant; a and b are standard generators for $L_2(7)$.

```
> Order(DeleteRelation(H,5)) eq Order(H);
true
```

80.8 Subgroups of Finite Index

The functions in this section are concerned with the construction of subgroups of finite index. We first describe a method for computing all subgroups whose index does not exceed some (modest) integer bound. The next family of functions are concerned with constructing new subgroups of finite index from one or more known ones.

80.8.1 Low Index Subgroups

If an fp-group G has subgroups of (small) finite index then there algorithms which can be used to find such subgroups and then use them to determine some properties of G.

LowIndexSubgroups(G, R : parameters): GrpFP, RngIntElt -> [GrpFP]

LowIndexSubgroups(G, R : parameters): GrpFP, RngIntElt -> [{ GrpFPElt }]

LowIndexSubgroups(G, R: parameters): GrpFP, <RngIntElt, RngIntElt> -> [GrpFP]

LowIndexSubgroups(G, R: parameters): GrpFP, <RngIntElt, RngIntElt> -> [{ GrpFPElt }]

Given a finitely presented group G (possibly the free group), and an expression R defining a positive integer range (see below), determine the conjugacy classes of subgroups of G whose indices lie in the range specified by R. The subgroups are generated by systematically building all coset tables consistent with the defining relations for G and which satisfy the range condition R. The argument R is one of the following:

(a) An integer n representing the range $[1, n]$;

(b) A tuple $< a, b >$ representing the range $[a, b]$;

The generation of subgroups can be controlled by a set of parameters described below. The returned sequence contains the subgroups found and is sorted in order of increasing index in G.

Example: Tutte's 8-Cage and the Conder Graph (ex-2e2f0b)

(Peter Lorimer) The two graphs known as Tutte's 8-cage and the Conder graph may be constructed as the Cayley graphs of two conjugacy classes of subgroups having index 10 in the finitely presented group

$$< q, r, s, h, a \mid h^3 = a^2 = p^2 = 1, p^h = p, p^a = q, q^h = r, r^a = s,$$
$$h^s = h^{-1}, r^h = pqr, sr = pqrs, pq = qp, pr = rp, ps = sp, qr = rq, qs = sq > .$$

We use the low index function to construct these subgroups.

```
> G<p, q, r, s, h, a> := FPGroup<p, q, r, s, h, a |
>                         h^3 = a^2 = p^2 = 1, p^h = p, p^a = q,
>                         q^h = r, r^a = s, h^s = h^-1, r^h = p * q * r,
>                         s * r = p * q * r * s, p * q = q * p,
>                         p * r = r * p, p * s = s * p, q * r = r * q,
>                         q * s = s * q>;
> LowIndexSubgroups(G, <10, 10>);
[
       Finitely presented group on 6 generators
       Index in group G is 10 = 2 * 5
       Generators as words in group G
              $.1 = p
              $.2 = s
              $.3 = h
              $.4 = q^-2
              $.5 = a * h^-1 * a * r^-1
              $.6 = a * h * a * h^-1 * a * q^-1,
```

(continues on next page)

(continued from previous page)

```
       Finitely presented group on 6 generators
       Index in group G is 10 = 2 * 5
       Generators as words in group G
              $.1 = p
              $.2 = s
              $.3 = h
              $.4 = q^-2
              $.5 = a * h^-1 * a * r^-1
              $.6 = a * h * a * h^-1 * a
]
```

Example: Conjugacy Classes via Low Index Subgroups (ex-18108c)

A fairly surprising application for the low index subgroup algorithm is the enumeration of the conjugacy classes of a finite fp-group. In this example, we consider the group

$$G \simeq \mathrm{PGL}_2(9) = \langle a, b | a^2, b^3, (ab)^8, [a,b]^5, [a,(ba)^3 b^{-1}]^2 \rangle.$$

```
> G<a,b> := FPGroup< a,b | a^2, b^3, (a*b)^8, (a,b)^5,
>                         (a,(b*a)^3*b^-1)^2  >;
> Order(G);
720
```

In an infinite fp-group, finding all classes of subgroups up to an index of 720 by applying the low index subgroup algorithm, would be extremely hard. In the case of the finite group G, however, we succeed.

```
> time sgG := LowIndexSubgroups(G, Order(G));
Time: 31.859
> #sgG;
26
```

We get a list of 26 representatives of the conjugacy classes of subgroups. For every representative, its index in G and a set of generating words are known. We just have a look at two of them.

```
> sgG[10];
Finitely presented group on 2 generators
Index in group G is 60 = 2^2 * 3 * 5
Generators as words in group G
    $.1 = b
    $.2 = a * b * a * b * a * b^-1 * a * b * a * b * a * b^-1 * a
       * b^-1 * a
> sgG[21];
Finitely presented group on 1 generator
Index in group G is 180 = 2^2 * 3^2 * 5
Generators as words in group G
    $.1 = (b * a)^2
```

The function *LowIndexSubgroups* constructs all conjugacy classes of subgroups of G satisfying the following two conditions:

(i) The index of each subgroup is in the range defined by R;

(ii) If the parameter `Subgroup` defines a subgroup H, then at least one subgroup in each conjugacy class contains the subgroup H.

The subgroups are returned as a sequence of subgroups G, unless otherwise specified by the parameter `GeneratingSets` (see below). The sequence is sorted by increasing index of the subgroups in G. The subgroups are constructed using an algorithm due to Sims [Sim94, sect. 5.6]. This algorithm constructs

the coset tables by using a backtrack algorithm. At a given position in the coset table, coset definitions are made systematically. Once a new definition has been made, the group relations are traced in an attempt to deduce further entries or to infer that this partial table will not extend to a table corresponding to a new class of subgroups. When either it cannot define a new entry, or when a complete table has been constructed, the algorithm backtracks to try the next possibility (this may introduce a new row, increasing the index). This algorithm may also be run as a process in such a way that the subgroups are returned one at a time, thereby allowing the user to analyze each subgroup as soon as it is found.

```
ColumnMajor: BoolElt                    Default: false
```

If `ColumnMajor` is set `false` (*default*), then the location for a new definition in the coset table is determined by searching the table in row major order for undefined entries. If `ColumnMajor` is set `true`, then the position for a new definition is determined by searching the table in column major order. If the presentation for G contains explicit relators expressing the fact that certain of the generators have large order, then the presentation should be organized so that these generators appear first and the column major order should be selected for new coset definition. This strategy often leads to greatly improved performance.

```
GeneratingSets: BoolElt                 Default: false
```

The conjugacy classes of subgroups are returned in the form of a sequence of sets of words, where the i-th set is a generating set for a representative subgroup from the i-th conjugacy class of subgroups satisfying the given conditions. This is a much more compact representation than returning the subgroups as a sequence of actual *subgroups* of G and should be used when a very large number of subgroups is expected, as there may be insufficient space to store each of them as a subgroup.

```
Limit: RngIntElt                        Default: Infinity()
```

Terminate after finding n conjugacy classes of subgroups satisfying the designated conditions.

```
Long: [ RngIntElt ]                     Default: []
```

This option enables the user to designate certain of the defining relators for G as *long relators*. The relators of G are numbered from 1 to r, in the order they appear in the *quo< F | R >*- or *FPGroup< X | R >*-constructors. The value L of `Long` is a subset of the integer set $\{1, \ldots, r\}$. Magma interprets the relators whose numbers appear in L as long relators. A relator designated as long is not used during the construction of a coset table. Rather, it is applied once a complete table has been found. There is some evidence to suggest that better performance is achieved in those groups having one or more very long relators by deferring application of these relators until such time as a complete coset table has been obtained.

```
Print: RngIntElt                        Default: 0
```

A description of each class of subgroups may be printed immediately after it is constructed. The value n assigned to the `Print` parameter specifies just what information is to be printed, according to the following rules:

$n = 0$
: No printing (*default*).

$n = 1$
: For each class, print a heading and a set of generators for the class representative.

$n = 2$
: The information printed for $n = 1$, together with the permutation representation of G on the right cosets of the class representative.

$n = 3$
: The information printed for $n = 2$, together with generators for the normalizer N of the class representative, and a system of right coset representatives for N in G.

```
Subgroup: GrpFP                    Default: sub< G | >
```

By specifying a value H for `Subgroup`, only subgroups containing H will be constructed.

```
TimeLimit: RngIntElt                    Default: 0 (no limit)
```

A time limit in seconds. A value of 0 (default) means no limit.

LowIndexProcess(G, R : parameters): GrpFP, RngIntElt -> Process(Lix)

LowIndexProcess(G, R: parameters): GrpFP, < RngIntElt, RngIntElt > -> Process(Lix)

```
ColumnMajor    : BoolElt                        Default: false
GeneratingSets: BoolElt                         Default: false
Long           : [ RngIntElt ]                  Default: []
Print          : RngIntElt                      Default: 0
Subgroup       : GrpFP                          Default: sub< G | >
TimeLimit      : RngIntElt                      Default: 0 (no limit)
```

Create a low index subgroups *process*. This process may be used to create the conjugacy classes of proper subgroups one at time, with control being handed back to the Magma language processor each time a new class of subgroups is found. This function returns a process which is used by the function *NextSubgroup* to actually produce the subgroups. The arguments and parameters have the same interpretation as for the function *LowIndexSubgroups*, except that `Limit` is not available (since the same effect can be achieved by limiting the number of calls to *NextSubgroup*). Setting a time limit for a process P limits the total amount of time spent in calls to *NextSubgroup*. If the time limit is exceeded in a call to *NextSubgroup*, this call is aborted and P becomes invalid. Further attempts to access P will cause a runtime error. The function *IsValid* can be used to check whether a process is valid in order to avoid runtime errors in loops or user written functions.

NextSubgroup(~P): GrpFPLixProc

NextSubgroup(~P, ~G): GrpFPLixProc, GrpFP

Given a low index subgroups process P, construct the next conjugacy class of proper subgroups. The process P must have been previously created using the function *LowIndexProcess* and must be valid. Calling `NextSubgroup` for an empty process has no effect.

ExtractGroup(P): GrpFPLixProc -> GrpFP

Extract a representative subgroup for the conjugacy class currently defined by the low index process P. The subgroup extracted will be the one found by the previous invocation of *NextSubgroup*, or the first subgroup if *NextSubgroup* has never been invoked on this process. Note that `ExtractGroup` will not search for a new subgroup. If P is empty or invalid, a runtime error will result.

ExtractGenerators(P): GrpFPLixProc -> { GrpFPElt }

Extract a generating set for the representative subgroup of the conjugacy class currently defined by the low index process P. The subgroup extracted will be the one found by the previous invocation of *NextSubgroup*, or the first subgroup if *NextSubgroup* has never been invoked on this process. Note that `ExtractGenerators` will not search for a new subgroup. If P is empty or invalid, a runtime error will result.

IsEmpty(P): GrpFPLixProc -> BoolElt

Return `true` if the low index process P has already found all conjugacy classes of subgroups. If `IsEmpty` is called immediately following the creation of the low index process, then it will return the value `false` if there are no subgroups satisfying the specified conditions or advance P to the first such subgroup otherwise.

IsValid(P): GrpFPLixProc -> BoolElt

Return `true` if the low index process P is valid, that is, no limit has been exceeded. If `IsValid` is called immediately following the creation of the low index process, then it will return the value `false` if no subgroups satisfying the specified conditions can be found within the specified time or advance P to the first such subgroup otherwise.

Example: Low Index Subgroups of a Triangle Group (ex-fd99df)

We determine all conjugacy classes of subgroups having index at most 15 in the triangle group

$$< a, b \mid a^2, b^3, (ab)^7 > .$$

```
> G<a, b> := FPGroup< a, b | a^2, b^3, (a*b)^7 >;
> L := LowIndexSubgroups(G, 15: Print := 1);

Subgroup class 1        Index 7 Length 7        Subgroup generators :-
{ a, b * a * b^-1, b^-1 * a * b * a * b^-1 * a * b }

Subgroup class 2        Index 7 Length 7        Subgroup generators :-
{ a, b^-1 * a * b^-1 * a * b * a * b, b * a * b^-1 }

Subgroup class 3        Index 15        Length 15       Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b * a * b^-1 * a * b, b * a * b^-1 }

Subgroup class 4        Index 15        Length 15       Subgroup generators :-
{ a, b * a * b^-1, b^-1 * a * b^-1 * a * b * a * b^-1 * a * b * a * b }

Subgroup class 5        Index 14        Length 14       Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b * a * b * a * b^-1 * a * b, b * a * b^-1 }

Subgroup class 6        Index 14        Length 7        Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b * a * b^-1 * a * b * a * b^-1 * a * b, b *
a * b * a * b^-1 }

Subgroup class 7        Index 14        Length 14       Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b^-1 * a * b * a * b * a * b^-1 * a * b, b *
a * b * a * b^-1 }

Subgroup class 8        Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b^-1 * a * b * a * b * a * b^-1 * a * b^-1, b^-1
 * a * b * a * b }

Subgroup class 9        Index 15        Length 15       Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b^-1, b^-1 * a * b^-1 * a * b * a * b }

Subgroup class 10       Index 9 Length 9        Subgroup generators :-
{ a, b * a * b * a * b * a * b^-1 }

Subgroup class 11       Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b * a * b^-1 * a * b, b * a * b^-1 * a * b * a * b^-1 }

Subgroup class 12       Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b * a * b, b * a * b^-1 * a * b * a * b^-1 }

Subgroup class 13       Index 14        Length 7        Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b, b * a * b * a * b * a * b^-1 * a * b^-1 }

Subgroup class 14       Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b * a * b, b^-1 * a * b * a * b^-1 * a * b }

Subgroup class 15       Index 14        Length 14       Subgroup generators :-
{ a, b^-1 * a * b * a * b * a * b^-1 * a * b, b * a * b * a * b * a * b^-1 * a *
 b^-1 }
```

(continues on next page)

(continued from previous page)

```
Subgroup class 16       Index 8 Length 8        Subgroup generators :-
{ b, a * b * a * b * a * b^-1 * a }
```

Example: PSL(2,8) as a Quotient of a Triangle Group (ex-55eb8b)

In this example we illustrate the use of the low index subgroup process by using it to determine whether the simple group $\mathrm{PSL}(2,8)$ is a homomorphic image of the triangle group

$$< x, y \mid x^2, y^3, (xy)^7 > .$$

```
> F<x, y> := FreeGroup(2);
> G<x, y> := quo< F | x^2, y^3, (x*y)^7 >;
> LP := LowIndexProcess(G, 30);
> i := 0;
> while i le 100 and not IsEmpty(LP) do
>    H := ExtractGroup(LP);
>    NextSubgroup(~LP);
>    P := CosetImage(G, H);
>    if Order(P) eq 504 and IsSimple(P) then
>        print "The group G has L(2, 8) as a homomorphic image.";
>        print "It is afforded by the subgroup:-", H;
>        break;
>    end if;
>    i +:= 1;
> end while;
The group G has L(2, 8) as a homomorphic image.
It is afforded by the subgroup:-
Finitely presented group H on 4 generators
Index in group G is 28 = 2^2 * 7
Generators as words in group G
       H.1 = x
       H.2 = y * x * y^-1
       H.3 = y^-1 * x * y * x * y^-1 * x * y * x * y^-1 * x * y * x * y^-1 *
       x * y * x * y
       H.4 = y^-1 * x * y * x * y^-1 * x * y^-1 * x * y * x * y^-1 * x * y *
       x * y * x * y^-1 * x * y
```

Example: Proving a Group Is Infinite (ex-dd899e)

This example shows how the low index subgroup machinery may be used as part of a function trying to prove that a group is infinite:

```
> function MyIsInfinite(G)
>
>  // ...
>

>  // Low index subgroup approach: check whether an obviously
>  //    infinite subgroup can be found in reasonable time.
>  P := LowIndexProcess(G, 30 : TimeLimit := 5);
>  while IsValid(P) and not IsEmpty(P) do
>     H := ExtractGroup(P);
>     NextSubgroup(~P);
>     if 0 in AbelianQuotientInvariants(H) then
>        print "The group G has subgroup:-", H;
```

(continues on next page)

(continued from previous page)

```
>        print "whose abelian quotient is infinite";
>        print "Hence G is infinite.";
>        return true;
>     end if;
>  end while;
>  print "Low index approach fails; trying other methods...";
>
>  // ...
>
> end function;
```

We try the code fragment on the group

$$< x, z \mid z^3xz^3x^{-1}, z^5x^2z^2x^2 > .$$

```
> G<x, z> := FPGroup<x,z | z^3*x*z^3*x^-1, z^5*x^2*z^2*x^2 >;
> MyIsInfinite(G);
The group G has subgroup:-
Finitely presented group H on 4 generators
Index in group G is 4 = 2^2
Generators as words in group G
    H.1 = x
    H.2 = z * x * z
    H.3 = z^3
    H.4 = z * x^-1 * z * x * z^-1
whose abelian quotient has structure [ 2, 6, 0 ]
Hence G is infinite.
true
```

LowIndexNormalSubgroups(G, n: parameters): GrpFP, RngIntElt -> [Rec]

PrintLevel: RingIntElt		Default: 0
Simplify : MonStgElt		Default: "No"
Workspace : RngIntElt		Default: 1000000000

The normal subgroups of finitely presented group G up to index n, $n \leq 100\,000$. The subgroups are returned as a sequence of records (ordered by subgroup index) where the ith record contains fields

`Group`: A presentation of the ith normal subgroup.

`Index`: The index of the ith normal subgroup in G.

`Supergroups`: The set of positions in the sequence of the groups which are supergroups of the ith group.

This parameter may be set to 0, 1, or 2. At 0, the function prints no diagnostic output. At level 1, it outputs details of each normal subgroup being tested for further normal subgroups. Level 2 gives details of each test being performed on each normal subgroup.

This sets the `Workspace` parameter for calls of `pQuotient`.

The possible values are `"No"`, `"Yes"`, and `"LengthLimit"`. This determines the parameter values passed to the `Rewrite(G,H)` function, when this function is used. The value `"No"` sets parameter `Simplify:=false`. The value `"Yes"` sets parameter `Simplify:=true`. The value `"LengthLimit"` sets parameter `LengthLimit:=Index(G,H)`.

80.8.2 Operations for Subgroups of Finite Index

Most operations described in this subsection require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

`H ^ u: GrpFP, GrpFPElt -> GrpFP`

`Conjugate(H, u): GrpFP, GrpFPElt -> GrpFP`

Given an fp-group H and a word u in an fp-group K, such that H and K are subgroups of some common fp-group G and words in terms of the generators of G are known for the generators of both H and K, construct the subgroup of G obtained by conjugating H by u.

`H meet K: GrpFP, GrpFP -> GrpFP`

Given subgroups H and K, both of finite index in some fp-group G, return the subgroup which is the intersection of H and K.

This function requires closed coset tables for both, H and K in G.

`Core(G, H): GrpFP, GrpFP -> GrpFP`

Given a subgroup H of finite index in the fp-group G, construct the core of H in G.

This function requires a closed coset table for H in G.

`GeneratingWords(G, H): GrpFP, GrpFP -> { GrpFPElt }`

Given a subgroup H of the fp-group G, this function returns a set of words in the generators of G, generating H as a subgroup of G (assuming such words are known or can be constructed). Note that the returned generating set does not necessarily correspond to the internal generators of H. In particular, generating words obtained using the function `GeneratingWords` cannot be used to coerce elements from H to G.

`MaximalOvergroup(G, H): GrpFP, GrpFP -> GrpFP`

Given a subgroup H of finite index in the fp-group G, construct a maximal overgroup of H in G. A *maximal overgroup* of H is a maximal subgroup of G that contains H. If H is already maximal, the group G is returned.

This function requires a closed coset table for H in G.

`MinimalOvergroup(G, H): GrpFP, GrpFP -> GrpFP`

Given a subgroup H of finite index in the fp-group G, construct a minimal overgroup of H in G. A *minimal overgroup* of a subgroup H is a subgroup K of G such that K contains H as a maximal subgroup. If H is already maximal in G, the group G is returned.

This function requires a closed coset table for H in G.

`H ^ G: GrpFP, GrpFP -> GrpFP`

`NormalClosure(G, H): GrpFP, GrpFP -> GrpFP`

Given a subgroup H of finite index in the fp-group G, construct the normal closure of H in G.

This function requires a closed coset table for H in G.

`Normaliser(G, H): GrpFP, GrpFP -> GrpFP`

`Normalizer(G, H): GrpFP, GrpFP -> GrpFP`

Given a subgroup H of finite index in the fp-group G, construct the normaliser of H in G. For a sample application of this function, see Example *Implicit Coset Enumeration*.

This function requires a closed coset table for H in G.

`SchreierGenerators(G, H : parameters): GrpFP, GrpFP -> { GrpFPElt }`

`Simplify: BoolElt`	`Default: true`

Given a subgroup H of finite index in the fp-group G, return the Schreier generators for H as a set of words in G.

If the parameter `Simplify` is set to `true` (default), a heuristic method of eliminating redundant Schreier generators is applied. To switch this feature off, set `Simplify` to `false`.

This function requires a closed coset table for H in G.

`SchreierSystem(G, H): GrpFP, GrpFP -> {@ GrpFPElt @}, Map`

`Transversal(G, H): GrpFP, GrpFP -> {@ GrpFPElt @}, Map`

Given a subgroup H of finite index in the fp-group G, construct a (right) Schreier system of coset representatives for H in G. The function returns

(a) the Schreier system as a set of words in G;

(b) the corresponding Schreier coset function.

This function requires a closed coset table for H in G.

`Transversal(G, H, K): GrpFP, GrpFP, GrpFP -> {@ GrpFPElt @}, Map`

Given subgroups H and K, both of finite index in the fp-group G, return an indexed set of words which comprise a set of representatives for the double cosets HuK of H and K in G, as well as a map from G to the representatives. It should be noted that this function is evaluated by first constructing the right cosets of H in G and then computing the orbits of the cosets under the action of the generators of the subgroup K.

This function requires a closed coset table for H in G.

Example: Subgroup Constructions (ex-793825)

We illustrate some of the subgroup constructions by using them to construct subgroups of small index in the two-dimensional space group p4g which has the presentation

$$< r, s \,|\, r^2, s^4, (r,s)^2 > .$$

```
> p4g<r, s> := FPGroup< r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> p4g;
Finitely presented group p4g on 2 generators
Relations
    r^2 = Id(p4g)
    s^4 = Id(p4g)
    (r * s^-1 * r * s)^2 = Id(p4g)
```

We define two subgroups of $p4g$ and compute their indices in $p4g$.

```
> h := sub< p4g | (s^-1*r)^4, s*r >;
> k := sub< p4g | (s^-1*r)^2, (s*r)^2 >;
> Index(p4g, h);
8
> Index(p4g, k);
8
```

We construct the normal closure of h in $p4g$.

```
> n := NormalClosure(p4g, h);
> n;
Finitely presented group n on 6 generators
```

(continues on next page)

(continued from previous page)

```
Index in group p4g is 2
Generators as words in group p4g
    n.1 = (s^-1 * r)^4
    n.2 = s * r
    n.3 = r * s
    n.4 = r^-1 * s * r^2
    n.5 = s^2 * r * s^-1
    n.6 = r * s
```

Next, we construct a subgroup of $p4g$ containing h as maximal subgroup...

```
> m := MinimalOvergroup(p4g, h);
> m;
Finitely presented group m on 3 generators
Index in group p4g is 4 = 2^2
Generators as words in group p4g
    m.1 = (s^-1 * r)^4
    m.2 = s * r
    m.3 = (r * s)^2
```

...and a maximal subgroup of $p4g$ containing k.

```
> n := MaximalOvergroup(p4g, k);
> n;
Finitely presented group n on 4 generators
Index in group p4g is 2
Generators as words in group p4g
    n.1 = (s^-1 * r)^2
    n.2 = (s * r)^2
    n.3 = r
    n.4 = s^2
```

Finally, we construct a transversal in $p4g$ for the normaliser of h in $p4g$...

```
> T := Transversal(p4g, Normaliser(p4g, h));
> T;
{@ Id(p4g), r, s^-1, r * s @}
```

...compute the intersection of h and the conjugate of h by r...

```
> l := h meet h^r;
> l;
Finitely presented group l
Index in group p4g is 32 = 2^5
Subgroup of group p4g defined by coset table
```

...and construct the core of h in $p4g$.

```
> c := Core(p4g, h);
> c;
Finitely presented group c
Index in group p4g is 32 = 2^5
Subgroup of group p4g defined by coset table
```

Note, that the two subgroups l and c constructed last are defined as finite index subgroups of $p4g$ by a coset table and that there are no generators known for them. Generators can be obtained e.g. by using the function *GeneratingWords*. We show this for the subgroup l.

```
> GeneratingWords(p4g, l);
{ (s * r)^4, (s^-1 * r)^4 }
```

Once computed, these generators are memorised. Compare the result of printing l to the output obtained above.

```
> l;
Finitely presented group l on 2 generators
Index in group p4g is 32 = 2^5
Generators as words in group p4g
    l.1 = (s * r)^4
    l.2 = (s^-1 * r)^4
```

Example: Schreier Generators (ex-f2320c)

Consider the group G given by the presentation $\langle x, y \,|\, x^2, y^3, (xy)^7 \rangle$.

```
> G<x,y> := FPGroup< x,y | x^2, y^3, (x*y)^7 >;
```

We construct the subgroups of index less than or equal to 7 using the low index algorithm.

```
> L := LowIndexSubgroups(G, 7);
> L;
[
    Finitely presented group on 2 generators
    Index in group G is 1
    Generators as words in group G
        $.1 = x
        $.2 = y,

    Finitely presented group on 3 generators
    Index in group G is 7
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y^-1
        $.3 = y^-1 * x * y^-1 * x * y * x * y,

    Finitely presented group on 3 generators
    Index in group G is 7
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y^-1
        $.3 = y^-1 * x * y * x * y^-1 * x * y
]
```

We define a subgroup as the core of one of the subgroups of index 7. The function *Core* returns a subgroup of G defined by a coset table.

```
> H := Core(G, L[2]);
> H;
Finitely presented group H
Index in group G is 168 = 2^3 * 3 * 7
Subgroup of group G defined by coset table
```

A set of generators for H can be obtained e.g. with the function *SchreierGenerators*.

```
> sgH := SchreierGenerators(G, H);
> #sgH;
6
```

By default, *SchreierGenerators* returns a reduced generating set. The unreduced set of Schreier generators can be obtained by setting the value of the parameter `Simplify` to `false`.

```
> sgHu := SchreierGenerators(G, H : Simplify := false);
> #sgHu;
85
```

80.8.3 Properties of Subgroups

The operations described in this subsection all require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

`u in H: GrpFPElt, GrpFP -> BoolElt`

Given an fp-group H and a word u in an fp-group K, such that H and K are subgroups of some common fp-group G, H is of finite index in G, and words for the generators of K in terms of the generators of G are known, return `true` if u is an element of H and `false` otherwise.

This function requires a closed coset table for H in G.

`u notin H: GrpFPElt, GrpFP -> BoolElt`

Given an fp-group H and a word u in an fp-group K, such that H and K are subgroups of some common fp-group G, H is of finite index in G, and words for the generators of K in terms of the generators of G are known, return `true` if u is not an element of H and `false` otherwise.

This function requires a closed coset table for H in G.

`H eq K: GrpFP, GrpFP -> BoolElt`

Given subgroups H and K, both of finite index in the fp-group G, return `true` if H and K are equal and `false` otherwise.

This function may require closed coset tables for both, H and K in G.

`H ne K: GrpFP, GrpFP -> BoolElt`

Given subgroups H and K, both of finite index in the fp-group G, return `true` if H and K are not equal and `false` otherwise.

This function may require closed coset tables for both, H and K in G.

`H subset K: GrpFP, GrpFP -> BoolElt`

`IsSubgroup(H, K): GrpFP, GrpFP -> BoolElt`

Given subgroups H and K, both of finite index in the fp-group G, return `true` if H is contained in K and `false` otherwise.

This function requires a closed coset table for K in G.

`H notsubset K: GrpFP, GrpFP -> BoolElt`

Given subgroups H and K, both of finite index in the fp-group G, return `true` if H is not contained in K and `false` otherwise.

This function requires a closed coset table for K in G.

`IsConjugate(G, H, K): GrpFP, GrpFP, GrpFP -> BoolElt, GrpFPElt`

Given subgroups H and K, both of finite index in the fp-group G, return `true` if H and K are conjugate subgroups of G and `false` otherwise. If H and K are conjugate in G, a conjugating element is returned as second return value.

This function requires a closed coset table for both, H and K in G.

`IsNormal(G, H): GrpFP, GrpFP -> BoolElt`

Given a subgroup H of finite index in the fp-group G, return `true` if H is a normal subgroup of G and `false` otherwise.

This function requires a closed coset table for H in G.

`IsMaximal(G, H): GrpFP, GrpFP -> BoolElt`

Given a subgroup H of finite index in the fp-group G, return `true` if H is a maximal subgroup of G and `false` otherwise.

This function requires a closed coset table for H in G.

`IsSelfNormalizing(G, H): GrpFP, GrpFP -> BoolElt`

Given a subgroup H of finite index in the fp-group G, return `true` if H is a self-normalizing subgroup of G and `false` otherwise. For a sample application of this function, see Example *Implicit Coset Enumeration*.

This function requires a closed coset table for H in G.

Example: Subgroup Predicates in a Space Group `(ex-afcd46)`

We illustrate some of the subgroup predicates by applying them to some subgroups of the two-dimensional space group $p4g =< r, s | r^2, s^4, (r,s)^2 >$ from Example *Subgroup Constructions*.

```
> p4g<r, s> := FPGroup< r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> h := sub< p4g | (s^-1*r)^4, s*r >;
> k := sub< p4g | (s^-1*r)^2, (s*r)^2 >;
```

h and k have the same index in $p4g$...

```
> Index(p4g, h);
8
> Index(p4g, k);
8
```

...but they are not equal.

```
> h eq k;
false
```

We check for normality of h and k in $p4g$.

```
> IsNormal(p4g, h);
false
> IsNormal(p4g, k);
true
```

We construct the normal closure of h in $p4g$.

```
> n := NormalClosure(p4g, h);
```

We see that it is maximal...

```
> IsMaximal(p4g, n);
true
```

...and that it contains k.

```
> k subset n;
true
```

We define another subgroup of $p4g$.

```
> l := sub< p4g | (s*r)^4, s^-1*r >;
```

In fact, it is conjugate to h.

```
> IsConjugate(p4g, h, l);
true r^-1
```

i.e. $l = h^{r^{-1}}$. The intersection of h and $h^{r^{-1}}$ already yields the core of h in $p4g$.

```
> h meet l eq Core(p4g, h);
true
```

Example: Maximal Subgroups of the Hall-Janko Group (ex-bf7b3f)

The constructions of the previous section together with the Boolean function *IsMaximal* may be used to locate large maximal subgroups in a finite group. Consider the Hall-Janko group J_2, which may be defined by the presentation

$$\begin{aligned} < a, b, c \mid\ & a^3, b^3, c^3, abab^{-1}a^{-1}b^{-1}, (ca)^5, (cb)^5, \\ & (cb^{-1}cb)^2, a^{-1}baca^{-1}bac^{-1}a^{-1}b^{-1}ac^{-1}, \\ & aba^{-1}caba^{-1}c^{-1}ab^{-1}a^{-1}c^{-1} > . \end{aligned}$$

We examine subgroups generated by pairs of randomly chosen short words. Whenever we obtain a proper subgroup, if it is not already maximal we replace it by a maximal subgroup that contains it.

```
> J2<a, b, c> := FPGroup<a, b, c | a^3, b^3, c^3, a*b*a*b^-1*a^-1*b^-1, (c*a)^5,
>                                  (c*b)^5, (c*b^-1*c*b)^2,
>                                  a^-1*b*a*c*a^-1*b*a*c^-1*a^-1*b^-1*a*c^-1,
>                                  a*b*a^-1*c*a*b*a^-1*c^-1*a*b^-1*a^-1*c^-1>;
>
> Seen := { 0, 1};
> Found := { };
> Sgs := [ ];
> for i := 1 to 30 do
>     u := Random(J2, 1, 1);
>     v := Random(J2, 3, 5);
>     H := sub< J2 | u, v >;
>     Indx := Index(J2, H);
>     if Indx notin Seen then
>        Include(~Seen, Indx);
>          if not IsMaximal(J2, H) then
>            H := MaximalOvergroup(J2, H);
>         end if;
>        if Indx notin Found then
>            Include(~Sgs, H);
>            Include(~Seen, Indx);
>            Include(~Found, Indx);
>        end if;
>    end if;
> end for;
> Sgs;

[
      Finitely presented group on 3 generators
      Index in group J2 is 315 = 3^2 * 5 * 7
      Generators as words in group J2
             $.1 = b^-1
```

(continues on next page)

(continued from previous page)

```
            $.2 = a^-2 * c * a^-1
            $.3 = c,

      Finitely presented group on 3 generators
      Index in group J2 is 1008 = 2^4 * 3^2 * 7
      Generators as words in group J2
            $.1 = b^-1
            $.2 = c^-1 * a^-1 * c^-1 * b
            $.3 = a * c * b * c^-1 * a^-1 * c * b * c^-1,

      Finitely presented group on 3 generators
      Index in group J2 is 100 = 2^2 * 5^2
      Generators as words in group J2
            $.1 = c
            $.2 = (b * a^-1)^2
            $.3 = a * b^-1
]
```

Thus after taking 30 2-generator random subgroups, we have obtained three maximal subgroups, including the two largest maximal subgroups.

80.9 Finite FP-Groups

In this section a number of intrinsics that apply only to finite fp-group will be described. Of course most of the intrinsics designed for working with general fp-groups will also apply to finite fp-groups.

80.9.1 Concrete Representations

If a finitely-presented group is known to have moderate order, it may be useful to write it concretely as a permutation or PC-group. There are two utilities provided for this purpose.

`PermutationGroup(G): GrpFP -> GrpPerm, GrpHom`

Construct a faithful permutation representation of G, and the isomorphism from G to the representation. The algorithm first computes the order of G, then takes the regular representation of G, and then tries to reduce the degree. Thus, the intrinsic is restricted to groups of order a million (depending upon the amount of memory available).

`PCGroup(G): GrpFP -> GrpPC, GrpHom`

Construct a faithful PC-group representation of G, and the isomorphism from G to the representation. The algorithm first computes the order of G, then computes the soluble quotient of G with the order found. Thus the command is restricted to small soluble groups.

80.9.2 The Cayley Graph

Given a finite fp-group G of moderate order we can construct its Cayley graph. This graph can then be studied using the extensive tools provided in Magma for graphs.

`CayleyGraph(G): GrpFP -> GrphVertSet, GrphEdgeSeto`

```
Labelled: BoolElt Default                    Default: true
```

Given a finite group G defined on generating set X, construct the Cayley graph C of G relative to the generating set X. This graph is defined as follows: The vertices correspond to the elements of G and two vertices u, v are adjacent if and only if there exists an generator x in X such that $u * x = v$.

The optional parameter Labelled (Labelled := true by default) can be set to false to prevent the graph being labelled. If this is not done, then the vertices of C will be labelled with the appropriate

elements of G and the (directed) edge from u to v will be labelled with the appropriate generator x as defined above.

SchreierGraph(G, H): Grp, Grp -> Grph, GrphVertSet, GrphEdgeSet

Labelled: BoolElt Default	Default: true
Directed: BoolElt Default	Default: true

Given a finite fp-group G defined on the generating set X and a subgroup H of G, construct the Schreier coset graph S for G over H, relative to X. The graph S is defined as follows: The vertices correspond to the cosets of H in G, and two vertices u, v are adjacent in S if and only if there exists an element x in X such that $u * x = v$.

The graph returned may be a labelled or unlabelled version and directed or undirected. These versions are controlled by the parameters **Labelled** and **Directed**, which are both true by default.

Example: Cayley Graph of a Dihedral Group (ex-6ec7fa)

We define the Cayley graph of the dihedral group of order 12 and then construct its directed Cayley graph.

```
> G<x, y> := DihedralFPGroup(6);
> C, V, E := CayleyGraph(G);  C;
Digraph
Vertex  Neighbours

1       2 3 ;
2       5 6 ;
3       1 7 ;
4       1 7 ;
5       9 10 ;
6       2 3 ;
7       4 11 ;
8       4 11 ;
9       8 12 ;
10      5 6 ;
11      8 12 ;
12      9 10 ;
```

We next find the diameter d of the graph and then construct the word corresponding to a path of length d.

```
> d := Diameter(C);  d;
4
> p := DiameterPath(C);  p;
[ 1, 2, 5, 9, 8 ]
> word := &* [ Label(E![p[i], p[i+1]]) : i in [1..d]];
> word;
x^4
```

We next construct the automorphism group of the unlabelled, undirected graph underlying C.

```
> D := UnderlyingGraph(C);  D;
Graph
Vertex  Neighbours

1       2 3 4 ;
2       1 5 6 ;
3       1 6 7 ;
4       1 7 8 ;
```

(continues on next page)

(continued from previous page)

```
5       2 9 10 ;
6       2 3 10 ;
7       3 4 11 ;
8       4 9 11 ;
9       5 8 12 ;
10      5 6 12 ;
11      7 8 12 ;
12      9 10 11 ;

> A := AutomorphismGroup(D);  A;
Permutation group A acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (2, 4)(5, 8)(6, 7)(10, 11)
    (1, 2)(3, 6)(4, 5)(7, 10)(8, 9)(11, 12)
    (1, 3)(2, 6)(4, 7)(5, 10)(8, 11)(9, 12)
```

Finally we check that S has the dihedral group of order 12 as a subgroup.

```
> subs := Subgroups(A);
> P := PermutationGroup(G);
> exists(j){ i : i in [1..#subs] | IsIsomorphic(P, subs[i]`subgroup)};
true
> j;
25
```

So subgroup 25 is isomorphic to G. In fact there are four conjugacy classes of subgroups in A that are isomorphic to G.

80.10 Homomorphisms

For a general description of homomorphisms, we refer to Chapter Mappings. This section describes some special aspects of homomorphisms the domain of which is a finitely presented group.

80.10.1 General Remarks

The kernel of a homomorphism with a domain of type `GrpFP` can be computed using the function *Kernel*, if the codomain is of one of the types `GrpGPC`, `GrpPC` (cf. Chapter Finite Soluble Groups), `GrpAb` (cf. Chapter *Abelian Groups*), `GrpPerm` (cf. Chapter Permutation Groups), `GrpMat` (cf. Chapter Matrix Groups over General Rings), `ModAlg`, or `ModGrp` (cf. Chapter Modules over an Algebra and Group Representations), if the image is finite and its order sufficiently small. In this case, a regular permutation representation of the image is constructed and the kernel is created as a subgroup of the domain, defined by a coset table.

The kernel may also be computable, if the codomain is of the type `GrpFP`, the image is sufficiently small, and a presentation for the image is known.

If the kernel of a map can be computed successfully, forming preimages of substructures is possible. An attempt to compute the kernel of a map will be made automatically, if the preimage of a substructure of the codomain is to be computed.

Note, that trying to compute the kernel may be very time and memory consuming; use this feature with care.

80.10.2 Construction of Homomorphisms

`hom< P -> G | S >: Struct, Struct -> Map`

Returns the homomorphism from the fp-group P to the group G defined by the assignment S. S can be the one of the following:

(i) A list, sequence, or indexed set containing the images of the n generators $P.1, \ldots, P.n$ of P. Here, the i-th element of S is interpreted as the image of $P.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set, or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i \text{ -> } y_i$, where x_i is a generator of P and $y_i \in G$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of P. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

Note, that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of P. It is the user's responsibility to ensure that the arguments passed to the `hom`-constructor actually yield a well-defined homomorphism. For certain codomain categories, this may be checked using the function *IsSatisfied* described below.

`IsSatisfied(U, E): { RelElt }, [ GrpElt ] -> BoolElt`

`IsSatisfied(U, E): { GrpFPElt }, [ GrpElt ] -> BoolElt`

`IsSatisfied(U, E): [ RelElt ], [ GrpElt ] -> BoolElt`

`IsSatisfied(U, E): [ GrpFPElt ], [ GrpElt ] -> BoolElt`

The argument U is a set or sequence of either words belonging to an n-generator fp-group H or relations over H. The second argument E is a sequence of n elements $[e_1, \ldots, e_n]$ belonging to a group G for which both, multiplication and comparison of elements are possible. Using the mapping $H.i \to e_i$ $(i = 1, \ldots, n)$, we evaluate the relations given by U. If U is a set or sequence of relations, the left and right hand sides of each relation are evaluated and compared for equality. Otherwise, each word in U is evaluated and compared to the identity. If all relations are satisfied, `IsSatisfied` returns the Boolean value `true`. On the other hand, if any relation is not satisfied, `IsSatisfied` returns the value `false`.

This function may be used to verify the correctness of the definition of a homomorphism from an fp-group to a group in a category for which both, multiplication and comparison of elements are possible.

80.10.3 Accessing Homomorphisms

`w @ f: GrpFPElt, Map -> GrpElt`

`f(w): Map, GrpFPElt -> GrpElt`

Given a homomorphism whose domain is an fp-group G and an element w of G, return the image of w under f as an element of the codomain of f.

`H @ f: GrpFP, Map -> Grp`

`f(H): Map, GrpFP -> Grp`

Given a homomorphism whose domain is an fp-group G and a subgroup H of G, return the image of H under f as a subgroup of the codomain of f.

Some maps do not support images of subgroups.

`g @@ f: GrpElt, Map -> GrpFPElt`

Given a homomorphism whose domain is an fp-group G and an element g of the image of f, return the preimage of g under f as an element of G.

Some maps do not support inverse images.

`H @@ f: Grp, Map -> GrpFP`

Given a homomorphism whose domain is an fp-group G and a subgroup H of the image of f, return the preimage of H under f as a subgroup of G.

Some maps do not support inverse images. The inverse image of a subgroup of the codomain can only be computed if the kernel of the homomorphism can be computed, i.e. if the kernel has moderate index in the domain.

`Domain(f): Map -> Grp`

The domain of the homomorphism f.

`Codomain(f): Map -> Grp`

The codomain of the homomorphism f.

`Image(f): Map -> Grp`

The image or range of the homomorphism f as a subgroup of the codomain of f.

Some maps do not support this function.

`Kernel(f): Map -> Grp`

The kernel of the homomorphism f as a (normal) subgroup of the domain of f, represented by a coset table.

Some maps do not support this function. The kernel of a homomorphism can only be computed, if it has moderate index in the domain.

Example: Braid Group onto Sym(5) (ex-63663e)

For arbitrary $n > 0$, the symmetric group of degree $n+1$ is an epimorphic image of the braid group on n generators. In this example, we exhibit this relationship for $n = 4$.

We start with creating the braid group B on 5 strings, i.e. 4 Artin generators.

```
> B := BraidFPGroup(5);
> B;
Finitely presented group B on 4 generators
Relations
    B.1 * B.2 * B.1 = B.2 * B.1 * B.2
    B.1 * B.3 = B.3 * B.1
    B.1 * B.4 = B.4 * B.1
    B.2 * B.3 * B.2 = B.3 * B.2 * B.3
    B.2 * B.4 = B.4 * B.2
    B.3 * B.4 * B.3 = B.4 * B.3 * B.4
```

In the symmetric group of degree 5, we define 4 transpositions which will be the images of the generators of B.

```
> S := Sym(5);
> imgs := [ S!(1,2), S!(2,3), S!(3,4), S!(4,5) ];
```

In order to verify that this assignment actually gives rise to a well defined homomorphism, we check whether the potential images satisfy the defining relations of B.

```
> rels := Relations(B);
> rels;
[ B.1 * B.2 * B.1 = B.2 * B.1 * B.2, B.1 * B.3 = B.3 * B.1,
  B.1 * B.4 = B.4 * B.1, B.2 * B.3 * B.2 = B.3 * B.2 * B.3,
  B.2 * B.4 = B.4 * B.2, B.3 * B.4 * B.3 = B.4 * B.3 * B.4 ]
> IsSatisfied(rels, imgs);
true
```

They do. So we can define the homomorphism from B to S.

```
> f := hom< B->S | imgs >;
```

We see that f is surjective, i.e. S is an epimorphic image of B as claimed above.

```
> f(B) eq S;
true
```

We now check the kernel of f.

```
> Kernel(f);
Finitely presented group
Index in group B is 120 = 2^3 * 3 * 5
Subgroup of group B defined by coset table
```

Using the function *GeneratingWords* described later, we can obtain a set of generators of $\ker(f)$ as a subgroup of B.

```
> GeneratingWords(B, Kernel(f));
{ B.2^-2, (B.1 * B.2 * B.3^-1 * B.2^-1 * B.1^-1)^2, B.1^-2,
  (B.3 * B.4^-1 * B.3^-1)^2, (B.2 * B.3^-1 * B.2^-1)^2,
  (B.2 * B.3 * B.4^-1 * B.3^-1 * B.2^-1)^2, B.4^-2,
  (B.1 * B.2^-1 * B.1^-1)^2, B.3^-2,
  (B.1 * B.2 * B.3 * B.4^-1 * B.3^-1 * B.2^-1 * B.1^-1)^2 }
```

It is easy to see that all generators of $\ker(f)$ are conjugates of words of the form $g^{\pm 2}$, where g is a generator of B. We check this, using the normal closure constructor *ncl< G | L >* described later.

```
> Kernel(f) eq ncl< B | B.1^2, B.2^2, B.3^2, B.4^2 >;
true
```

Thus, the braid relations together with the relations $B.1^2, B.2^2, B.3^2, B.4^2$ are a set of defining relations for S.

80.10.4 Constructing Homomorphisms onto Finite Groups

This section describes functions for computing representatives of the classes of homomorphisms from a finitely presented group F to a finite group G modulo a group A of automorphisms of G.

Homomorphisms(F, G, A : parameters): GrpFP, GrpPerm, GrpPerm -> [HomGrp]

Homomorphisms(F, G : parameters): GrpFP, GrpPerm -> [HomGrp]

```
Surjective       : BoolElt                    Default: true
Limit            : RngIntElt                  Default: 0 (no limit)
TimeLimit        : RngIntElt                  Default: 0 (no limit)
CosetEnumeration: BoolElt                     Default: true
CacheCosetAction: BoolElt                     Default: true
```

Given a finitely presented group F and two permutation groups G and A with $G \triangleleft A$, return a sequence containing representatives of the classes of homomorphisms from F to G modulo automorphisms of G induced by elements of A. (That is, two homomorphisms f_1, $f_2 : F \to G$ are considered equivalent if there exists an element $a \in A$ such that $f_1(x) = f_2(x)^a$ for all $x \in F$.) The call `Homomorphisms(F, G)` is equivalent to `Homomorphisms(F, G, G)`. The function uses a backtrack algorithm testing certain conjugates of representatives of the A-classes in G as possible images of the generators of F. The following parameters are available for this function.

If this parameter is set to `true` (default), only epimorphisms are considered.

If this parameter is set to n, the function terminates after n classes of homomorphisms satisfying the specified conditions have been found. A value of 0 (default) means no limit.

A limit in seconds for the amount of time spent in the backtrack search for homomorphisms. The time spent in initial coset enumerations is *not* counted towards this limit. A value of 0 (default) means no limit.

If this parameter is set to `true` (default), a number of short coset enumerations are performed for each generator of F in order to check whether some A-classes in G can be ruled out as possible images for this generator. If a class can be ruled out, the complexity of the backtrack search may be reduced significantly. In situations where it seems unlikely that classes can be ruled out as possible generator images, experienced users may wish to turn this feature off in order to save the time spent on the coset enumerations.

The value of this parameter indicates whether the actions of A on the cosets of the centralisers of representatives of the A-classes in G are cached during the backtrack search. Setting this parameter to `true` (default) results in faster computations and is recommended for normal applications. When computing homomorphisms to large groups with many conjugate classes, this parameter can be set to `false` in order to reduce memory requirements at the expense of increased computing time.

Example: Finding Homomorphisms onto Finite Groups (ex-731266)

Consider the finitely presented group

$$F := < a, b, c \mid ac^{-1}bc^{-1}aba^{-1}b, abab^{-1}c^2b^{-1}, a^2b^{-1}(ca)^4cb^{-1} > .$$

```
> F := FPGroup< a,b,c | a*c^-1*b*c^-1*a*b*a^-1*b,
>                       a*b*a*b^-1*c^2*b^-1,
>                       a^2*b^-1*c*a*c*a*c*a*c*a*c*b^-1 >;
```

We use the function *Homomorphisms* to prove that F maps onto A_5.

```
> G := Alt(5);
> homs := Homomorphisms(F, G : Limit := 1);
> #homs gt 0;
true
```

Homomorphisms(F, G, A : parameters): GrpFP, GrpPC, GrpPC -> [HomGrp]

Homomorphisms(F, G : parameters): GrpFP, GrpPC -> [HomGrp]

```
Surjective: BoolElt                    Default: true
Limit     : RngIntElt                  Default: 0 (no limit)
```

Given a finitely presented group F and two finite polycyclic groups G and A with $G \triangleleft A$, return a sequence containing representatives of the classes of homomorphisms from F to G modulo automorphisms of G induced by elements of A. (That is, two homomorphisms f_1, $f_2 : F \to G$ are considered equivalent if there exists an element $a \in A$ such that $f_1(x) = f_2(x)^a$ for all $x \in F$.) The call `Homomorphisms(F, G)` is equivalent to `Homomorphisms(F, G, G)`. The following parameters are available for this function.

If this parameter is set to `true` (default), only epimorphisms are considered.

If this parameter is set to n, the function terminates after n classes of homomorphisms satisfying the specified conditions have been found. A value of 0 (default) means no limit.

The next group of intrinsics describe a process version of the algorithm described above which computes the homomorphisms one at a time. When searching for a quotient of the group having certain properties this technique can save a great deal of time.

HomomorphismsProcess(F, G, A : parameters): GrpFP, GrpPerm, GrpPerm -> GrpFPHomsProc

HomomorphismsProcess(F, G : parameters): GrpFP, GrpPerm -> GrpFPHomsProc

```
Surjective      : BoolElt                       Default: true
Limit           : RngIntElt                     Default: 0 (no limit)
TimeLimit       : RngIntElt                     Default: 0 (no limit)
CosetEnumeration: BoolElt                       Default: true
CacheCosetAction: BoolElt                       Default: true
```

Given a finitely presented group F and two permutation groups G and A with $G \lhd A$, return a process P for computing representatives of the classes of homomorphisms from F to G modulo automorphisms of G induced by elements of A. (That is, two homomorphisms f_1, $f_2 : F \to G$ are considered equivalent if there exists an element $a \in A$ such that $f_1(x) = f_2(x)^a$ for all $x \in F$.) `HomomorphismsProcess(F, G)` is equivalent to `HomomorphismsProcess(F, G, G)`. After constructing the process, the search for homomorphisms is started and runs until the first homomorphism is found, the time limit is reached (in which case P is marked as *invalid*) or the search is completed without finding a homomorphism (in which case P is marked as *empty*). The parameters have the same meaning as for the function *Homomorphisms*.

Setting a time limit for a process P limits the total amount of time spent in the backtrack search for homomorphisms during the construction of P and in subsequent calls to *NextElement*. The time spent in initial coset enumerations is *not* counted towards this limit. If the time limit or the limit on the number of homomorphisms set for a process P is reached, P becomes *invalid*. Calling *NextElement* for an invalid process or for a process which is *empty*, that is, which has found all possible classes of homomorphisms, will cause a runtime error. The functions *IsValid* and *IsEmpty* can be used to check whether a process is valid or empty, respectively. The use of these functions is recommended to avoid runtime errors in loops or user written functions.

`NextElement(~P): GrpFPHomsProc`

Given a valid and non-empty process P, continue the backtrack search until a new class of homomorphisms is found. If the search completes without a new representative being found, P is marked as empty. If a limit set for P is reached, P is marked as invalid.

`IsEmpty(P): GrpFPHomsProc -> BoolElt`

Returns whether P is empty, that is, whether all possible classes of homomorphisms have been found.

`IsValid(P): GrpFPHomsProc -> BoolElt`

Returns `false` if a limit set for P has been reached and `true` otherwise. Note that the return value of `IsValid` is not related to whether or not P currently defines a homomorphism.

`DefinesHomomorphism(P): GrpFPHomsProc -> BoolElt`

Returns whether P currently defines a homomorphism which can be extracted using the function *Homomorphism*.

`Homomorphism(P): GrpFPHomsProc -> HomGrp`

Given a process P which defines a homomorphism, return this homomorphism, that is, the homomorphism most recently found by P. If P does not define a homomorphism, a runtime error will result. The function *DefinesHomomorphism* can be used to test whether a call to `Homomorphism` is legal for a process.

`# P: GrpFPHomsProc -> RngIntElt`

Return the number of homomorphisms that have been found by the process P.

Example: Braid Group onto PSL(2, 16) (ex-9b82f6)

Consider the braid group B on 4 strings.

We show how the interactive computation of homomorphisms can be used to determine a homomorphism $f : B \to \mathrm{PSL}(2, 16)$ whose image is a maximal subgroup of $\mathrm{PSL}(2, 16)$. Note how the functions *IsValid*, *IsEmpty*, and *DefinesHomomorphism* are used to avoid runtime errors in the `while` loop.

```
> B := BraidFPGroup(4);
> G := PSL(2,16);
> P := HomomorphismsProcess(B, G : Surjective := false,
>                                  TimeLimit := 10);
> while not IsEmpty(P) do
>    if DefinesHomomorphism(P) then
>       f := Homomorphism(P);
>       img := Image(f);
>       if IsMaximal(G,img) then
>          print "found image which is maximal subgroup";
>          break;
>       end if;
>    end if;
>    if IsValid(P) then
>       NextElement(~P);
>     else
>       print "Limit has been reached";
>       break;
>    end if;
> end while;
found image which is maximal subgroup
>
> f;
Homomorphism of GrpFP: B into GrpPerm: G, induced by
  B.1 |--> (1, 4, 3, 6, 12)(2, 11, 9, 16, 7)(5, 17, 8, 15, 13)
  B.2 |--> (1, 4, 2, 10, 16)(3, 11, 9, 12, 14)(5, 15, 8, 13, 17)
  B.3 |--> (1, 4, 3, 6, 12)(2, 11, 9, 16, 7)(5, 17, 8, 15, 13)
```

Example: Proving Infiniteness Interactively (ex-3b7c60)

This example sketches how the interactive version of the homomorphism algorithm could be used as part of a function trying to prove that a group is infinite. Note again how the functions *IsValid*, *IsEmpty*, and *DefinesHomomorphism* are used to avoid runtime errors.

```
> function MyIsInfinite(F)
>
>  // ...
>
>  // quotient approach: check whether an obviously infinite
>  //    normal subgroup can be found in reasonable time.
>  S := [ Alt(5), PSL(2,7), PSL(2,9), PSL(2,11) ];
>  for G in S do
>     P := HomomorphismsProcess(F, G : Surjective := false,
>                                      TimeLimit := 5);
>     while IsValid(P) and not IsEmpty(P) do
>       if DefinesHomomorphism(P) then
>          f := Homomorphism(P);
>          if 0 in AQInvariants(Kernel(f)) then
>             print "found infinite normal subgroup";
>             print "Hence group is infinite";
>             return true;
>          end if;
>       end if;
>       if IsValid(P) then
>          NextElement(~P);
>       end if;
>    end while;
```

(continues on next page)

(continued from previous page)

```
>  end for;
>  print "quotient approach failed; trying other strategies";
>
>  // ...
>
> end function;
```

We try the code fragment on the group

$$< a, b \mid ab^{-1}a^{-1}ba^{-1}b^{-1}abb, ab^{-1}a^{-1}baaba^{-1}b^{-1}ab^{-1}a^{-1}baba^{-1}b^{-1} > .$$

```
> F := FPGroup< a,b |
>    a*b^-1*a^-1*b*a^-1*b^-1*a*b*b,
>    a*b^-1*a^-1*b*a*a*b*a^-1*b^-1*a*b^-1*a^-1*b*a*b*a^-1*b^-1 >;
> MyIsInfinite(F);
found infinite normal subgroup
Hence group is infinite
true
```

80.10.5 Searching for Isomorphisms

This section describes a function for searching for isomorphisms between two finitely presented groups.

SearchForIsomorphism(F, G, m : parameters): GrpFP, GrpFP, RngIntElt -> BoolElt, HomGrp, HomGrp

```
All        : BoolElt                    Default: false
IsomsOnly  : BoolElt                    Default: true
MaxRels    : RngIntElt                  Default: 250*m
CycConjTest: BoolElt                    Default: true
```

Attempt to find an isomorphism from the finitely presented group F to the finitely presented group G. The search will be restricted to those homomorphisms for which the sum of the word-lengths of the images of the generators of F in G is at most m.

If an isomorphism ϕ is found, then the values `true`, ϕ, ϕ^{-1} are returned. Otherwise, the values `false`, _, _ are returned; of course, that does not necessarily mean that the groups are not isomorphic.

An error will result if any of the generators of F turn out to be trivial.

By setting the verbose flag `"IsoSearch"` to 1, information about the progress of the search will be printed.

The parameters available for the function *SearchForIsomorphism* are:

If `All` is `false` (default), then the function halts and returns as soon as a single isomorphism is found. If `All` is `true`, then the search continues through all possible images that satisfy the image length condition, and a list of pairs $\langle \phi, \phi^{-1} \rangle$ for all isomorphisms $\phi : F \to G$ found is returned as the second return value.

If `IsomsOnly` is set to `false`, then all homomorphisms $F \to G$ will be returned if `All` is `true`, and the first nontrivial homomorphism found will be returned if `All` is `false`.

The value of the `MaxRelations` parameter used in runs of *RWSGroup*. It is hard to find a sensible default, because if the value is unnecessarily large then time can be wasted unnecessarily. This may need to be increased if the function is used with finite groups, for example (although it is usually much more efficient to use permutation or matrix representations when testing isomorphism of finite groups).

When `CycConjTest` is `true` and `All` is `false`, then images of the first generator which have a cyclic conjugate that comes earlier in the lexicographical order are rejected, because there would be a conjugate isomorphism in which the image was the cyclic conjugate. This nearly always results in faster run-times, but occasionally it can happen that the conjugate isomorphism has a larger sum of lengths of generator images, which is clearly bad. So the user has the option of not rejecting such images.

Example: Isomorphism Search for Hillman's Groups `(ex-d9442c)`

John Hillman asked whether the following two groups are isomorphic.

```
> G1<s,t,u> := FPGroup <s,t,u | s*u*s^-1=u^-1, t^2=u^2, t*s^2*t^-1=s^-2,
>                              u*(s*t)^2=(s*t)^2*u >;
> G2<x,y,z> := FPGroup<x,y,z | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=z^2*(x*y)^2,
>                             y^2=(z^-1*x)^2, z*(x*y)^2=(x*y)^2*z >;
> isiso, f1, f2 := SearchForIsomorphism(G1,G2,7);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1 into GrpFP: G2 induced by
    s |--> x * z^-1
    t |--> y * z
    u |--> x * y^-1 * z
> f2;
Homomorphism of GrpFP: G2 into GrpFP: G1 induced by
    x |--> s^2 * u * t^-1
    y |--> s^-1 * u^-1
    z |--> s * u * t^-1
```

The search for an isomorphism succeeded and returned the isomorphisms explicitly.

Example: Isomorphism Search for Neumann's Groups `(ex-4192a7)`

Walter Neumann asked whether the next two groups are isomorphic.

```
> G1<x,y,z> := FPGroup< x,y,z | x^2*y^5, x^14*z^23, (x^2,y), (x^2,z), x*y*z>;
> G2<a,b> := FPGroup<a,b | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7>;
```

It is a good idea to minimize the number of generators of `G1` - since there is clearly a redundant generator here.

```
> G1s := Simplify(G1);
> Ngens(G1s);
2
> G1!G1s.1, G1!G1s.2;
x z
```

Now a direct call `SearchForIsomorphism(G1, G2, 15)` will succeed, but will take many hours, and also use a lot of memory.

In contrast to `G1`, it can sometimes help to introduce a new generator in `G2` if there are common substrings in relators.

```
> G2b<a,b,c> := FPGroup< a,b,c | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7, c=b^7>;
> isiso, f1, f2 := SearchForIsomorphism(G1s,G2b,4);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1s into GrpFP: G2b induced by
    G1s.1 |--> a * c^-1
```

(continues on next page)

(continued from previous page)

```
    G1s.2 |--> c
> f2;
Homomorphism of GrpFP: G2b into GrpFP: G1s induced by
    a |--> G1s.1 * G1s.2
    b |--> G1s.1^6 * G1s.2^10
    c |--> G1s.2
```

80.11 Quotient Group Methods

A powerful technique for investigating a fp-group is to study some of its quotients. In this section we present tools for constructing various types of quotient group. Specifically, machinery is available for constructing abelian quotients, p-quotients, nilpotent quotients, soluble quotients, and $L(2)$, $L(3)$, and $U(3)$ quotients.

80.11.1 Abelian Quotient

The functions in this section compute information about the abelian quotient of an fp-group G. Some functions may require the computation of a coset table. Experienced users can control the behaviour of an implicit coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

The functions returning the abelian quotient or the abelian quotient invariants report an error if the abelian quotient cannot be computed, for example, because the relation matrix is too large. To avoid problems in user written programs or loops, the functions *HasComputableAbelianQuotient* and *HasInfiniteComputableAbelianQuotient* can be used.

AbelianQuotient(G): GrpFP -> GrpAb, Map

The maximal abelian quotient G/G' of the group G as GrpAb (cf. Chapter *Abelian Groups*). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

ElementaryAbelianQuotient(G, p): GrpFP, RngIntElt -> GrpAb, Map

The maximal p-elementary abelian quotient Q of the group G as GrpAb (cf. Chapter *Abelian Groups*). The natural epimorphism $\pi : G \to Q$ is returned as second value.

AbelianQuotientInvariants(G): GrpFP -> [RngIntElt]

AQInvariants(G): GrpFP -> [RngIntElt]

Given a finitely presented group G, this function computes the elementary divisors of the derived quotient group G/G', by constructing the relation matrix for G and transforming it into Smith normal form. The algorithm

the ring of integers $\mathbb{Z}$ using clever heuristics to minimize the growth of coefficients.

The divisors are returned as a sequence of integers.

AbelianQuotientInvariants(H): GrpFP -> [RngIntElt]

AQInvariants(H): GrpFP -> [RngIntElt]

AbelianQuotientInvariants(G, T): GrpFP, Map -> [RngIntElt]

AQInvariants(G, T): GrpFP, Map -> [RngIntElt]

Given a subgroup H of the finitely presented group G, this function computes the elementary divisors of the derived quotient group of H. (The coset table T may be used to define H.) This is done by abelianising the Reidemeister-Schreier presentation for H and then proceeding as above. The divisors are returned as a sequence of integers.

AbelianQuotientInvariants(G, n): GrpFP, RngIntElt -> [RngIntElt]

AQInvariants(G, n): GrpFP, RngIntElt -> [RngIntElt]

Given a finitely presented group G, this function computes the elementary divisors of the quotient group G/N, where N is the normal subgroup of G generated by the derived group G' together with all n-th powers of elements of G. The algorithm constructs the relation matrix corresponding to the presentation of G and computes its Smith normal form over the ring Z/nZ. The calculation is particularly efficient when n is a small prime. The divisors are returned as a sequence of integers.

AbelianQuotientInvariants(H, n): GrpFP, RngIntElt -> [RngIntElt]

AQInvariants(H, n): GrpFP, RngIntElt -> [RngIntElt]

AbelianQuotientInvariants(G, T, n): GrpFP, Map, RngIntElt -> [RngIntElt]

AQInvariants(G, T, n): GrpFP, Map, RngIntElt -> [RngIntElt]

Given a subgroup H of the finitely presented group G, this function computes the elementary divisors of the quotient group H/N, where N is the normal subgroup of H generated by H' together with all n-th powers of elements of H. (The coset table T may be used to define H.) This is done by abelianising the Reidemeister-Schreier presentation for H and then proceeding as above. The divisors are returned as a sequence of integers.

HasComputableAbelianQuotient(G): GrpFP -> BoolElt, GrpAb, Map

Given an fp-group G, this function tests whether the abelian quotient of G can be computed. If so, it returns the value `true`, the abelian quotient A of G, and the natural epimorphism $\pi : G \to A$. If the abelian quotient of G cannot be computed, the value `false` is returned.

This function is especially useful to avoid runtime errors in user written loops or functions.

HasInfiniteComputableAbelianQuotient(G): GrpFP -> BoolElt, GrpAb, Map

Given an fp-group G, this function tests whether the abelian quotient of G can be computed and is infinite. If so, it returns the value `true`, the abelian quotient A of G, and the natural epimorphism $\pi : G \to A$. If the abelian quotient of G cannot be computed or if it is finite, the value `false` is returned.

The function first checks the modular abelian invariants for a set of small primes. If for one of these primes, the modular abelian quotient is trivial, A must be finite and the function returns without actually computing the abelian quotient. If one is interested only in infinite quotients, this heuristics may save time.

IsPerfect(G): GrpFP -> BoolElt

Given an fp-group G, this function tries to decide whether G is perfect by checking whether the abelian quotient of G is trivial.

TorsionFreeRank(G): GrpFP -> RngIntElt

Given the finitely presented group G, return the torsion-free rank of the derived quotient group of G.

Example: Structure of the Fibonacci Group F(7) (ex-5b646e)

The Fibonacci group $F(7)$ has the following 2-generator presentation:

$$< a, b \mid a^2b^{-2}a^{-1}b^{-2}(a^{-1}b^{-1})^2, abab^2abab^{-1}(ab^2)^2 > .$$

We proceed to investigate the structure of this group.

```
> F<a, b> := FreeGroup(2);
> F7<a, b> := quo< F |  a^2*b^-2*a^-1*b^-2*(a^-1*b^-1)^2,
>        a*b*a*b^2*a*b*a*b^-1*(a*b^2)^2 >;
> F7;
Finitely presented group F7 on 2 generators
Relations
       a^2 * b^-2 * a^-1 * b^-2 * a^-1 * b^-1 * a^-1 * b^-1 =Id(F7)
       a * b * a * b^2 * a * b * a * b^-1 * a * b^2 * a * b^2 = Id()7
```

We begin by determining the structure of the maximal abelian quotient of $F(7)$.

```
> AbelianQuotientInvariants(F7);
[ 29 ]
```

The maximal abelian quotient of $F(7)$ is cyclic of order 29. At this point there is no obvious way to proceed, so we attempt to determine the index of some subgroups.

```
> Index( F7, sub< F7 | a > );
1
```

We are in luck: $F(7)$ is generated by a and so must be cyclic. This fact coupled with the knowledge that its abelian quotient has order 29 tells us that the group is cyclic of order 29.

Example: Abelian Quotient Invariants Modulo 2 (ex-818c7a)

The group $G = (8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 > .$$

We consider the subgroup H of G, generated by the words a^2 and $a^{-1}b$:

```
> G<a, b> := FPGroup<a, b| a^8, b^7, (a * b)^2, (a^-1 * b)^3>;
> H<x, y> := sub< G | a^2, a^-1 * b >;
```

The fastest way to determine the order of the maximal 2-elementary abelian quotient of H is to use the function *AbelianQuotientInvariants*:

```
> #AbelianQuotientInvariants(H,2);
1
```

We see that the maximal 2-elementary abelian quotient of H has order 2^1.

HasFiniteAbelianQuotient(G): GrpFP -> [RngIntElt]

HasFiniteAQ(G): GrpFP -> [RngIntElt]

Return `true` if G has finite abelian quotient. The algorithm requires either a presentation for G, or the coset table for G inside some supergroup of G which has a presentation. If such a coset table cannot be computed, an error will result.

AQPrimes(G): GrpFP -> [RngIntElt]

Return a sequence of primes containing all prime divisors of the order of G/G', if this is finite. Note that there may be primes in the return sequence which do not divide this order. If `[ 0 ]` is returned then G/G' is infinite. If `[]` is returned, then G is perfect. The algorithm requires either a presentation for G, or the coset table for G inside some supergroup of G which has a presentation. If such a coset table cannot be computed, an error will result.

80.11.2 p-Quotient

Let F be a finitely presented group, p a prime, and c a positive integer. A p-quotient algorithm constructs a consistent power-conjugate presentation for the largest p-quotient of F having lower exponent-p class at most c. The p-quotient algorithm used by Magma is part of the ANU p-Quotient program. For details of the algorithm, see [NO96]. In Magma the result is returned as a group of type `GrpPC` (cf. Chapter Finite Soluble Groups).

Assume that the p-quotient has order p^n, Frattini rank d, and that its generators are $a_1, \ldots, a_n$. Then the power-conjugate presentation constructed has the following additional structure. The set $\{a_1, \ldots, a_d\}$ is a generating set for G. For each a_k in $\{a_{d+1}, \ldots, a_n\}$, there is at least one relation whose right hand side is a_k. One of these relations is taken as the *definition* of a_k. (The list of definitions is also returned by `pQuotient`.) The power-conjugate generators also have a *weight* associated with them: a generator is assigned a weight corresponding to the stage at which it is added and this weight is extended to all normal words in a natural way.

The p-quotient function and its associated commands allows the user to construct a power-conjugate presentation (pcp) for a p-group. Note that there is also a process version of the p-quotient algorithm, which gives the user complete control over its execution. For a description, we refer to Subsec. *p-Quotient Process*.

pQuotient(F, p, c: parameters): GrpFP, RngIntElt, RngIntElt -> GrpPC, Map, SeqEnum, BoolElt

Given an fp-group F, a prime p, and a positive integer c, construct a pcp for the largest p-quotient G of F having lower exponent-p class at most c. If c is given as 0, then the limit 127 is placed on the class. The prime p must be less than 65536. The function also returns the natural homomorphism π from F to G, a sequence S describing the definitions of the pc-generators of G and a flag indicating whether G is the maximal p-quotient of F.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $G.k$ of G as follows.

- If $S[k] = [0, r]$, then $G.k$ is defined via the image of $F.r$ under π.
- If $S[k] = [r, 0]$, then $G.k$ is defined via the power relation for $G.r$.
- If $S[k] = [r, s]$, then $G.k$ is defined via the conjugate relation involving $G.r^{G.s}$.

There exist a number of parameters for controlling the behaviour of this function, which are described below.

Example: Largest 2-Quotient of Class 6 (ex-e7285c)

We construct the largest 2-quotient of class 6 for a two-generator, two-relator group.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | (b, a, a) = 1, (a * b * a)^4 = 1 >;
> Q, fQ := pQuotient(G, 2, 6);
> Order(Q);
524288
> fQ;
Mapping from: GrpFP: G to GrpPC: Q
```

The parameters available for the function *pQuotient* are:

```
Exponent: RngIntElt                    Default: 0
```

If `Exponent := m`, enforce the exponent law, $x^m = 1$, on the group.

```
Metabelian: BoolElt                    Default: false
```

If `Metabelian := true`, then a consistent pcp is constructed for the largest metabelian p-quotient of F having lower exponent-p class at most c.

```
Print: RngIntElt                    Default: 0
```

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("pQuotient")`, which is 0 unless it has been changed through use of `SetVerbose`. The effect is the following:

`Print := 0`: No output.

`Print := 1`: Report order of p-quotient at each class.

`Print := 2`: Report statistics and redundancy information about tails, consistency, collection of relations, and exponent enforcement components of calculation.

`Print := 3`: Report in detail on the construction of each class.

Note that the presentation displayed is a *power-commutator* presentation (since this is the version stored by the p-quotient).

```
Workspace: RngIntElt                    Default: 5000000
```

The amount of space requested for the p-quotient computation.

Example: Largest 3-Quotient of Class 6 (ex-090d87)

We construct the largest 3-quotient of class 6 for a two-generator group of exponent 9.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | a^3 = b^3 = 1 >;
> q := pQuotient(G, 3, 6: Print := 1, Exponent := 9);
Lower exponent-3 central series for G
Group: G to lower exponent-3 central class 1 has order 3^2
Group: G to lower exponent-3 central class 2 has order 3^3
Group: G to lower exponent-3 central class 3 has order 3^5
Group: G to lower exponent-3 central class 4 has order 3^7
Group: G to lower exponent-3 central class 5 has order 3^9
Group: G to lower exponent-3 central class 6 has order 3^11
```

Example: Metabelian 5-Quotient (ex-bccae5)

We use the metabelian parameter to construct a metabelian 5-quotient of the group

$$< a, b \mid a^{625} = b^{625} = 1, (b, a, b) = 1, (b, a, a, a, a) = (b, a)^5 > .$$

```
> F<a, b> := FreeGroup(2);
> G := quo< F | a^625 = b^625 = 1, (b, a, b) = 1,
>           (b, a, a, a, a) = (b, a)^5 >;
> q := pQuotient(G, 5, 20: Print := 1, Metabelian := true);
Lower exponent-5 central series for G
Group: G to lower exponent-5 central class 1 has order 5^2
Group: G to lower exponent-5 central class 2 has order 5^5
Group: G to lower exponent-5 central class 3 has order 5^8
Group: G to lower exponent-5 central class 4 has order 5^11
Group: G to lower exponent-5 central class 5 has order 5^12
Group: G to lower exponent-5 central class 6 has order 5^13
Group: G to lower exponent-5 central class 7 has order 5^14
Group: G to lower exponent-5 central class 8 has order 5^15
Group: G to lower exponent-5 central class 9 has order 5^16
Group: G to lower exponent-5 central class 10 has order 5^17
Group: G to lower exponent-5 central class 11 has order 5^18
Group: G to lower exponent-5 central class 12 has order 5^19
Group: G to lower exponent-5 central class 13 has order 5^20
Group completed. Lower exponent-5 central class = 13, order = 5^20
```

Example: Largest 2-Generator Group of Exponent 5 (ex-a5910e)

In the final example, we construct the largest finite 2-generator group having exponent 5.

```
> F := FreeGroup(2);
> q := pQuotient (F, 5, 14: Print := 1, Exponent := 5);
Lower exponent-5 central series for F
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14
```

(continues on next page)

(continued from previous page)

```
Group: F to lower exponent-5 central class 7 has order 5^18
Group: F to lower exponent-5 central class 8 has order 5^22
Group: F to lower exponent-5 central class 9 has order 5^28
Group: F to lower exponent-5 central class 10 has order 5^31
Group: F to lower exponent-5 central class 11 has order 5^33
Group: F to lower exponent-5 central class 12 has order 5^34
Group completed. Lower exponent-5 central class = 12, order = 5^34
```

`HaspQuotientDefinitions(G): GrpPC -> BoolElt`

Given a pc-group G, this function returns `true` iff and only if G has generator definitions set by `pQuotient`.

`pQuotientDefinitions(G): GrpPC -> SeqEnum`

Given a pc-group G, this function returns the generator definitions set by `pQuotient`, and will throw an error if none exist.

80.11.3 p-Quotient Process

In this and the next subsections the interative p-quotient process is described.

`pQuotientProcess(F, p, c: parameters): GrpFP, RngIntElt, RngIntElt -> Process`

Given an fp-group F, a prime p, and a positive integer c, create a p-quotient process for the group F with the indicated arguments. As part of the initialisation of the process, a pcp for the largest p-quotient of F having class at most c will be constructed. If c is given as 0, then the limit 127 is placed on the class. This function supports the same parameters as `pQuotient` and returns a process P.

`NextClass(~P : parameters): GrpPCpQuotientProc`

`NextClass(~P, k : parameters): GrpPCpQuotientProc, RngIntElt`

```
Exponent     : RngIntElt                    Default:
Metabelian   : BoolElt                      Default:
Print        : RngIntElt                    Default:
MaxOccurrence: [ RngIntElt ]                Default: []
```

Assumes that a pcp has already been constructed for the class c quotient of F. It seeks to construct a pcp for the class $c+1$ p-quotient of F. If k is supplied, continue to construct until the pcp for the largest quotient of class k is constructed.

The parameters `Exponent`, `Print`, and `Metabelian` are used as before. If `MaxOccurrence := Q`, then the sequence Q has length equal to the rank of the class 1 quotient of F; its entries are integers which specify the maximum number of occurrences of the class 1 generators in the definitions of pcp generators of F. An entry of 0 for a particular generator indicates that no limit is placed on the number of occurrences of this generator.

Care should be exercised when supplying values for parameters. Once set, they retain their values until explicitly reassigned.

Using p-Quotient Interactively

We assume that we have constructed a pcp for the largest class c p-quotient of F and now seek to construct a pcp for the largest class $c+1$ p-quotient. The following options allow the user to construct a pcp for the next class of the group interactively. The steps are laid out in one of a number of natural sequences in which they may be executed. Some of them may be interleaved; however, the user should pay particular attention to the assumptions mentioned below. The procedures that drive the process do not verify that the assumptions are satisfied.

`StartNewClass(~P: parameters): GrpPCpQuotientProc`

If P is a process for a class c p-quotient, commence construction of class $c+1$.

Tails(~P: parameters): GrpPCpQuotientProc

Tails(~P, k: parameters): GrpPCpQuotientProc, RngIntElt

```
Metabelian: BoolElt                    Default: false
```

Add tails to the current pcp; default is to add all tails for this class. If k is supplied, then tails for weight k only are added; in this case, it is assumed that the tails for each of weight $c+1, c, \ldots, k+1$ have already been added. The valid range of k is $2, \ldots, c+1$. The one valid parameter is Metabelian; if true, then only the tails for the metabelian p-quotient are inserted.

Consistency(~P: parameters): GrpPCpQuotientProc

Consistency(~P, k: parameters): GrpPCpQuotientProc

```
Metabelian: BoolElt                    Default: false
```

Apply the consistency algorithm to the pcp to compute any redundancies among the tails already added. Default is to apply it to all tails; in this case, it is assumed that all tails have been added. If k is supplied, it is assumed that tails for weight k have been added; in this case, the tails added for weight k only are checked. The range of k is $3, \ldots, c+1$. The one valid parameter is Metabelian; if true, we assume that the tails inserted were those for a metabelian p-quotient and hence invoke the (less expensive) metabelian consistency algorithm.

CollectRelations(~P): GrpPCpQuotientProc

Collect the defining relations (if any) in the current pcp. If the tails operation is not complete, then the relations may be evaluated incorrectly.

ExponentLaw(~P : parameters): GrpPCpQuotientProc

ExponentLaw(~P, Start, Fin: parameters): GrpPCpQuotientProc, RngIntElt, RngIntElt

```
Exponent       : RngIntElt                          Default: 0
Print          : RngIntElt                          Default: 1
Trial          : BoolElt                            Default: false
ShortList      : BoolElt                            Default: false
DisplayList    : BoolElt                            Default: false
IdentifyFilters: BoolElt                            Default: false
InitialSegment : [<GrpFPElt, RngIntElt>]            Default: []
Report         : RngIntElt                          Default: 0
```

Enforce the supplied exponent law on the current pcp. If Start and Fin are supplied, then enforce the law for those weights between Start and Fin; otherwise, enforce the law for all weights. It is assumed that the tails operation is complete. If the display parameter DisplayLevel (which may be set using SetDisplayLevel) has value 2, those words whose powers are collected and give redundancies among the pcp generators are printed out. If DisplayLevel has value 3, all words whose powers are collected are printed out. The following additional parameters are available:

If Exponent := m, enforce the exponent law, $x^m = 1$, on the group.

As for pQuotient.

Generate the list of words used to enforce the exponent law and print out statistics but do not power words or echelonise the results.

Generate the list of enforcement words whose entries have the form w or $1 * w$ where w is an element of the Frattini subgroup of F.

Display the list of all enforcement words generated – not just those which are collected.

Identify filters used to eliminate words from list.

If InitialSegment := w, generate only those enforcement words which have w as an initial segment, where w is supplied as a sequence of generator-exponent pairs.

If Report := n, report after computing the powers of each collection of n enforcement words.

`EliminateRedundancy(~P): GrpPCpQuotientProc`

Eliminate all redundant generators from the pcp defined by process P. This operation may be performed at any time.

We now list the remaining functions which can be applied to a pQuotient process.

`Display(P): GrpPCpQuotientProc`

`Display(P, DisplayLevel): GrpPCpQuotientProc, RngIntElt`

Display the pcp for the p-quotient G of the fp-group F. The argument `DisplayLevel` may be 1, 2, or 3, and is used to control the amount of information given:

`1 :` Display order and class of G;

`2 :` Display non-trivial relations for G;

`3 :` Display the structure of pcp generators of G, non-trivial relations of G, and the map from the defining generators of F to the pcp generators of G.

The presentation displayed by this function is in power-commutator form. If `DisplayLevel` is not supplied, the information displayed is determined by its existing (or default) value.

`RevertClass(~P): GrpPCpQuotientProc`

Given a pcp for the class $c+1$ p-quotient of F, this procedure reverts to the pcp for the class c p-quotient of F. Note that this command can be applied only **once** during construction of a single class.

`pCoveringGroup(~P): GrpPCpQuotientProc`

`pCoveringGroup(G): GrpPC -> GrpPC`

Given a process or a pcp for a p-group, this procedure computes a pcp for the p-covering group of this group. In the process case, it is equivalent to `Tails(~P); Consistency(~P); EliminateRedundancy(~P)`.

`GeneratorStructure(P): GrpPCpQuotientProc`

`GeneratorStructure(P, Start, Fin): GrpPCpQuotientProc, RngIntElt, RngIntElt`

Display the structure of the generators in the pcp. If *Start* and *Fin* are given, then print out the structure of those pcp generators numbered from *Start* to *Fin*.

```
Jacobi(~P, c, b, a, ~r): GrpPCpQuotientProc, RngIntElt, RngIntElt, RngIntElt ->
      RngIntElt ->
```

```
Jacobi(~P, c, b, a): GrpPCpQuotientProc, RngIntElt, RngIntElt, RngIntElt -> RngIntElt
      ->
```

Calculate the Jacobi c, b, a and echelonise the resulting relation against the current pcp. If a redundant generator results from the echelonisation, the optional variable r is the number of that generator; otherwise r has value 0.

`Collect(P, Q): GrpPCpQuotientProc, [ <RngIntElt, RngIntElt> ] -> [ RngIntElt ] ->`

The sequence Q, consisting of generator-exponent pairs, defines a word w in the pcp generators of the group defined by the process P. Collect this word and return the resulting normal word as an exponent vector.

`EcheloniseWord(~P, ~r): GrpPCpQuotientProc -> RngIntElt`

`EcheloniseWord(~P): GrpPCpQuotientProc -> RngIntElt`

Echelonise the word most recently collected using `Collect` against the relations of the pcp. If a redundant generator results from the echelonisation, the optional variable r is the number of that generator; otherwise r has value 0. This function must be called immediately after `Collect`.

`SetDisplayLevel(~P, Level): GrpPCpQuotientProc, RngIntElt`

This procedure alters the display level for the process to the supplied value, `Level`.

ExtractGroup(P): GrpPCpQuotientProc -> GrpPC

Extract the group G defined by the pcp associated with the process P, as a member of the category GrpPC of finite soluble groups. The function also returns the natural homomorphism π from the original group F to G, a sequence S describing the definitions of the pc-generators of G and a flag indicating whether G is the maximal p-quotient of F.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $G.k$ of G as follows.

- If $S[k] = [0, r]$, then $G.k$ is defined via the image of $F.r$ under π.
- If $S[k] = [r, 0]$, then $G.k$ is defined via the power relation for $G.r$.
- If $S[k] = [r, s]$, then $G.k$ is defined via the conjugate relation involving $G.r^{G.s}$.

Order(P): GrpPCpQuotientProc -> RngIntElt

The order of the group defined by the pcp associated with the process P.

FactoredOrder(P): GrpPCpQuotientProc -> [<RngIntElt, RngIntElt>]

The factored order of the group defined by the pcp associated with the process P.

NumberOfPCGenerators(P): GrpPCpQuotientProc -> RngIntElt

The number of pc-generators of the group defined by the pcp associated with the process P.

pClass(P): GrpPCpQuotientProc -> RngIntElt

The lower exponent-p class of the group defined by the pcp associated with the process P.

NuclearRank(G): GrpPC -> RngIntElt

NuclearRank(P): GrpPCpQuotientProc -> RngIntElt

Return the rank of the p-multiplicator of the p-group G, where G may be supplied or defined by the process P.

pMultiplicatorRank(G): GrpPC -> RngIntElt

pMultiplicatorRank(P): GrpPCpQuotientProc -> RngIntElt

Return the rank of the p-multiplicator of the p-group G, where G may be supplied or defined by the process P.

Example: Extending a 3-Quotient Interactively (ex-ea9994)

Starting with an exponent 9 group having two generators of order 3, we set up the largest class 4 quotient and then interactively compute two more classes.

```
> G<a, b> := FPGroup<a, b | a^3, b^3>;
> q := pQuotientProcess(G, 3, 4: Exponent := 9, Print :=1);

Lower exponent-3 central series for G

Group: G to lower exponent-3 central class 1 has order 3^2

Group: G to lower exponent-3 central class 2 has order 3^3

Group: G to lower exponent-3 central class 3 has order 3^5

Group: G to lower exponent-3 central class 4 has order 3^7
> Display(q, 2);

Group: G to lower exponent-3 central class 4 has order 3^7

Non-trivial powers:
 .3^3 = .6^2
```

(continues on next page)

(continued from previous page)

```
Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
[ .5, .1 ] = .7
[ .5, .2 ] = .6
```

We construct the class 5 quotient using the function *NextClass*.

```
> NextClass(~q);

Group: G to lower exponent-3 central class 5 has order 3^9
```

We now construct the class 6 quotient step by step. For this, we set the output level to 1.

```
> SetDisplayLevel(~q, 1);
```

Now we start the next class.

```
> StartNewClass(~q);
```

The first step is to add the tails.

```
> Tails(~q);
```

After that, we apply the consistency algorithm,...

```
> Consistency(~q);
```

...collect the defining relations,...

```
> CollectRelations(~q);
```

...and enforce the exponent law.

```
> ExponentLaw(~q);
```

Finally, we eliminate redundant generators.

```
> EliminateRedundancy(~q);
```

This results in the following presentation for class 6 quotient.

```
> Display(q, 2);

Group: G to lower exponent-3 central class 6 has order 3^11

Non-trivial powers:
 .3^3 = .6^2 .8^2 .10 .11

Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
```

(continues on next page)

(continued from previous page)

```
[ .4, .3 ] = .8 .10^2 .11^2
[ .5, .1 ] = .7 .8 .9^2 .10^2 .11^2
[ .5, .2 ] = .6 .8 .9^2 .10^2
[ .5, .3 ] = .9^2 .11
[ .5, .4 ] = .10 .11
[ .6, .2 ] = .10^2
[ .7, .1 ] = .8
[ .7, .2 ] = .9
[ .7, .3 ] = .10^2 .11
[ .8, .2 ] = .10
[ .9, .1 ] = .11
```

Example: Bounding Definitions of Pcp Generators (ex-0cb2c6)

Starting with the free product of two cyclic groups of order 5, we bound the number of occurrences of pcp generators of the class 1 quotient in definitions of new pcp generators.

We start with setting up the class 1 quotient of the group.

```
> G := FPGroup<a, b | a^5, b^5>;
> q := pQuotientProcess(G, 5, 1);
> Display(q, 1);

Group: G to lower exponent-5 central class 1 has order 5^2
```

Now we start the next class, setting bounds on the number of occurrences of the pcp generators of the class 1 quotient in the definitions of new pcp generators.

```
> NextClass(~q, 6: MaxOccurrence := [3, 2]);
Group: G to lower exponent-5 central class 2 has order 5^3
Group: G to lower exponent-5 central class 3 has order 5^5
Group: G to lower exponent-5 central class 4 has order 5^7
Group: G to lower exponent-5 central class 5 has order 5^9

> Display(q, 2);

Group: G to lower exponent-5 central class 5 has order 5^9

Non-trivial powers:

Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
[ .4, .3 ] = .8^4 .9
[ .5, .1 ] = .7 .8^4 .9
[ .6, .2 ] = .8
[ .7, .1 ] = .9
```

Example: Class 7 Quotient of R(2, 5) (ex-371e40)

We construct the class 6 quotient q of $R(2,5)$ and then partially construct the class 7 quotient interactively to find out how many normal words having initial segment $q.1^2$ need to be considered when imposing the exponent law.

```
> F := FreeGroup(2);
> q := pQuotientProcess(F, 5, 6: Exponent := 5);

Lower exponent-5 central series for F
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14

> StartNewClass(~q);
> Tails(~q);
> Consistency(~q);
> SetDisplayLevel(~q, 3);
> ExponentLaw(~q, 1, 6: InitialSegment := [<1, 2>], Trial := true);
0 Relations of class 1 will be collected
0 Relations of class 2 will be collected
Will collect power 5 of the following word: 1^2 2^1
1 Relation of class 3 will be collected
Will collect power 5 of the following word: 1^2 2^2
1 Relation of class 4 will be collected
Will collect power 5 of the following word: 1^2 2^3
Will collect power 5 of the following word: 1^2 5^1
2 Relations of class 5 will be collected
Will collect power 5 of the following word: 1^2 2^4
Will collect power 5 of the following word: 1^2 2^1 4^1
2 Relations of class 6 will be collected
```

Example: Interactively Extending a Class 6 Quotient (ex-0ff68f)

We demonstrate several of the remaining procedures in the course of interactively extending the class 6 quotient of the group

$$< a, b, c, d \mid (bc^{-1}d)^7, (cd^{-1})^7, (b, a) = c^{-1}, (c, a) = 1, (c, b) = d^{-1} >$$

to class 7.

```
> G := FPGroup<a, b, c, d | (b * c^-1 * d)^7, (c * d^-1)^7, (b,a) = c^-1,
>                           (c,a) = 1, (c,b) = d^-1>;
> q := pQuotientProcess(G, 7, 6);

Lower exponent-7 central series for G
Group: G to lower exponent-7 central class 1 has order 7^2
Group: G to lower exponent-7 central class 2 has order 7^4
Group: G to lower exponent-7 central class 3 has order 7^6
Group: G to lower exponent-7 central class 4 has order 7^8
Group: G to lower exponent-7 central class 5 has order 7^11
Group: G to lower exponent-7 central class 6 has order 7^14

> StartNewClass(~q);
> Tails(~q);
> GeneratorStructure(q, 15, 34);

Class 7
15 is defined on [12, 1] = 2 1 2 2 1 2 1
16 is defined on [12, 2] = 2 1 2 2 1 2 2
17 is defined on [13, 1] = 2 1 2 2 2 2 1
```

(continues on next page)

(continued from previous page)

```
18 is defined on [13, 2] = 2 1 2 2 2 2 2
19 is defined on [14, 1] = 1 1 1 1 1 1 1
20 is defined on [14, 2] = 1 1 1 1 1 1 2
21 is defined on 14^7 = 1 1 1 1 1 1
22 is defined on [9, 1] = 2 1 2 2 1 1
23 is defined on [10, 1] = 2 1 2 2 2 1
24 is defined on [11, 1] = 1 1 1 1 1 1
25 is defined on [11, 2] = 1 1 1 1 1 2
26 is defined on [8, 1] = 1 1 1 1 1
27 is defined on [8, 2] = 1 1 1 1 2
28 is defined on [5, 1] = 2 1 2 1
29 is defined on [6, 1] = 1 1 1 1
30 is defined on [6, 2] = 1 1 1 2
31 is defined on [3, 1] = 2 1 1
32 is defined on [4, 1] = 1 1 1
33 is defined on [4, 2] = 1 1 2
34 is defined on 2^7 = 2 2

> Jacobi(~q, 6, 6, 1);
Generator 26 is trivial
Jacobi was 6 6 1
> Jacobi(~q, 3, 2, 1);
Generator 28 is redundant
Jacobi was 3 2 1
> v := Collect(q,  [<29, 2>, <26, -3>]);
> v;
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 4, 0, 0, 2, 0, 0, 0, 0, 0 ]
> EcheloniseWord(~q, ~redgen);
Generator 29 is trivial
> Display(q, 1);

Group: G to lower exponent-7 central class 7 has order 7^34
```

Now we enforce the relations…

```
> CollectRelations(~q);
```

…and apply the consistency algorithm.

```
> Consistency(~q);
> Display(q, 1);

Group: G to lower exponent-7 central class 7 has order 7^36

> EliminateRedundancy(~q);
> Display(q, 1);
Group: G to lower exponent-7 central class 7 has order 7^19
```

80.11.4 Nilpotent Quotient

A nilpotent quotient algorithm constructs, from a finite presentation of a group, a polycyclic presentation of a nilpotent quotient of the finitely presented group. The nilpotent quotient algorithm used by Magma is the Australian National University Nilpotent Quotient program, as described in [Nic96]. The version included in Magma is Version 2.2 of January 2007.

The lower central series $G_0, G_1, \ldots$ of a group G can be defined inductively as $G_0 = G$, $G_i = [G_(i-1), G]$.

G is said to have nilpotency class c if c is the smallest non-zero integer such that $G_c = 1$. If N is a normal subgroup of G and G/N is nilpotent, then N contains G_i for some non-negative integer i. G has infinite nilpotent quotients if and only if G/G_1 (the maximal abelian quotient of G) is infinite and a prime p divides a finite factor of a nilpotent quotient if and only if p divides a cyclic factor of G/G_1. The i-th ($i > 1$) factor G_{i-1}/G_i of the lower central series is generated by the elements $[g, h]G_i$, where g runs through a set of representatives of G/G_1 and h runs through a set of representatives of G_{i-2}/G_{i-1}.

Any finitely generated nilpotent group is polycyclic and, therefore, has a subnormal series with cyclic factors. Such a subnormal series can be used to represent the group in terms of a polycyclic presentation. The ANU NQ computes successively the factor groups modulo the terms of the lower central series. Each factor group is represented by a special form of polycyclic presentation, a nilpotent presentation, that makes use of the nilpotent structure of the factor group.

The algorithm has highly efficient code for enforcing the n-Engel identity in nilpotent groups. When appropriate parameters are set, the algorithm computes the largest n-Engel quotient of G/G_c.

More generally, the algorithm has code for enforcing arbitrary identical relations. You can even enforce relations which combine generators of G with "free variables". (Werner Nickel calls these "identical generators".) For instance, the relation `(a,x) = 1`, where a is a group generator and x is a free variable, will force the construction of quotients where the image of a is central.

Mike Vaughan-Lee offers advice on when to use the nilpotent quotient algorithm. If you know nothing about the finitely presented group, it is probably a good idea to look at the abelian quotient first. If the abelian quotient is trivial then all nilpotent quotients will be trivial. Similarly, if the abelian quotient is cyclic, then all nilpotent quotients will be cyclic. Less trivially, if the abelian quotient is finite then all nilpotent quotients will be finite and so will be the direct product of finite p-groups. Moreover, the relevant primes p will all occur in the abelian quotient. Usually it will be more effective to use the p-quotient algorithm to study the direct factors. However, if you want to study 3-Engel quotients (say), then you are better using the nilpotent quotient even when the abelian quotient is finite.

`NilpotentQuotient(G, c: parameters): GrpFP, RngIntElt -> GrpGPC, Map`

This function returns the class c nilpotent quotient of G as a group in category `GrpGPC`, together with the epimorphism π from G onto this quotient. When c is set to zero, the function attempts to compute the maximal nilpotent quotient of G. Using the parameters described below, the user can enforce certain conditions on the quotient found.

Example: Class 2 Nilpotent Quotient (ex-7067cc)

Here is a finitely presented group. The abelian quotient is infinite, so we look at the class 2 nilpotent quotient.

```
> G := FPGroup<x,y,z|(x*y*z^-1)^2, (x^-1*y^2*z)^2, (x*y^-2*x^-1)^2 >;
> AbelianQuotient(G);
Abelian Group isomorphic to Z/2 + Z/2 + Z
Defined on 3 generators
Relations:
  2*$.1 = 0
  2*$.2 = 0
> N := NilpotentQuotient(G,2); N;
GrpGPC : N of infinite order on 6 PC-generators
PC-Relations:
  N.1^2 = N.3^2 * N.5,
  N.2^2 = N.4 * N.6,
  N.4^2 = Id(N),
  N.5^2 = Id(N),
  N.6^2 = Id(N),
  N.2^N.1 = N.2 * N.4,
  N.3^N.1 = N.3 * N.5,
  N.3^N.2 = N.3 * N.6
```

Example: Free Nilpotent Groups (ex-8a4940)

The free nilpotent group of rank r and class e is defined as $F/\gamma_{e+1}(F)$, where F is a free group of rank r and $\gamma_{e+1}(F)$ denotes the $(e+1)$ st term of the lower central series of F.

We construct the free nilpotent group N of rank 2 and class 3 as quotient of the free group F of rank 2 and the natural epimorphism from F onto N.

```
> F<a,b> := FreeGroup(2);
> N<[x]>, pi := NilpotentQuotient(F, 3);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
    x[2]^x[1] = x[2] * x[3],
    x[2]^(x[1]^-1) = x[2] * x[3]^-1 * x[4],
    x[3]^x[1] = x[3] * x[4],
    x[3]^(x[1]^-1) = x[3] * x[4]^-1,
    x[3]^x[2] = x[3] * x[5],
    x[3]^(x[2]^-1) = x[3] * x[5]^-1
```

Using the function *NilpotencyClass* described in Chapter *Polycyclic Groups*, we check the nilpotency class of the quotient.

```
> NilpotencyClass(N);
3
```

Example: Nilpotent Quotients of Baumslag-Solitar Groups (ex-b77879)

The Baumslag-Solitar groups

$$BS(p,q) = \langle < a, b | ab^p a^{-1} = b^q \rangle$$

form a fascinating class of 1-relator groups. We compute the nilpotent of class 4 quotient of two of them, and print out the resulting presentations, and the structure of the generators.

```
> G<a,b> := FPGroup<a,b|a*b*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
  N.2^3 = N.3^2 * N.4^2 * N.5,
  N.3^3 = N.4^2 * N.5^2,
  N.4^3 = N.5^2,
  N.5^3 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3^2 * N.4^2 * N.5,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4^2 * N.5^2,
  N.4^N.1 = N.4 * N.5,
  N.4^(N.1^-1) = N.4 * N.5^2
> for i := 1 to Ngens(N) do
> N.i @@ f;
> end for;
a
b
(b, a)
(b, a, a)
(b, a, a, a)
> G<a,b> := FPGroup<a,b|a*b^2*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
```

(continues on next page)

(continued from previous page)

```
GrpGPC : N of infinite order on 8 PC-generators
PC-Relations:
  N.2^2 = Id(N),
  N.3^2 = N.5 * N.8,
  N.4^2 = N.7,
  N.5^2 = N.8,
  N.6^2 = Id(N),
  N.7^2 = Id(N),
  N.8^2 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3 * N.4 * N.5 * N.6,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4 * N.6 * N.7,
  N.3^N.2 = N.3 * N.5,
  N.4^N.1 = N.4 * N.6,
  N.4^(N.1^-1) = N.4 * N.6,
  N.4^N.2 = N.4 * N.7,
  N.5^N.1 = N.5 * N.7,
  N.5^(N.1^-1) = N.5 * N.7,
  N.5^N.2 = N.5 * N.8
> for i := 1 to Ngens(N) do
> N.i @@ f;
> end for;
a
b
(b, a)
(b, a, a)
(b, a, b)
(b, a, a, a)
(b, a, b, a)
(b, a, b, b)
```

The parameters available for the function *NilpotentQuotient* are:

```
NumberOfEngelGenerators: RngIntElt                    Default: 1
```

Setting this parameter to k forces the first k generators to be left or right Engel elements, provided one (or both) of the parameters `LeftEngel` or `RightEngel` is positive. Otherwise it is ignored.

```
LeftEngel: RngIntElt                    Default: 0
```

Setting this parameter to n forces the first k generators $g_1, \ldots, g_k$ of the quotient Q to be left n-Engel elements. That is, they satisfy $[x, \ldots, x, g_i] = 1$ (x appearing n times) for all x in Q. The value of k is determined by the parameter `NumberOfEngelGenerators`.

```
RightEngel: RngIntElt                    Default: 0
```

This is the same as for `LeftEngel`, but here the generators are right n-Engel elements, so $[g_i, x, \ldots, x] = 1$.

```
Engel: RngIntElt                    Default: 0
```

Setting this parameter to n enforces the nth Engel law on the quotient Q. That is, $[x, y, \ldots, y] = 1$ (y appearing n times) for all x, y in Q.

```
SemigroupOnly: BoolElt                    Default: true
```

This option causes the program to check only semigroup words in the generating set of the nilpotent quotient when an Engel condition is enforced. If none of the Engel parameters are set, then it is ignored.

```
SemigroupFirst: BoolElt                    Default: false
```

This option causes the program to check semigroup words in the generating set of the quotient first and then all other words, when an Engel condition is enforced. If `SemigroupOnly` is set, or no Engel condition is enforced, then it is ignored.

```
ReverseOrder: BoolElt                      Default: false
```

In checking Engel identities, instances are processed in order of increasing weight. This flag reverses the order.

```
ReverseEngel: BoolElt                      Default: false
```

This flag changes the Engel conditions from the first k generators, to the *last* k generators.

```
CheckFewInstances: BoolElt                 Default: false
```

This option stops checking the Engel law at each class if all the checks of a certain weight did not yield any non-trivial instances of the law.

```
Nickel: BoolElt                    Default: false
```

Enforce the identities x^8 and $[[x_1, x_2, x_3], [x_4, x_5, x_6]]$ on the nilpotent quotient.

```
NumberOfFreeVariables: RngIntElt           Default: 0
```

If this parameter is set to $n > 0$ then the last n variables of the group are treated as variables rather than generators of the group. In any relation they are treated as standing for all group elements, enabling the user to enforce identical relations on the quotient being computed. The value of n must be less than the number of generators of the input group. When this facility is used, the domain of the epimorphism returned is the subgroup of G generated by the (first) non-free generators.

```
PrintResult: BoolElt                       Default: false
```

If set to true, this parameter switches on the printing of results given by the stand alone version of NQ.

`SetVerbose("NilpotentQuotient", n): MonStgElt, RngIntElt`

Turn on and off the printing of information as the nilpotent quotient algorithm proceeds. n may be set to any of 0, 1, 2, or 3. Setting n to be 0 turns printing off, while successively higher values for n print more and more information as the algorithm progresses. The default value of n is 0.

Example: Maximal Nilpotent Quotient `(ex-85800d)`

We compute the maximal nilpotent quotient $N1$ of the group G given by the following presentation

$$\langle a, b, c, d, e \mid (b, a),\ (c, a),\ (d, a) = (c, b),\ (e, a) = (d, b), (e, b) = (d, c),\ (e, c),\ (e, d)\rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> :=
>          quo<F | (b,a), (c,a),
>          (d,a)=(c,b), (e,a)=(d,b), (e,b)=(d,c),
>          (e,c), (e,d)>;
> N1<[x]>, pi1 := NilpotentQuotient(G, 0);
```

Using the function *`NilpotencyClass`* described in Chapter *Polycyclic Groups*, we check the nilpotency class of the quotient. It turns out to have nilpotency class 6.

```
> NilpotencyClass(N1);
6
```

Next we compute a metabelian quotient $N2$ of G, construct the natural epimorphism from $N1$ onto $N2$ and check its kernel. (The functions applied to the nilpotent quotients are described in Chapter *Polycyclic Groups*.) To get a metabelian quotient we adjoin 4 variables and a relation to the presentation of G and use the "free variables" facility.

```
> M := FPGroup<w,x,y,z|((w,x),(y,z))>; // metabelian identity
> D := FreeProduct(G,M); // adjoin to G
> N2, pi2 := NilpotentQuotient(D, 0: NumberOfFreeVariables := 4);
> NilpotencyClass(N2);
4
> DerivedLength(N2);
2
> f := hom< N1->N2 | [ pi1(G.i)->pi2(D.i) : i in [1..Ngens(G)] ] >;
> PCGenerators(Kernel(f), N1);
{@ x[14], x[15] * x[17], x[16], x[17]^2, x[18], x[19], x[20],
x[21], x[22], x[23], x[24], x[25], x[26], x[27], x[28], x[29],
x[30], x[31] @}
```

We compute a quotient $N3$ of G satisfying the 4 th Engel law, construct the natural epimorphism from $N1$ onto $N3$ and again check the kernel. (The functions applied to the nilpotent quotients are described in Chapter *Polycyclic Groups*.)

```
> N3, pi3 := NilpotentQuotient(G, 0 : Engel := 4);
> NilpotencyClass(N3);
4
> DerivedLength(N3);
3
> h := hom< N1->N3 | [ pi1(g)->pi3(g) : g in Generators(G) ] >;
> PCGenerators(Kernel(h), N1);
{@ x[19], x[20], x[21], x[22], x[23], x[24], x[25], x[26], x[27],
x[28], x[29], x[30], x[31] @}
```

80.11.5 Soluble Quotient

A soluble quotient algorithm computes a consistent power-conjugate presentation of the largest finite soluble quotient of a finitely presented group, subject to certain algorithmic and user supplied restrictions. In this section we describe only the simplest use of such an algorithm within Magma. For more information the user is referred to Subsec. *Soluble Quotient Advanced*.

SolvableQuotient(G : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt
SolubleQuotient(G : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt

Let G be a finitely presented group. The function constructs the largest finite soluble quotient of G.

Example: Soluble Quotient of a Two-Generator Group (ex-a7ae66)

We compute the soluble quotient of the group

$$\langle a, b \mid a^2, b^4, ab^{-1}ab(abab^{-1})^5ab^2ab^{-2}\rangle.$$

```
> G<a,b> := FPGroup< a, b | a^2, b^4,
>                     a*b^-1*a*b*(a*b*a*b^-1)^5*a*b^2*a*b^-2 >;
> Q := SolubleQuotient(G);
> Q;
GrpPC : Q of order 1920 = 2^7 * 3 * 5
PC-Relations:
    Q.1^2 = Q.4,
    Q.2^2 = Id(Q),
```

(continues on next page)

(continued from previous page)

```
    Q.3^2 = Q.6,
    Q.4^2 = Id(Q),
    Q.5^2 = Q.7,
    Q.6^2 = Id(Q),
    Q.7^2 = Id(Q),
    Q.8^3 = Id(Q),
    Q.9^5 = Id(Q),
    Q.2^Q.1 = Q.2 * Q.3,
    Q.3^Q.1 = Q.3 * Q.5,
    Q.3^Q.2 = Q.3 * Q.6,
    Q.4^Q.2 = Q.4 * Q.5 * Q.6 * Q.7,
    Q.4^Q.3 = Q.4 * Q.6 * Q.7,
    Q.5^Q.1 = Q.5 * Q.6,
    Q.5^Q.2 = Q.5 * Q.7,
    Q.5^Q.4 = Q.5 * Q.7,
    Q.6^Q.1 = Q.6 * Q.7,
    Q.8^Q.1 = Q.8^2,
    Q.8^Q.2 = Q.8^2,
    Q.9^Q.1 = Q.9^3,
    Q.9^Q.2 = Q.9^4,
    Q.9^Q.4 = Q.9^4
```

SolvableQuotient(F, n : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt

SolubleQuotient(F, n : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt

SolvableQuotient(F, P : parameters): GrpFP, Set -> GrpPC, Map, SeqEnum, MonStgElt

SolubleQuotient(F, P : parameters): GrpFP, Set -> GrpPC, Map, SeqEnum, MonStgElt

Find a soluble quotient G and the epimorphism $\pi : F \twoheadrightarrow G$ with a specified order. n must be a nonnegative integer. P must be a set of primes.

The three forms reflect possible information about the order of an expected soluble quotient. In the first form the order of G is given by n, if n is greater than zero. If n equals zero, nothing about the order is known and the relevant primes will be calculated completely.

The second form, with no n argument, is equivalent to the first with $n = 0$. This is a standard argument, and usually it is the most efficient way to calculate soluble quotients.

Note that, if $n > 0$ is not the order of the maximal finite soluble quotient, it may happen that no group of order n can be found, since an epimorphic image of size n may not be exhibited by the chosen series.

In the third form a set P of relevant primes is given. The algorithm calculates the biggest quotient such that the order has prime divisors only in P. P may have a zero as element, this is just for consistency reasons. It is equivalent to the first form with n equal zero.

For a description of the algorithm used and of the set of parameters available for this function, see Subsec. *Soluble Quotient Advanced*. Other more specialised functions for computing soluble quotients and some examples can be found there as well.

Example: Order of a Soluble Quotient (ex-130896)

Consider the group G defined by the presentation

$$\langle x, y \mid x^3, y^8, [x, y^4], x^{-1}yx^{-1}y^{-1}xyxy^{-1}, (xy^{-2})^2(x^{-1}y^{-2})^2(xy^2)^2(x^{-1}y^2)^2, (x^{-1}y^{-2})^6(x^{-1}y^2)^6\rangle.$$

```
> G<x, y> := FPGroup< x, y |
>     x^3, y^8, (x,y^4), x^-1*y*x^-1*y^-1*x*y*x*y^-1,
>     (x*y^-2)^2*(x^-1*y^-2)^2*(x*y^2)^2*(x^-1*y^2)^2,
>     (x^-1*y^-2)^6*(x^-1*y^2)^6 >;
```

We apply the soluble quotient algorithm to G and compute the order of the soluble quotient Q.

```
> time Q := SolubleQuotient(G);
Time: 116.920
> Order(Q);
165888
```

Note that $165888 = 2^{11} \cdot 3^4$. If we knew the possible primes in advance the soluble quotient can be computed much more quickly by using this knowledge.

```
> time Q := SolubleQuotient(G, {2, 3});
Time: 39.400
```

Note that this reduces the execution time by a factor of almost three.

We now assume that G is finite and try to compute its order by means of the Todd-Coxeter algorithm.

```
> Order(G);
165888
```

Hence the group is finite and soluble and G is isomorphic to Q. We may use the tools for finite soluble groups to investigate the structure of G. For example we can easily find the number of involutions in G.

```
> cls := ConjugacyClasses(Q);
> &+ [ cl[2] : cl in cls | cl[1] eq 2 ];
511
```

80.11.6 Soluble Quotient Advanced

Introduction

This section presents a short overview towards the theory of the soluble quotient algorithm, the functions designed for computing soluble quotients and the functions designed for dealing with soluble quotient processes.

Construction

For a finite group G, the property of being soluble means that the derived series $G = G^{(0)} > G^{(1)} > ... > G^{(n)}$ terminates with $G^{(n)} = \langle 1 \rangle$. Each section $G^{(i)}/G^{(i+1)}$ is a finite abelian group, hence it can be identified with a $\mathbb{Z}G/G^{(i)}$-module M, where the action of $G/G^{(i)}$ on M is given by the conjugation action on $G^{(i)}/G^{(i+1)}$.

From module theory we see that there exists a series $M = M^{(0)} > M^{(1)} > ... > M^{(r_i)}$, where each section is an irreducible $GF(p)$ $G/G(i)$-module for some prime p. Using these series we obtain a refinement $G = G_{(0)} > G_{(1)} > ... > G_{(n)} = \langle 1 \rangle$ of the commutator series with the properties:

- The series is normal,
- Each section $G_{(i)}/G_{(i+1)}$ is elementary abelian of prime power order, and is irreducible as a $G/G_{(i)}$-module.
- If $G = H_{(0)} > H_{(1)} > ... > H_{(t)} = \langle 1 \rangle$ is another series with these properties, then $n = t$ and there exists a permutation $\pi \in S_n$ such that $G_{(i)}/G_{(i+1)}$ is isomorphic to $H_{(i\pi)}/H_{(i\pi+1)}$ (as $GF(p)$ modules).

Note: A PC-presentation defined by a further refinement of this series leads to a "conditioned" presentation. The converse is not always true, because the irreducibility of the sections is not required for a conditioned presentation.

The soluble quotient algorithm uses these series to construct soluble groups. Starting with the trivial group $G/G_{(0)}$, it successively chooses irreducible $G/G_{(i)}$ modules M_i and extensions

$$\zeta_i \in H^2(G/G_{(i)}, M_i)$$

which give rise to exact sequences

$$1 \to M_i \to G/G_{(i+1)} = G/G_{(i)}.M_i \to G/G_{(i)} \to 1.$$

To describe the algorithmic approach, we consider the following situation. Let G be a finite soluble group, M a normal elementary abelian subgroup of G such that M is a $H = G/M$ irreducible module. (The action of H on M is identified with the conjugation action of G/M on the subgroup M.) Then the relations of the group G have the shape

$$g_i^{p_i} = w_i m_i \text{ resp. } g_j^{g_i} = w_{ij} m_{ij}, m_i, m_{ij} \in M$$

where w_i, w_{ij} are the canonical representatives of H in G. Then $\bar{g_i}^{p_i} = \bar{w_i}$ resp. $\bar{g_j}^{\bar{g_i}} = \bar{w_{ij}}$ is a PC-presentation of H and the set of images $\left((g_i^{p_i-1}, g_i) \to m_i, (g_j, g_i) \mapsto m_{ij}\right)$ determines a unique element of the cocycle space $C^2(H, M)$.

According to the p-group situation, we call the system $t = (m_i, m_{ij})$, the tail defining $G = H.M_t$ as an extension of M by H.

Not every choice of a system $t = (m_i, m_{ij})$ defines an extension. To make sure that t corresponds to an element s of $C^2(G, M)$ it must satisfy a certain equation system, the so-called consistency equations (see Vaughan-Lee). These are linear homogeneous equations in M, so the solution space can be determined.

For the construction of soluble quotients we also have to find the epimorphism $\varepsilon : F \twoheadrightarrow G$. The existence of the epimorphism is obviously a restriction of the possible images G, and it can be checked simultaneously with constructing G. Let G be an extension $G = H.M$, where M is a H-module and H is a known soluble quotient $\delta : F \twoheadrightarrow H$. Then δ is uniquely determined by the images $\delta(f_i) = h_i, 1 \le i \le r$, where $\{f_1, \ldots, f_r\}$ is a generating set of F. We want to find a lift ε of δ, i.e. $\varepsilon(f) = \delta(f) \bmod M$ for all $f \in F$. This means that $\varepsilon(f_i) = h_i x_i$ for all $i \le r$, where $x_i \in M$ are to be determined. Since ε will be a homomorphism, we require $\varepsilon(r_j(f_1, \ldots, f_r)) = 1_G$ for the defining relations r_j of F. This leads to a linear equation system for the variables $(x_1, \ldots, x_r) \in M^r$. We solve this equation system together with the consistency equations, hence we find a subspace S of $M^r \times H^2(H, M)$ of those extensions for which the epimorphism δ has a lift. Let us call an element of S an extended tail.

Let K be the minimal splitting field of M, i.e. the character field of the unique character of H which corresponds to M. Then M is obviously a KH-module and S is also a K-space. The space S has a K-subspace S_S of split extensions, i.e. the projection of S_S into $H^2(H, M)$ is only the trivial element. For any element $t \in S/S_S$ the corresponding map $\varepsilon_t : F \to H.M_t$ is necessarily surjective, hence defines a soluble quotient. Let s_1, s_2 be two elements of S/S_S and let $\varepsilon_i : F \twoheadrightarrow H.M_{s_i} =: G_i$ denote the corresponding soluble quotients. If s_1 and s_2 are K-linear dependent, the groups G_1 and G_2 will be isomorphic. Unfortunately, the converse is not true, even K-linear independent elements may lead to isomorphic groups. Nevertheless, if s_1 and s_2 are independent, the kernels of the epimorphisms will be different, hence one can iterate the lifting to

$$\varepsilon_{1,2} : F \twoheadrightarrow (G_1).M_{s_2} = (H.M_{s_1}).M_{s_2} = H.(M \oplus M)_{s_1 \cdot s_2}.$$

(The last equality can be read as a definition of $s_1 \cdot s_2$ in the righthand side term.)

The dimension $a = dim_K(S/S_S)$ is therefore characterised as the maximal multiplicity

$$a = max\left\{z \in \mathbb{Z}^+ \mid \exists \varepsilon : F \twoheadrightarrow H, M^z\right\}.$$

S_S itself also has a K-subspace S_C, for which the map

$$\varepsilon_s : F \to H.M_s, s \in S_C$$

is not surjective. The K-dimension $b = dim_K(S_S/S_C)$ is again characterised as the maximal multiplicity

$$b = \max\left\{z \in \mathbb{Z}^+ \mid \exists \varepsilon : F \twoheadrightarrow H, M^z\right\}.$$

Moreover, after taking a maximal extension $\varepsilon : F \twoheadrightarrow H.M^b$, M never has to be considered for split extensions again.

Calculating the Relevant Primes

A crucial step in finding finite soluble quotients $\varepsilon : F \twoheadrightarrow G$ is the calculation of the relevant primes, i.e. the prime divisors of $|G|$. This is the most time consuming part of the algorithm, so it is crucial to apply it as efficiently as possible, and any other information about possible prime divisors is very helpful. We do not explain this calculation in detail. However, to use the functions and options provided by Magma in a correct manner, we outline the idea of the calculation.

Let $\varepsilon : F \twoheadrightarrow G$ be a finite soluble quotient and let N denote the kernel of ε. If a module M allows an extension $\bar{\varepsilon} : F \twoheadrightarrow G.M$, then M must be a constituent of N/N', and the relevant primes are the prime divisors of N/N'. Again N/N' can be viewed as an F/N- module, hence an H-module. Therefore it is a direct sum $N/N' \cong M_{p_1} \oplus M_{p_2} \oplus \ldots \oplus \mathbb{Z}^d$, where the M_{p_i} are finite modules of order a power of the prime p_i. Let p not divide $|H|$. Then by Zassenhaus the extension $H.M_p$ must split. We consider an irreducible module M in the head of M_p, i.e. there is an H-epimorphism $M_p \twoheadrightarrow M$. Then there is a valid soluble quotient $\varepsilon : F \twoheadrightarrow H.M$ and the extension $H.M$ splits.

Now there exists an irreducible $\mathbb{Z}H$- module L such that M is a $GF(p)$-modular constituent of L/pL. Moreover, $\Delta : H \to GL(L)$ is the corresponding representation of H and any Δ-module will have this property.

Now using the theory of space groups, one can construct homomorphisms $\varepsilon_1 : F \twoheadrightarrow H.L$ and $\varepsilon_2 : H.L \to$ $\to H.M$ such that the composition is surjective. Hence p can be detected in Δ. Let P_Δ denote the set of primes obtained from Δ. To find these primes, we have to know a set D of representatives of the irreducible rational representations of F/N, hence of H. The set of prime divisors of K/K' is a subset of

$$P = \bigcup_{\Delta \in D} P_\Delta \cup \{p \mid p \text{ prime}, p \mid |G|\}.$$

We want to point out the following:

•

: Usually the set of primes dividing $|K/K'|$ is a proper subset of P, because the primes dividing $|G|$ may or may not divide $|K/K'|$.

- Conversely, if p does not divide $|G|$, then there certainly exists a module M in characteristic p such that there is a soluble quotient $F \twoheadrightarrow G.M$, and the extension splits.

- The algorithm can also recognise infinite abelian sections. This means that already $\varepsilon_1 : F \twoheadrightarrow H.L$ is surjective. All primes are relevant and no maximal finite soluble quotient exists.

- Let $\phi : F \twoheadrightarrow H$ be a lift of $\varepsilon : F \twoheadrightarrow G$, i.e. G is a quotient of H: $\phi : H \twoheadrightarrow G$. If the relevant primes of G are known, these are also relevant primes of H, and only those representations Δ of H must be considered for which $\ker \phi \not\subset \ker \Delta$ is valid.

The Intrinsics

Magma provides two different ways to calculate finite soluble quotients: a main function for the whole calculation, and a process which gives control over each individual step.

Let F be a finitely presented group.

```
SolubleQuotient(F, n : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt
SolvableQuotient(F, n : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum,
                MonStgElt
SolubleQuotient(F : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt
SolvableQuotient(F : parameters): GrpFP, RngIntElt -> GrpPC, Map, SeqEnum, MonStgElt
SolubleQuotient(F, P : parameters): GrpFP, Set -> GrpPC, Map, SeqEnum, MonStgElt
SolvableQuotient(F, P : parameters): GrpFP, Set -> GrpPC, Map, SeqEnum, MonStgElt
```

```
NilpotencyClass     : RngIntElt                    Default: 0
LCSlimit            : RngIntElt                    Default: 0
QuotientSize        : RngIntElt                    Default: 0
RatRepDegreeLimit   : RngIntElt                    Default: 0
SQTimeLimit         : RngIntElt                    Default: 0
MSQ_PrimeSearchModus: RngIntElt                    Default: 0
MSQ_ModulCalcModus  : RngIntElt                    Default: 0
MSQ_CollectorModus  : RngIntElt                    Default: 2
Print               : RngIntElt                    Default: -1
MSQ_Messages        : Verbose                      Default: 2
MSQ_PrimeSearch     : Verbose                      Default: 15
MSQ_RepsCheck       : Verbose                      Default: 3
MSQ_RepsCalc        : Verbose                      Default: 3
MSQ_Collector       : Verbose                      Default: 1
MSQ_TraceFunc       : Verbose                      Default: 2
```

Find a soluble quotient $\varepsilon : F \twoheadrightarrow G$ with a specified order. The argument n must be a nonnegative integer and P must be a set of primes.

The three forms reflect possible information about the order of an expected soluble quotient. In the first form the order of G is given by n, if n is greater than zero. If n equals zero, nothing about the order is known and the relevant primes will be calculated completely.

The second form, with no n argument, is equivalent to the first with $n = 0$. This is a standard argument, and usually it is the most efficient way to calculate soluble quotients.

Note that, if $n > 0$ is not the order of the maximal finite soluble quotient, it may happen that no group of order n can be found, since an epimorphic image of size n may not be exhibited by the chosen series.

In the third form a set P of relevant primes is given. The algorithm calculates the biggest quotient such that the order has prime divisors only in P. The set P may have a zero as an element, this is just for consistency reasons. It is equivalent to the first form with n equal zero.

The returned values are the group G and the epimorphism $\varepsilon : F \twoheadrightarrow G$. The third returned value is a sequence describing the series and modules by which G has been constructed.

The fourth value is a string which explain the reason for termination. The following list gives the termination conditions. The algorithm terminates normally if:

1. A quotient of the given order has been constructed.
2. A maximal quotient (with respect to the conditions given on the order) has been constructed.

The algorithm will be aborted and returns a warning if:

3. A bound on the length of a series or subseries has been hit.
4. A limit on the size of the quotient, or on the running time, has been hit.
5. The algorithm detects a free abelian section.

With the following options one can define abort conditions corresponding to the third and fourth item. The idea of all these conditions is to control the occurrence of infinite soluble quotients.

`NilpotencyClass`: Limits the nilpotent length of the series along which the quotient is constructed. The default value of zero means no limit.

If the algorithm hits the limit, it gives a warning message and returns the last soluble quotient.

`LCSlimit`: Limits the length of the lower descending series within a section. The default value of zero means no limit.

`QuotientSize`: If the value n is bigger than zero, the algorithm returns if a quotient of order bigger or equal to n has been found. A warning message will be printed.

Note: Since an additive bound on a group order is not as meaningful as a multiplicative bound, this option is only useful as a break condition when the quotient gets too big for further calculations. The return quotient which hits such a bound is somewhat randomly chosen, since only a change in the order of checking modules may lead to other quotients.

The next two options bound the search for the relevant primes rather than the quotient.

`RatRepDegreeLimit`: Limits the degree of the rational representations used to find the relevant primes; one of larger degree is not constructed, and one already in hand is skipped with a message. The default value of zero means no limit. Imposing a limit can lose relevant primes, so a quotient found under one is not known to be maximal.

`SQTimeLimit`: Limits the CPU time, in seconds, that the whole calculation may take. The value is turned into a deadline when the algorithm starts, and is tested at each step; once it has passed, the algorithm returns the quotient it has reached, with `"Soluble quotient reached time limit."` as its termination string. The default value of zero means no limit.

With the following options the strategy of the algorithm and some subalgorithms can be chosen. The series along which the quotient is constructed is always the sag-series.

`MSQ_PrimeSearchModus`: Defines at what status of the algorithm the relevant prime search is called.

The possible choices reflect the different intentions of constructing a soluble quotient; for the general situation, (i.e. finding a finite soluble quotient without any information about relevant primes and check its maximality) this option makes only little difference in runtime behaviour.

0: No calculation of relevant primes. This is the default value, and it stands when the arguments already settle the primes, as a positive n or a set P does. When they do not, the algorithm substitutes a modus of its own: 4 for the form taking F alone, and 3 for $n = 0$.

1: The relevant primes will be calculated after the soluble quotient algorithm (with the given input) terminates normally. Possibly new relevant primes are returned in a message. If, for example, the second argument is a set S (so no limit on the exponent) and no new relevant primes have been found, the maximality of the soluble quotient is proved.

2: As 1, but continues the algorithm when finding new relevant primes.

3: Perform the relevant prime calculation after a "main" step in the series, i.e. after completing a nilpotent section in the sag-series. This is what a modus of 0 becomes when $n = 0$. It is a good choice if one wants to construct large finite quotients quickly.

- **Note:** This option can cause problems in case of a sag-series when infinite soluble quotients exist. For finite quotients, it seems to be the best choice.

4: Perform the relevant prime calculation after calculating an elementary abelian layer. This option is preferable, if the sag-series is used and an infinite soluble quotient is possible.

5: Perform the relevant prime calculation after each successful lift of a quotient. This option is preferable when infinite sections shall be detected as soon as possible (with respect to the chosen series).

`MSQ_ModulCalcModus`: In the construction of soluble quotients using a sag-series one can restrict the number of modules by using tensor products and skew symmetric products. This can improve the performance in the case of big soluble quotients, for small quotients the overhead may invalidate the improvement. The possible values are:

0: Do not apply this technique (default).

1: Fast version, just apply those parts which can be calculated quickly.

2: Full version, this is only recommended for "big" soluble quotients, i.e. quotients with long descending series in nilpotent sections.

`MSQ_CollectorModus`: Defines the setup modus for the symbolic collector, i.e. the ratio of precalculation to dynamic setup:

0: Full precalculation, preferable for small soluble groups.

1: Partial precalculation (test version).

2: Dynamic setup (default).

The function also provides a general print option determining the amount of timings status information during the function call. Additionally there are some verbose flags which determine the amount of information given about various subalgorithms. If both a general print value and a verbose flag are given, the verbose flag has higher preference.

`Print`: Determines what timing information and status messages are given during the calculation (0 = no printing, 5 = maximal information). The default of -1 sets no level of its own, leaving the verbose flags below as they already stand.

`MSQ_Messages`: If set to 1, the sizes of new soluble quotients are printed.

`MSQ_PrimeSearch`: Bitflag for print levels during the calculation of relevant primes:

1: Timings and statistics about the calculation of rational representations.

2: Timings for transforming rational into integral representations.

4: Timing for finding the relevant primes.

8: Printing of new relevant primes.

`MSQ_RepsCheck`: Bitflag for print levels during the check of module extensions:

1: Timing for checking extensions of modules.

2: Statistics about the modules to be checked.

`MSQ_RepsCalc`: Bitflag for print levels during the module calculation:

1: Timing information about the module calculation.

2: Statistics about the module calculation.

`MSQ_Collector`: If set to 1, the timing for the setup of the symbolic collector is printed.

`MSQ_TraceFunc`: Give messages about the main function calls (1) resp. most function calls (2) in the Magma language during the algorithm.

80.11.7 Simple Group Quotients

We describe intrinsics for finding homomorphisms of an fp-group G onto simple groups. This may be useful when the presentation for G defines a perfect group. The algorithm maintains a list of simple groups to try and the function *Homomorphisms* is used to run through the list looking for a homomorphism onto G.

The list of simple groups supplied in versions beginning with V2.25 contain all non-abelian simple groups with order $\leq 10^{10}$. Such a list is dominated by $\mathrm{PSL}(2,q)$'s with q odd. In this implementation these $\mathrm{PSL}(2,q)$'s are treated as an infinite family rather than stored individually, and so continue beyond the above limit.

```
SimpleQuotients(F, deg1, deg2, ord1, ord2: parameters): GrpFP, RngIntElt, RngIntElt,
                RngIntElt, RngIntElt -> List
SimpleQuotients(F, ord1, ord2: parameters): GrpFP, RngIntElt, RngIntElt -> List
SimpleQuotients(F, ord2: parameters): GrpFP, RngIntElt -> List
```

```
Family  : Any                          Default: "All"
Limit   : RngIntElt                    Default: 1
HomLimit: RngIntElt                    Default: 0
```

Uses *Homomorphisms* to find epimorphisms from F onto simple groups in a fixed list. The arguments $deg1$ and $deg2$ are respectively lower and upper bounds for the degree of the image group. If the degree arguments are not present then bounds of 5 and 10^7 are used. The arguments $ord1$ and $ord2$ are respectively lower and upper bounds for the orders of the image group. (Setting $ord2$ low enough is particularly important if a quick search is wanted.) If $ord1$ is not given then it defaults to 1.

The return value is a list of sequences of epimorphisms found. Each sequence contains epimorphisms onto one simple group. The parameter `Limit` limits the number of successful searches to be carried out by *Homomorphisms*. The default value is 1, so by default the search terminates with the first simple group found to be a homomorphic image of F.

The parameter `HomLimit` limits the number of homomorphisms that will be searched for by any particular call to *Homomorphisms*. It defaults to zero, so that all homomorphisms for any group found will be returned.

The parameter `Family` selects sublists of the main list to search. Possible values of this parameter are `"All"`, `"PSL"`, `"PSL2"`, `"Mathieu"`, `"Alt"`, `"PSp"`, `"PSU"`, `"Other"`, and `"notPSL2"`; sets of these strings are also allowed, which searches on the union of the appropriate sublists.

SimpleQuotientProcess(F, deg1, deg2, ord1, ord2: parameters): GrpFP, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> Rec

Family: Any	Default: "All"

Produce a record that defines a process for searching for simple quotients of F as *SimpleQuotients* does. Calling this function sets up the record and conducts the initial search until a quotient is found. Continuing the search for another quotient is done by calling *NextSimpleQuotient*. Extracting the epimorphisms found is achieved using *SimpleEpimorphisms*, and testing if the process has expired is the task of *IsEmptySimpleQuotientProcess*.

NextSimpleQuotient(~P): Rec

When P is a record returned by *SimpleQuotientProcess*, advance the search to the next simple group which is a homomorphic image of the finitely presented group. Does nothing if the process has expired.

IsEmptySimpleQuotientProcess(P): Rec -> BoolElt

When P is a record returned by *SimpleQuotientProcess*, test whether or not P has expired.

SimpleEpimorphisms(P): Rec -> SeqEnum, Tup

When P is a record returned by *SimpleQuotientProcess*, extract the most recently found epimorphisms onto a simple group, plus a tuple describing the image group. This is a valid operation when *IsEmptySimpleQuotientProcess* returns `false`.

Example: Simple Quotients of a Perfect Group (ex-b2e434)

We take a perfect finitely presented group and search for simple quotients.

```
> F := FPGroup<a,b,c|a^13,b^3,c^2,a = b*c>;
> IsPerfect(F);
true
> L := SimpleQuotients(F,1, 100, 2, 10^5:Limit := 2);
> #L;
2
> for x in L do CompositionFactors(Image(x[1])); end for;
    G
    |  A(1, 13)               = L(2, 13)
    1
    G
    |  A(2, 3)                = L(3, 3)
    1
```

(continues on next page)

(continued from previous page)

```
> L[2,1];
Homomorphism of GrpFP: F into GrpPerm: $, Degree 13, Order
2^4 * 3^3 * 13 induced by
F.1 |--> (1, 10, 4, 5, 11, 8, 3, 6, 7, 12, 9, 13, 2)
F.2 |--> (2, 10, 4)(3, 6, 7)(5, 11, 13)(8, 12, 9)
F.3 |--> (1, 10)(2, 5)(3, 12)(8, 13)
> #L[2];
2
```

We've found L(2,13) and L(3,3) as images, with 2 inequivalent homomorphisms onto the second. We'll try a process looking through a smaller family.

```
> P := SimpleQuotientProcess(F,1, 100, 2, 10^6:Family:="PSU");
> IsEmptySimpleQuotientProcess(P);
false
> eps, info := SimpleEpimorphisms(P);
> info;
<65, 62400, PSU(3, 4)>
```

We've found PSU(3,4) of order 62400 and degree 65 as an image. We continue with this process.

```
> NextSimpleQuotient(~P);
> IsEmptySimpleQuotientProcess(P);
true
```

No, there are no more within the limits given.

80.11.8 The L_2-Quotient Algorithm

Given an fp-group G, the L_2-quotient algorithm of Plesken, Fabianska, and Jambor computes all quotients of G which are isomorphic to the 2-dimensional linear groups $PSL(2,q)$ or $PGL(2,q)$ simultaneously for any prime power q. The algorithm can handle the case of infinitely many quotients, and also works for very large prime powers. In practice it is fast when the number of generators is small and can be used to show that many fp-groups are infinite. While the simple groups $PSL(2,q)$ represent just one dimension in one family out of 16 families of simple groups, in a given range of orders, they are very numerous compared to other simple groups. See [Jam12] or [Jam15] for more information about the L_2-quotient algorithm.

Note that the algorithm does **not** return images onto the groups $\mathrm{PSL}(2,2)$, $\mathrm{PSL}(2,3)$, and $\mathrm{PSL}(2,4) = \mathrm{PSL}(2,5)$.

Basic Usage

`L2Quotients(G): GrpFP -> [ L2Quotient ]`

This is the main method. It takes as parameter a finitely presented group and returns a sequence of L_2-quotients, which have type `L2Quotient`.

Example: Computing L2-Quotients (ex-656deb)

We first compute L_2-quotients of a certain finitely-presented group.

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^7, (a,b)^11 >;
> L2Quotients(G);
[
    L_2(43)
]
> H := FPGroup< a,b,c | a^3, b^7, c^19, (a*b)^2, (a*c)^2, (b*c)^2, (a*b*c)^2 >;
> L2Quotients(H);
```

(continues on next page)

(continued from previous page)

```
[
    L_2(113)
]
```

This means that G has $\mathrm{PSL}(2,43)$ as quotient, but no other $\mathrm{PSL}(2,q)$ or $\mathrm{PGL}(2,q)$ is a quotient of G. Similarly, the only L_2-quotient of H is $\mathrm{PSL}(2,113)$.

The next example has more quotients:

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^16, (a,b)^11 >;
> L2Quotients(G);
[
    PGL(2,23),
    PGL(2,23),
    PGL(2,463)
]
```

Here $\mathrm{PGL}(2,23)$ occurs twice. This means that there are two epimorphisms of G onto $\mathrm{PGL}(2,23)$ which do not differ by an automorphism of $\mathrm{PGL}(2,23)$. In other words, the kernels of the epimorphisms are distinct.

Some groups have infinitely many L_2-quotients. This is indicated by one of the L_2-quotients `L_2(infty^k)`, `L_2(p^(infty^d))`, or `L_2(infty^(infty^d))`, as the follow examples illustrate.

```
> G := FPGroup< a,b,c | a^3, b^7, (a*b)^2, (a*c)^2, (b*c)^2, (a*b*c)^2 >;
> L2Quotients(G);
[
    L_2(infty^6)
]
> H := FPGroup< a,b,c | a^3, (a,c) = (c,a^-1), a*b*a = b*a*b,
>                       a*b*a*c^-1 = c*a*b*a >;
> L2Quotients(H);
[
    L_2(3^infty)
]
> K := FPGroup< a,b | a^3*b^3 >;
> L2Quotients(K);
[
    L_2(infty^infty)
]
```

See below for an interpretation of this output, and for instructions as how to use Magma to get more information about the quotients.

`GetMatrices(Q): L2Quotient -> GrpMat, SeqEnum`

For a finite L_2-quotient Q of G, that is, a quotient $L_2(p^k)$ or $\mathrm{PGL}(2,p^k)$, this function returns a matrix group H and a sequence A of 2×2 matrices in H, where $A[i]$ corresponds to the i-th generator of G. Note that $G\to H$, $G.i\to A[i]$ does not in general define a homomorphism, but the induced map $G\to H/Z(H)$ does.

Example: Matrix Groups of L2-Quotients (ex-d6f5a7)

The following example illustrates how the quotient matrix groups can be accessed.

```
> H := FPGroup< a,b,c | a^3, b^7, c^19, (a*b)^2, (a*c)^2, (b*c)^2, (a*b*c)^2 >;
> quot := L2Quotients(H); quot;
[
    L_2(113)
]
```

(continues on next page)

(continued from previous page)

```
> H, A := GetMatrices(quot[1]);
> H;
MatrixGroup(2, GF(113))
Generators:
    [  0   1]
    [112 112]

    [  0  85]
    [109  24]

    [102 104]
    [ 63  72]
> A;
[
    [  0   1]
    [112 112],

    [  0  85]
    [109  24],

    [102 104]
    [ 63  72]
]
```

Example: Finite Quotients of the Modular Group (ex-0ea7a6)

The next few examples are taken from Conder, Havas, and Newman [CHN11].

```
> Gamma< x, y > := FPGroup< x, y | x^2, y^3 >;
> u := x*y; v := x*y^-1;
> G := quo< Gamma | u^10*v^2*u*v*u*v^2 >;
> quot := L2Quotients(G); quot;
[
    L_2(5^2)
]
> H, A := GetMatrices(quot[1]);
> H;
MatrixGroup(2, GF(5^2))
Generators:
    [        2 4*$.1 + 2]
    [        0         3]

    [0 4]
    [1 4]
```

There are other quotients of the modular group with only finitely many L_2 quotients:

```
> G := quo< Gamma | u^3*v*u^3*v*u^3*v^2*u*v^2 >;
> quot := L2Quotients(G);
> quot;
[
    PGL(2,13)
]
> G := quo< Gamma | u^3*v*u^3*v^2*u*v^3*u*v^2 >;
> quot := L2Quotients(G);
> quot;
[]
```

This tells us that the first group has only one L_2 quotient, isomorphic to $\mathrm{PGL}(2,13)$, and the second group has no L_2 quotients at all.

Example: L2-Quotients of Coxeter Presentations (ex-3ecfb1)

Next, we look at Coxeter presentations of the form

$$(l, m|n, k) = \langle x, y | x^l, y^m, (xy)^n, (x^{-1}y)^k \rangle$$

and

$$(l, m, n; q) = \langle x, y | x^l, y^m, (xy)^n, [x, y]^k \rangle$$

for various values of l, m, n, k, q.

```
> G := FPGroup< x,y | x^8, y^9, (x*y)^5, (x^-1*y)^7 >;
> quot := L2Quotients(G);
> quot;
[
    L_2(71),
    L_2(71),
    L_2(71),
    L_2(2521),
    L_2(41),
    L_2(3056201),
    L_2(239),
    L_2(449),
    L_2(3876207679),
    L_2(1009),
    L_2(29113631)
]
```

The group $\mathrm{PSL}(2,71)$ occurs three times. This means that there are three essentially different epimorphisms of G onto $\mathrm{PSL}(2,71)$, i.e., there is no automorphism of $\mathrm{PSL}(2,71)$ which transforms one epimorphism into another.

Here is another example, this time for the other family.

```
> G := FPGroup< x, y | x^4, y^3, (x*y)^5, (x,y)^6 >;
> L2Quotients(G);
[
    L_2(79),
    L_2(3^2),
    L_2(5^2)
]
```

There are three groups in the second family for which it is not known whether they are finite or infinite (Havas & Holt (2010) [HH10]). They are $(3,4,9;2)$, $(3,4,11;2)$, and $(3,5,6;2)$. Each of them has only one L_2 image.

```
> G := FPGroup< x, y | x^3, y^4, (x*y)^9, (x,y)^2 >;
> L2Quotients(G);
[
    L_2(89)
]
> G := FPGroup< x, y | x^3, y^4, (x*y)^11, (x,y)^2 >;
> L2Quotients(G);
[
    L_2(769)
]
```

(continues on next page)

(continued from previous page)

```
> G := FPGroup< x, y | x^3, y^5, (x*y)^6, (x,y)^2 >;
> L2Quotients(G);
[
    L_2(61)
]
```

Intermediate Usage

`L2Quotients(M): AlgMatElt -> [ L2Quotient ]`

A Coxeter group is a finitely presented group on m generators, such that the only other relations are $(g_i g_j)^{M_{ij}}$, where M is a symmetric matrix with non-negative integer entries and 1's along the diagonal. The matrix M is also called a Coxeter matrix. Often when dealing with Coxeter groups we are only interested in smooth quotients, that is, those which preserve the orders. The method `L2Quotients` accepts a Coxeter matrix as input, and computes the smooth quotients of the associated Coxeter group. However, it omits the quotients in characteristic p if $M_{12} = p$. Furthermore, if a quotient has characteristic p and p divides some entry of M, then the quotient is not guaranteed to be smooth.

Example: L2-Quotients from a Coxeter Matrix (ex-cc44da)

```
> M := Matrix([[1,2,3,3],[2,1,4,5],[3,4,1,6],[3,5,6,1]]);
> L2Quotients(M);
[
    L_2(4391),
    L_2(71)
]
```

`L2Quotients(G): GrpFP -> [ L2Quotient ]`

```
ExactOrders: SeqEnum                    Default: []
```

Even if the finitely presented group in question is not a Coxeter group, we often are only interested in quotients where certain orders are satisfied (for instance, we might know that the generator must have a certain order). Usually this yields a great speed-up in the computation, or even allows the computation to finish in the first place. The orders can be specified using the optional parameter `ExactOrders`. This is a list of pairs, where the first entry is a word in the group, and the second entry is the order.

Example: L2-Quotients with Exact Orders (ex-39db62)

```
> G< a,b,c > := FPGroup< a,b,c | a^16, b^4, c^2, (a*b)^8, (a,b)^4 >;
> time quot := L2Quotients(G : ExactOrders := [< a, 16 >, < b, 4 >,
>    < c, 2 >, < a*b ,8 >, < (a,b), 4 >]);
Time: 1.530
```

Advanced Usage

`L2Quotients(G): GrpFP -> [ L2Quotient ]`

```
Saturate           : Boolean                Default: false
AddMoreRelations   : Boolean                Default: false
Primes             : SeqEnum                Default: []
ExcludePrimes      : SeqEnum                Default: []
SingleInfinite     : Boolean                Default: false
UseRandomTechniques: Boolean                Default:
FactorizationBound : RngIntElt              Default:
```

(continues on next page)

(continued from previous page)

```
TrialDivisionBound : RngIntElt                    Default:
GroebnerBasisBound : RngIntElt                    Default:
```

The algorithm discards prime ideals containing $\rho = x_1^2 + x_2^2 + x_{12}^2 - x_1x_2x_{12} - 4$. If the optional boolean parameter `Saturate` is `true`, then prior to the primary decomposition the ideal is saturated at ρ, which can be faster in some cases.

If the optional boolean parameter `AddMoreRelations` is `true`, then the algorithm adds further relations to the ideal, which speeds up the computation in some cases.

Given a sequence P of primes, this variant only produces quotients Q in characteristic p only, where p must lie in P.

The optional parameter `ExcludePrimes` is a sequence of primes. The algorithm does not compute L_2-quotients in characteristic p if p is in `exclude`.

With this option, the function returns immediately when an infinite family of quotients is found and this family is returned. This is useful when attempting to prove an FP group infinite, since only one infinite family is needed to prove the group infinite.

These parameters are passed to the method `MinimalAssociatedPrimes` (see documentation there).

Handling Infinite L_2-quotients

Quotients of type $\mathrm{L}_2(\infty^k)$

If G has a quotient $\mathrm{L}_2(\infty^k)$, then for almost all (all but finitely many) primes p, G has finitely many quotients of type $\mathrm{PSL}(2, p^r)$ or $\mathrm{PGL}(2, p^{r/2})$ with $r \leq k$. So $\mathrm{L}_2(\infty^k)$ is a mnemonic, where ∞ in the base stands for infinitely many primes, and k stands for the highest possible exponent.

There are two basic methods to further investigate such quotients.

`SpecifyCharacteristic(Q, n): L2Quotient, RngIntElt -> [ L2Quotient ]`

Compute the L_2-quotients in characteristic $p \mid n$.

Example: Specifying the Characteristic `(ex-dd2847)`

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^7 >;
> quot := L2Quotients(G); quot;
[
    L_2(infty^3)
]
> Q := quot[1];
> SpecifyCharacteristic(Q, 7);
[
    L_2(7)
]
> SpecifyCharacteristic(Q, 11);
[
    L_2(11^3)
]
> SpecifyCharacteristic(Q, 13);
[
    L_2(13),
    L_2(13),
    L_2(13)
]
```

`AddGroupRelations(Q, R): L2Quotient, [ GrpFPElt ] -> [ L2Quotient ]`

Compute the L_2-quotients which satisfy the additional relations R.

Example: Adding Group Relations (ex-6e7481)

```
> G<a,b> := FPGroup< a,b | a^2, b^3, (a*b)^7 >;
> quot := L2Quotients(G); quot;
[
    L_2(infty^3)
]
> Q := quot[1];
> AddGroupRelations(Q, [(a*b*a*b^-1*a*b)^12]);
[
    L_2(13),
    L_2(7)
]
```

Quotients of type $\mathrm{L}_2(p^{\infty^d})$

If G has a quotient $\mathrm{L}_2(p^{\infty^d})$, then there are infinitely many positive integers k such that G has a quotient of type $\mathrm{PSL}(2, p^k)$ or $\mathrm{PGL}(2, p^k)$. So $\mathrm{L}_2(p^{\infty^d})$ is a mnemonic, where ∞ in the exponent stands for infinitely many possible exponents. The parameter d describes the degree of infinity, and is omitted if $d = 1$.

Again, we can use `AddGroupRelations` to sudy this quotient further.

Example: Adding Group Relations in Characteristic 3 (ex-d05ba8)

```
> H< a,b,c > := FPGroup< a,b,c | a^3, (a,c) = (c,a^-1), a*b*a = b*a*b,
>    a*b*a*c^-1 = c*a*b*a >;
> quot := L2Quotients(H); quot;
[
    L_2(3^infty)
]
> Q := quot[1];
> AddGroupRelations(Q, [((b^2*c^3)^2*a)^5]);
[
    L_2(3^2),
    L_2(3^4)
]
```

`AddRingRelations(Q, R): L2Quotient, [ RngMPolElt ] -> [ L2Quotient ]`

Compute the L_2-quotients whose traces satisfy the polynomial relations given in R.

Example: Adding Ring Relations (ex-6b7dfd)

```
> H< a,b,c > := FPGroup< a,b,c | a^3, (a,c) = (c,a^-1), a*b*a = b*a*b,
>   a*b*a*c^-1 = c*a*b*a >;
> quot := L2Quotients(H); quot;
[
    L_2(3^infty)
]
> Q := quot[1];
> Q`Ideal;
Ideal of Graded Polynomial ring of rank 7 over Integer Ring
Order: Grevlex with weights [3, 2, 2, 2, 1, 1, 1]
Variables: x123, x23, x13, x12, x3, x2, x1
Variable weights: [3, 2, 2, 2, 1, 1, 1]
Inhomogeneous, Dimension >0
Groebner basis:
[
    x123^2 + 2*x123*x3 + 1,
```

(continues on next page)

(continued from previous page)

```
    x23 + 2*x3,
    x13 + 2*x3,
    x12 + 2,
    x2 + 1,
    x1 + 1,
    3
]
> R< x123, x23, x13, x12, x3, x2, x1 > := Generic(Q`Ideal);
> AddRingRelations(Q, [x3^(3^4) - x3]);
[
    L_2(3^2),
    L_2(3^4),
    L_2(3^4),
    L_2(3^4),
    L_2(3^4),
    L_2(3^4),
    L_2(3^4),
    PGL(2,3^2),
    PGL(2,3^2),
    L_2(3^4),
    L_2(3^8),
    L_2(3^8),
    L_2(3^8),
    L_2(3^8),
    L_2(3^8),
    L_2(3^4)
]
```

Quotients of type $\mathrm{L}_2(\infty^{\infty^d})$

If G has a quotient $\mathrm{L}_2(\infty^{\infty^d})$, then for almost all primes p and infinitely many positive integers k, G has a quotient of type $\mathrm{PSL}(2, p^k)$ or $\mathrm{PGL}(2, p^k)$. So $\mathrm{L}_2(\infty^{\infty^d})$ is a mnemonic, where ∞ in the base stands for infinitely many primes, and ∞ in the exponent stands for infinitely many possible exponents. The parameter d describes the degree of infinity, and is omitted if $d = 1$. These quotients can be further investigated using the methods `AddGroupRelations`, `AddRingRelations`, and `SpecifyCharacteristic`.

80.11.9 Infinite L2 Quotients

This section contains functions that use the L2-quotient algorithm of Plesken and Fabianska ([PF09]) to establish the existence or not of an infinite quotient of a finitely-presented group in $\mathrm{PSL}_2(K)$ for K a field of characteristic zero. The algorithm is often used to find surjective quotients over finite fields but as a test for non-finiteness it is easier to work directly in characteristic zero. We make some comments on the relation to finite fields below.

The algorithm first constructs an affine scheme X over $\mathbb{Q}$ from the "trace ideal" of the group G, for which the algebraic points over a field K are in 1-1 correspondence with equivalence classes of representations of G into $SL_2(K)$. If $g_1, \ldots, g_n$ are the generators of G, then the affine coordinates of X correspond to the trace under the representation of various products of the G_i. For example, when $n = 2$, X lies in 3-dimensional affine space and the coordinates correspond to the traces of g_1, g_2, and $g_1 g_2$. The ideal is actually generated by polynomials with coefficients in $\mathbb{Z}$ so X is naturally defined as a scheme over $Spec(\mathbb{Z})$, whose reduction mod p just gives the scheme corresponding to characteristic p representations of G for $p > 2$.

Homomorphisms into PSL_2 rather than SL_2 are dealt with by considering a number of X schemes for a particular G. Each one represents the homomorphisms from the free cover of G to SL_2 that takes each word in the set of relations defining G to either I or $-I$. Each of the 2^r choices of sign for the r defining relations gives an X (by the same procedure) and the totality cover all possibilities for maps to PSL_2.

The set of points of X corresponding to geometrically reducible, dihedral A_4, S_4, or A_5 images in PSL_2

is a closed subscheme Y. In fact the first two possibilities give closed subschemes with equations defined over $\mathbb{Z}$ and the last three give a dimension zero scheme over $\mathbb{Q}$. The overall equations defining Y reduce mod p to those defining the corresponding subscheme in characteristic p for almost all primes p. The explicit equations defining Y are determined by the algorithm as explained for $n = 2$ or 3 in the above reference or in more detail in Fabianska's MSc thesis ([Fab09]). Let U be the open complement of Y in X.

There are two possibilities.

1 A U is non-empty, so an algebraic point gives $\phi : G \to \mathrm{PSL}_2(K)$, K a number field, with infinite image, so G is infinite. Further, for all but finitely many primes p, ϕ can be reduced mod v for any place v of characteristic p AND the reduction ϕ_v corresponds to a point in the analogue of U over the finite field. Thus the reductions give surjections of G onto a $\mathrm{PSL}_2(k)$ [or maybe a $\mathrm{PGL}_2(k)$] for finite fields of any characteristic outside of a finite set.

2 All Us are empty. Here, all the homomorphisms of G into $\mathrm{PSL}_2(K)$ in characteristic zero have geometrically reducible, dihedral, A_4, S_4 or A_5 images and the same holds for K a field of characteristic p for all p outside of a finite set of primes. G may or may not be infinite.

NB: In case 2), there may still be images of G which ARE infinite, but lie in a Cartan or Borel subgroup, so are geometrically reducible, or lie in the normaliser of a Cartan and have an infinite dihedral image. These infinite reducible/dihedral possibilities are not currently checked for.

`HasInfinitePSL2Quotient(G): GrpFP -> BoolElt, SeqEnum`

```
Signs   : SeqEnum                    Default: 0
Full    : BoolElt                    Default: false
IsInfGrp: Verbose                    Default: 1
```

Function to return whether the two-generator finitely-presented group G has an infinite quotient lying in a $\mathrm{PSL}_2(K)$ with K a field of characteristic 0 as described above.

For the convenience of the user, we have provided some parameters to output more detailed information coming from the analysis of the X representation schemes as well as to control which sign variations are considered.

Let ws be the sequence of words giving the defining relations of G (if a defining relation is of the form $w = v$ where v is not the identity word, then the corresponding word is $w * v^{-1}$). Let $ws = [w_1, .., w_r]$.

As described in the introduction, for each of the 2^r combinations of signs attached to the w_i - let s be one - the program will consider the scheme X of homomorphisms of G into PSL_2 such that, when lifted to a homomorphism of the two-generator free cover F of G into SL_2, w_i maps to $s_i * I$.

In some cases, the user may realise that there is no point in considering certain choices of signs. For example, if g_1^2 is a relation in ws, $g_1^2 \mapsto I$ in $SL_2(K)$ means that $g_1 \mapsto 1$ in $\mathrm{PSL}_2(K)$, so the $\mathrm{PSL}_2(K)$ image would be (maybe infinitely) cyclic, and it is a waste of time to consider this possibility. Similarly, if g_1^r, r odd, is a relation then ,without loss of generality, in a PSL_2 to SL_2 lift, $g_1^r \mapsto I$, which could be specified.

The parameter `Signs` allows the user to specify a restricted set of sign options to analyse. `Signs` should be an $0, 1, -1$ or a sequence of length $\#ws$ of such integers. A single value is converted into the sequence containing that value $\#ws$ times. An entry e of 1 or -1 in position i means that the function will only consider sign sequences s with $s[i] = e$, ie homomorphisms where w_i map to eI in the lift to SL_2. If $e = 0$, then there is no condition at place i of the sign sequences considered. The default value for `Signs` is the single value 0.

For each representation space X (corresponding to an allowable choice of signs s), after removing positive dimensional components of Y corresponding to geometrically reducible or dihedral type representations, there is often a zero-dimensional subscheme left. The existence of an infinite (non-cyclic or dihedral) image homomorphism to PSL_2 then comes down to examining the finite set of closed points remaining and finding one that is not geometrically reducible, dihedral, A_4, S_4, or A_5 type. Clearly, once one such is found the procedure can stop. However, it might be of interest

for the user to see the types of ALL of the representations corresponding to closed points if the zero-dimensional analysis stage is performed.

If parameter `Full` is set to `true` (the default is `false`), the program will continue analysing all of the representations in the 0-dimensional locus, even after one corresponding to an infinite image is found. Furthermore a sequence of (signs,types) pairs is also returned which gives, for each sign combination s of maps considered, the sequence of types corresponding to the 0-dimensional locus (or empty if we don't reduce to dimension 0). These types are given as strings: "infinite", "reducible", "dihedral", "A4", "S4", and "A5".

Setting the verbose flag `IsInfGrp` to `true` or 1 gives output of information on the various stages as the function progresses, including the analysis of dimension zero loci.

Example: Infinite and Finite L2-Quotients (ex-5b2c63)

We look at two examples of quotients of the two-generator free group with relations g_1^2, g_2^3, and one further word. The first is infinite, the second is not. As noted above, we may as well specify that g_1^2 maps to $-I$ and g_2^3 to I in SL_2 and this can be done with the `Signs` parameter.

```
> F := FreeGroup(2);
> rel := (F.1 * F.2 * F.1 * F.2 * F.1 * F.2 * F.1 * F.2 * F.1 * F.2 *
>   F.1 * F.2 * F.1 * F.2 * F.1 * F.2^-1 * F.1 * F.2 * F.1 * F.2^-1)^2;
> G := quo<F | [F.1^2 ,F.2^3, rel]>;
> HasInfinitePSL2Quotient(G : Full := true);
true
[ [*
    [ 1, 1, 1 ],
    []
*], [*
    [ 1, 1, -1 ],
    []
*], [*
    [ 1, -1, 1 ],
    []
*], [*
    [ 1, -1, -1 ],
    []
*], [*
    [ -1, 1, 1 ],
    [ A4, A4 ]
*], [*
    [ -1, 1, -1 ],
    [ A5, A5, infinite ]
*], [*
    [ -1, -1, 1 ],
    [ A4, A4 ]
*], [*
    [ -1, -1, -1 ],
    [ A5, A5, infinite ]
*] ]
```

Now try it with the sign restriction.

```
> HasInfinitePSL2Quotient(G : Full := true, Signs := [-1,1,0]);
true
[ [*
    [ -1, 1, 1 ],
    [ A4, A4 ]
*], [*
```

(continues on next page)

(continued from previous page)

```
    [ -1, 1, -1 ],
    [ A5, A5, infinite ]
*] ]
```

The second example is just A_5.

```
> G := quo<F | [F.1^2 ,F.2^3, (F.1*F.2)^5]>;
> HasInfinitePSL2Quotient(G);
false
> HasInfinitePSL2Quotient(G : Full := true, Signs := [-1,1,0]);
false
[ [*
    [ -1, 1, 1 ],
    [ A5 ]
*], [*
    [ -1, 1, -1 ],
    [ A5 ]
*] ]
```

80.11.10 The L_3U_3-Quotient Algorithm

Given a finitely presented group G on two generators, the L_3U_3-quotient algorithm of Jambor [Jam12] computes all quotients of G which are isomorphic to some $\mathrm{PSL}(3,q)$, $\mathrm{PGL}(3,q)$, $\mathrm{PSU}(3,q)$, or $\mathrm{PGU}(3,q)$, simultaneously for all prime powers q. It can handle the case of infinitely many quotients, and also works for very large prime powers.

Note that the algorithm does not return images onto groups $\mathrm{PSL}(3,2) = \mathrm{PSL}(2,7)$.

L3Quotients(G: parameters): GrpFP -> [L3Quotient]

```
ExactOrders  : SeqEnum                    Default: []
ExcludePrimes: SeqEnum                    Default: []
```

Given a group G with 2 generators, this function returns a sequence of L_3-quotients, which have type `L3Quotient`.

Given a sequence of pairs $\langle w, n\rangle$ where w is a word in the generators of G and n is a positive integer, this returns only those quotients Q for which the order of the image of w in Q is n.

Given a sequence P of primes, this variant only produces quotients Q in characteristic p, where p does not lie in P.

SpecifyCharacteristic(Q, p): L2Quotient, RngIntElt -> [L2Quotient]

SpecifyCharacteristic(Q, p): L3Quotient, RngIntElt -> [L3Quotient]

Compute the restrictions of the given quotient object Q to the prime characteristic p.

AddGroupRelations(Q, r): L2Quotient, [GrpFPElt] -> [L2Quotient]

AddGroupRelations(Q, r): L3Quotient, [GrpFPElt] -> [L3Quotient]

Compute quotient objects from Q for the group of Q with relators r added.

GetMatrices(Q): L2Quotient -> GrpMat

GetMatrices(Q): L3Quotient -> GrpMat

Compute the matrix images of the generators of the group of the quotient object Q as a matrix group. Requires Q to be a finite object. Note that `G.i` $\to$ `M.i`, where M is the matrix group returned, does not always define a homomorphism, but the induced map into $M/Z(M)$ is a homomorphism.

Example: Computing L3- and U3-Quotients (ex-0214ac)

We first compute L_3U_3-quotients of a some finitely-presented groups.

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^11, (a,b)^11 >;
> L3Quotients(G);
[
    U_3(43)
]
> H := FPGroup< a,b | a^2, b^3, (a*b)^11, (a,b)^28  >;
> Q := L3Quotients(H); Q;
[
    U_3(43),
    U_3(14057)
]
```

This means that G has $\mathrm{PSU}(3,43)$ as quotient, but no other $\mathrm{PSL}(3,q)$, $\mathrm{PGL}(3,q)$, $\mathrm{PSU}(3,q)$, or $\mathrm{PGU}(3,q)$ is a quotient of G. Similarly, the only $\mathrm{L_3U_3}$-quotients of H are $\mathrm{PSU}(3,43)$ and $\mathrm{PSU}(3,14057)$.

We can further investigate finite quotients using `GetMatrices`.

```
> M := GetMatrices(Q[1]); M;
MatrixGroup(3, GF(43^2))
Generators:
  [ $.1^756  $.1^202       38]
  [ $.1^960 $.1^1243  $.1^934]
  [ $.1^192  $.1^192 $.1^1529]

  [ $.1^379 $.1^1759  $.1^503]
  [$.1^1326  $.1^804  $.1^374]
  [$.1^1214  $.1^524  $.1^612]
```

The next example has more quotients:

```
> G := FPGroup< a,b | a^2, b^3, (a*b)^18, (a,b)^16  >;
> L3Quotients(G);
[
    PGU(3, 71),
    U_3(1889),
    PGU(3, 17),
    PGU(3, 17),
    PGU(3, 17),
    PGL(3, 19)
]
```

In this example, $\mathrm{PGU}(3,17)$ occurs three times. This means that there are three epimorphisms of G onto $\mathrm{PGU}(3,17)$ which do not differ by an automorphism of $\mathrm{PGU}(3,17)$. In other words, the kernels of the epimorphisms are distinct.

Example: Infinitely Many L3-Quotients (ex-ba1556)

Some groups have infinitely many L3-quotients. This is indicated by one of the L3-quotients `L_3(infty^k)`, `L_3(p^(infty^d))`, or `L_3(infty^(infty^d))`. See below for an interpretation of this output, and how to get more information about the quotients.

```
> G := FPGroup< a, b | a^2, b^3, (a*b)^9 >;
> QG := L3Quotients(G); QG;
[
    L_3(infty^18)
]
> H := FPGroup< a, b | a^2, b^3, (a,b)^5, (a, b*a*b*a*b)^3 >;
> L3Quotients(H);
```

(continues on next page)

(continued from previous page)

```
[
    L_3(2^infty)
]
> K := FPGroup< a, b | a^2, b^3 >;
> L3Quotients(K);
[
    L_3(infty^(inft ^2))
]
```

Even without further interpretation of the output, this tells us that these groups are all infinite.

The object `L_3(infty^18)` means that for almost all primes p there are quotients of G of the form $L_3(p^r)$ or $U_3(p^r)$, with $r \leq 18$. We can specify the characteristic to get explicit examples.

```
> SpecifyCharacteristic(QG[1], 37);
[
  PGL(3, 37),
  PGL(3, 37),
  PGL(3, 37)
]
> X := SpecifyCharacteristic(QG[1], 2); X;
[
  PGU(3, 2^3)
]
> M := GetMatrices(X[1]);
> M:Minimal;
MatrixGroup(3, GF(2^18))
> LMGChiefFactors(M);
  G
  |  Cyclic(3)
  *
  |  2A(2, 8)                    = U(3, 8)
  *
  |  Cyclic(3)
  1
```

If G has a quotient `L_3(p^(infty^d))`, then there are infinitely many positive integers k such that G has a quotient of type $\mathrm{PSL}(3, p^k)$, $\mathrm{PGL}(3, p^k)$, $\mathrm{PSU}(3, \mathrm{p}^{\mathrm{k}})$, or $\mathrm{PGU}(3, \mathrm{p}^{\mathrm{k}})$. So `L_3(p^(infty^d))` is a mnemonic, where infty in the exponent stands for infinitely many possible exponents. The parameter d describes the degree of infinity, and is omitted if $d = 1$. We can use `AddGroupRelations` to study this quotient further.

```
> H< a, b > := FPGroup< a, b | a^2, b^3, (a,b)^5, (a, b*a*b*a*b)^3 >;
> quot := L3Quotients(H); quot;
[
    L_3(2^infty)
]
> Q := quot[1];
> AddGroupRelations(Q, [(a*b)^(2*3*5*7)]);
[
    PGL(3, 2^2),
    U_3(2^2)
]
```

Another possibility is to add further ring relations to the ideal describing the L3-quotient, using the intrinsic `AddRingRelations`. It takes an L3-quotient and a list of polynomials, and computes the L3-quotients whose traces satisfy these polynomial relations.

```
> Q := quot[1];
> Q`Ideal;
Ideal of Graded Polynomial ring of rank 10 over Integer Ring
Order: Grevlex with weights [8, 2, 2, 2, 2, 4, 4, 4, 4, 1]
Variables: xcom, x1, xm1, x2, xm2, x12, xm12, xm21, xm2m1, zeta
Variable weights: [8, 2, 2, 2, 2, 4, 4, 4, 4, 1]
Inhomogeneous, Dimension >0
Groebner basis:
[
    xcom + zeta + 1,
    xm12*xm2m1 + zeta,
    x12 + xm12,
    xm21 + xm2m1,
    x1 + 1,
    xm1 + 1,
    x2,
    xm2,
    zeta^2 + zeta + 1,
    2
]
> R< xcom, x1, xm1, x2, xm2, x12, xm12, xm21, xm2m1, zeta > := Generic(Q`Ideal);
> AddRingRelations(Q, [x12^5 + xm21^2 + 1]);
[
    L_3(2^8),
    L_3(2^6)
]
```

80.12 KG-Modules

This section describes some functions for creating finite dimensional $K[G]$-modules for a finitely presented group G. These are obtained from the conjugation action of G on a basis for a maximal elementary abelian quotient of a normal subgroup H of G, where H has finite index in G. For a complete description of the functions available for working with $K[G]$-modules we refer to Chapter Modules over an Algebra and Group Representations.

Note that the function `GModuleAction(M): ModGrp → Map(Hom)` can be used to extract the matrix representation associated to an $K[G]$-module.

All operations described in this subsection may require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function *SetGlobalTCParameters*. For a detailed description of the available parameters and their meanings, we refer to Subsec. *Interactive Coset Enumeration*.

GModulePrimes(G, A): GrpFP, GrpFP -> SetMulti

Let G be a finitely presented group and A a normal subgroup of G of finite index. Given any prime p, the maximal p-elementary abelian quotient of A can be viewed as a $\mathbf{F}_p[G]$-module M_p. This function determines all primes p such that M_p is not trivial (i.e. zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A contains m copies of $\mathbb{Z}$. In this case, M_p is non-trivial for every prime p. The rank of M_p in this case is the sum of m and the multiplicity of p in S.

`GModulePrimes(G, A, B): GrpFP, GrpFP, GrpFP -> SetMulti`

Let G be a finitely presented group, A a normal subgroup of finite index in G and B a normal subgroup of G contained in A. Given any prime p, the maximal p-elementary abelian quotient of A/B can be viewed as a $\mathbf{F}_p[G]$-module M_p. This function determines all primes p such that M_p is not trivial (i.e. zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A/B is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A/B contains m copies of $\mathbb{Z}$. In this case, M_p is non-trivial for every prime p. The rank of M_p in this case is the sum of m and the multiplicity of p in S.

`GModule(G, A, p): GrpFP, GrpFP, RngIntElt -> ModGrp, Map`

Given a finitely presented group G, a normal subgroup A of finite index in G and a prime p, create the $\mathbf{F}_p[G]$-module M corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A. The function also returns the epimorphism $\pi : A \to M$.

Note that normality of A in G is not checked. The results for invalid input data are undefined.

`GModule(G, A, B, p): GrpFP, GrpFP, GrpFP, RngIntElt -> ModGrp, Map`

`GModule(G, A, B): GrpFP, GrpFP, GrpFP -> ModGrp, Map`

Given a finitely presented group G, a normal subgroup A of G of finite index, a normal subgroup B of G contained in A and a prime p, create the $\mathbf{F}_p[G]$-module M corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A/B.

The integer p can be omitted, if the maximal elementary abelian quotient of A/B is a p-group for some prime p. Note, however, that the computation is much faster, if a prime is specified.

The function also returns the epimorphism $\pi : A \to M$.

Note that normality of A and B in G is not checked. The results for invalid input data are undefined.

`Pullback(f, N): Map, ModGrp -> GrpFP`

Given a map $f : A \to M$ from a normal subgroup A of an fp-group G onto a $\mathbf{F}_p[G]$-module M and a submodule N of M, try to compute the preimage of N under f using a fast pullback method. If successful, the preimage is returned as subgroup of A.

If the pullback works, it is in general faster than a direct computation of the preimage using the preimage operator and it produces a more concise generating set for the preimage; see the following example. In cases where the pullback fails, a runtime error is reported and a preimage construction should be used instead.

Example: KG-Modules of a Finitely Presented Group `(ex-00eade)`

Consider the group G defined by the presentation

$$\langle a,b,c,d,e \mid a^4, b^{42}, c^6, e^3, b^a = b^{-1}, [a,c], [a,d], [a,e], \\ c^b = ce, d^b = d^{-1}, e^b = e^2, d^c = de, e^c = e^2, [d,e]\rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< F | a^4, b^42, c^6, e^3,
>                         b^a=b^-1, (a,c), (a,d), (a,e),
>                         c^b=c*e, d^b=d^-1, e^b=e^2,
>                         d^c=d*e, e^c=e^2,
>                         (d,e) >;
```

The finite index subgroup H of G generated by c, d, e is normal in G.

```
> H := sub< G | c,d,e >;
> Index(G, H);
168
> IsNormal(G, H);
true
```

We check, for which characteristics the action of G on H yields non-trivial modules.

```
> GModulePrimes(G, H);
{* 0, 2, 3 *}
```

We construct the $\mathbf{F}_3[G]$-module M given by the action of G on the maximal 3-elementary abelian quotient of H and the natural epimorphism π from H onto the additive group of M.

```
> M, pi := GModule(G, H, 3);
> M;
GModule M of dimension 2 over GF(3)
```

Using the function `Submodules(M): ModRng → [ModRng]`, we obtain the submodules of M. Their preimages under π are precisely the normal subgroups of G which are contained in H and contain $\ker(\pi)$.

```
> submod := Submodules(M);
> time nsgs := [ m @@ pi : m in submod ];
Time: 11.640
> [ Index(G, s) : s in nsgs ];
[ 1512, 504, 504, 168 ]
```

The generating sets for the normal subgroups obtained in this way, contain in general many redundant generators. (e.g. each will contain a generating set for $\ker(\pi)$.)

```
> [ NumberOfGenerators(s) : s in nsgs ];
[ 19, 20, 20, 21 ]
```

Optimised generating sets can be obtained using the function *ReduceGenerators*.

```
> nsgs_red := [ ReduceGenerators(s) : s in nsgs ];
> [ NumberOfGenerators(s) : s in nsgs_red ];
[ 2, 2, 3, 2 ]
```

Alternatively, and in fact this is the recommended way, we can use the function *Pullback* to compute the preimages of the submodules under π. Note that the generating sets for the preimages computed this way contain fewer redundant generators.

```
> time nsgs := [ Pullback(pi, m) : m in submod ];
Time: 8.560
> [ Index(G, s) : s in nsgs ];
[ 1512, 504, 504, 168 ]
> [ NumberOfGenerators(s) : s in nsgs ];
[ 4, 4, 3, 2 ]
```

Example: Relevant Primes for Module Construction (ex-c74200)

Consider the group defined by the presentation

$$< a,b,c,d,e \mid a^5, b^5, c^6, d^5, e^3, b^a = bd, \\ (a,c),(a,d),(a,e),(b,c),(b,d),(b,e),(c,d),(c,e),(d,e) > .$$

```
> G<a,b,c,d,e> := FPGroup< a,b,c,d,e |
>                          a^5, b^5, c^6, d^5, e^3, b^a = b*d,
>                          (a,c), (a,d), (a,e), (b,c), (b,d), (b,e),
>                          (c,d), (c,e), (d,e) >;
```

Obviously the subgroup of G generated by b, c, d, e is normal in G.

```
> H := sub< G | b,c,d,e >;
> IsNormal(G, H);
true
```

We use the function *GModulePrimes* to determine the set of primes p for which the action of G on the maximal p-elementary abelian quotient of H induces a nontrivial $\mathbf{F}_p[G]$-module.

```
> P := GModulePrimes(G, H);
> 0 in P;
false
```

0 is not contained in P, i.e. the maximal free abelian quotient of H is trivial. Hence, there are only finitely many primes, satisfying the condition above.

We loop over the distinct elements of P and for each element p we construct the induced $\mathbf{F}_p[G]$-module, print its dimension, and check whether it is decomposable. Note that the dimension of the module for p must be equal to the multiplicity of p in P.

```
> for p in MultisetToSet(P) do
>    M := GModule(G, H, p);
>    dim := Dimension(M);
>    decomp := IsDecomposable(M);
>
>    assert dim eq Multiplicity(P, p);
>
>    print "prime", p, ": module of dimension", dim;
>    if decomp then
>       print "  has a nontrivial decomposition";
>     else
>       print "  is indecomposable";
>    end if;
> end for;
prime 2 : module of dimension 1
  is indecomposable
prime 3 : module of dimension 2
  has a nontrivial decomposition
prime 5 : module of dimension 2
  is indecomposable
```

80.13 Some Developed Examples

Example: Finiteness of the Fibonacci Group F(9) (ex-c85753)

The finiteness of the last of the Fibonacci groups, $F(9)$, was settled in 1988 by M.F. Newman using the following result:

Theorem. *Let G be a group with a finite presentation on b generators and r relations, and suppose p is an odd prime. Let d denote the rank of the elementary abelian group $G_1 = [G, G]G^p$ and let e denote the rank of $G_2 = [G_1, G]G^p$. If*

$$r - b < d^2/2 - d/2 - d - e$$

or

$$r - b \leq d^2/2 - d/2 - d - e + (e + d/2 - d^2/4)d/2,$$

then G has arbitrary large quotients of p-power order.

We present a proof that $F(9)$ is infinite using this result.

```
> Left := func< b, r | r - b >;
> Right := func< d, e | d^2 div 2 - d div 2 - d - e +
>                       (e + d div 2 - d^2 div 4)*(d div 2) >;
>
>
> F< x1,x2,x3,x4,x5,x6,x7,x8,x9 > :=
>     FPGroup< x1, x2, x3, x4, x5, x6, x7, x8, x9 |
>              x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6, x5*x6=x7,
>              x6*x7=x8, x7*x8=x9, x8*x9=x1, x9*x1=x2 >;
>
> F;
Finitely presented group F on 9 generators
Relations
       x1 * x2 = x3
       x2 * x3 = x4
       x3 * x4 = x5
       x4 * x5 = x6
       x5 * x6 = x7
       x6 * x7 = x8
       x7 * x8 = x9
       x8 * x9 = x1
       x9 * x1 = x2

> AbelianQuotientInvariants(F);
[ 2, 38 ]
```

Thus the nilpotent quotient of F is divisible by 2 and 19. We examine the 2- and 19-quotients of F.

```
> Q1 := pQuotient(F, 2, 0: Print := 1);
Class limit set to 127.

Lower exponent-2 central series for F

Group: F to lower exponent-2 central class 1 has order 2^2

Group: F to lower exponent-2 central class 2 has order 2^3

Group completed. Lower exponent-2 central class = 2, Order = 2^3
> Q2 := pQuotient(F, 19, 0: Print := 1);
Class limit set to 127.

Lower exponent-19 central series for F

Group: F to lower exponent-19 central class 1 has order 19^1

Group completed. Lower exponent-19 central class = 1, Order = 19^1
```

Thus, the nilpotent residual of F has index 152. We try to locate this subgroup. We first take a 2-generator presentation for F.

```
> G := Simplify(F);
> G;
Finitely presented group G on 2 generators
Generators as words in group F
   G.1 = x4
   G.2 = x5
```

(continues on next page)

(continued from previous page)

```
Relations
   G.2 * G.1 * G.2 * G.1 * G.2^2 * G.1 * G.2^2 * G.1^-1 * G.2 *
      G.1^-2 * G.2 * G.1^-2 = Id(G)
   G.1 * G.2^2 * G.1 * G.2 * G.1^2 * G.2^-1 * G.1^2 * G.2^-1 *
      G.1 * G.2^-1 * G.1^2 * G.2^-1 * G.1 * G.2^-1 = Id(G)
> H := ncl< G | (G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 76 = 2^2 * 19
Subgroup of group G defined by coset table
```

We haven't got the full nilpotent residual yet, so we try again.

```
> H := ncl< G | (G.1*G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 152 = 2^3 * 19
Subgroup of group G defined by coset table
```

Now, we have it.

```
> AbelianQuotientInvariants(H);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
```

The nilpotent residual H has a 5-quotient. We construct a presentation for H and then calculate d and e for its 5-quotient.

```
> K := Rewrite(G, H: Simplify := false);
> KP := pQuotientProcess(K, 5, 1);
> d := FactoredOrder(ExtractGroup(KP))[1][2];
> NextClass(~KP);
> e := FactoredOrder(ExtractGroup(KP))[1][2] - d;
> "D = ", d, "e = ", e;
d =  18 e =  81
> "Right hand side = ", Right(d, e);
Right hand side =  135
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  151
```

Since `Left` is greater than `Right`, this presentation for H doesn't work. Thus we start eliminating generators.

```
> K := Simplify(K: Iterations := 1);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  136
> K := Simplify(K: Iterations := 1);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  123
```

Got it! By Newman's theorem, H has an infinite 5-quotient and so F must be infinite.

Example: Identifying PSL(3, 7):2 from a Presentation (ex-82d19f)

In this example, we consider a – quite unpleasant – presentation for some group G. In fact, it is a presentation for the group $\mathrm{PSL}_3(7)$:2, but we assume that we do not know this and want to compute the order of the finitely presented group G using coset enumeration.

We note in passing that the strategy outlined in this example is, together with other approaches, applied by the function *Order*.

```
> F<a,b,c> := FreeGroup(3);
> rels := [ a^4, b^2, (a,b)^3, (a*b)^8, ((a*b)^2*(a^-1*b)^3)^2,
>  c^2, (c*a*c*a^2)^2, (c*a)^3*(c*a^-1)^3,
>  c*a*b*c*a^-1*b*a^-1*c*a*b*c*a^2*a*b*a^-1,
>  c*a*b*c*b*a*c*a*c*a^-1*b*c*b*a^-1*c*a^-1,
>  c*a*b*a^-1*c*a*b*a^-1*c*a*b*a^-1*c*a*b*a^-1,
>  c*b*a^2*b*c*b*c*a^2*c*b*c*b*a^2*b,
>  c*a^2*c*b*a*c*b*a*c*b*a*c*a^-1*c*a*b*a^-1,
>  c*a^-1*b*a*c*a^-1*b*a*c*b*a*b*a^2*b*a^-1*b,
>  c*a*b*a^-1*b*c*b*a^-1*b*c*a^-1*b*a*b*a*c*b*c*b,
>  c*a*c*b*a*b*c*a*c*b*a*b*c*a*c*b*a*b,
>  c*b*a^-1*b*c*a^-1*c*a^-1*b*a*b*c*b*c*a^2*b*a*b*a^-1,
>  c*b*a^-1*b*a*b*c*b*a^-1*b*a*b*c*b*a^-1*b*a*b,
>  c*a^2*b*a^-1*b*c*b*c*b*a^-1*b*a*c*b*a^2*b*a^-1*b
> ];
> G<a,b,c> := quo< F | rels >;
```

As it happens, trying to determine the order of G by enumerating the cosets of the trivial subgroup is quite hard. – Even the predefined enumeration strategy `"Hard"` (cf. *ToddCoxeter* and *CosetEnumerationProcess*) does not give a finite result.

```
> time ToddCoxeter(G, sub<G|> : Strategy := "Hard");
0
Time: 199.620
```

Of course we could try to increase the workspace for the coset enumeration, but we decide to be more clever. Trying random words in the generators of G, we easily find some subgroup S of G with pretty small index in G.

```
> S := sub< G | b, c*a*c*b*a*b >;
> time Index(G, S);
114
Time: 0.120
```

We now have to compute the order of S. In order to be able to do this using coset enumeration, we have to construct a presentation for S by rewriting S w.r.t. G.

```
> time R := Rewrite(G, S : Simplify := false);
Time: 0.030
```

However, the presentation obtained by Reidemeister-Schreier rewriting without any simplification is not suitable for coset enumeration: it contains too many generators and its total length is quite high.

```
> NumberOfGenerators(R);
133
> PresentationLength(R);
14384
```

An enumeration with the predefined enumeration strategy `"Hard"` does not produce a finite result. (Note that in consideration of the high total relator length, we select a coset table based enumeration style; cf. *CosetEnumerationProcess*.)

```
> time ToddCoxeter(R, sub<R|> : Strategy := "Hard", Style := "C");
0
Time: 4.330
```

On the other hand, simplifying the presentation by reducing the number of generators as much as possible is not a good idea either, since the total relator length grows massively.

```
> time Rs := Simplify(R);
Time: 43.900
> NumberOfGenerators(Rs);
3
> PresentationLength(Rs);
797701
```

Again, an enumeration with the predefined enumeration strategy "Hard" does not produce a finite result.

```
> time ToddCoxeter(Rs, sub<Rs|> : Strategy := "Hard", Style := "C");
0
Time: 22015.849
```

The best strategy is, to simplify the presentation obtained from the Reidemeister-Schreier procedure by eliminating generators until the total length of the relators starts to grow.

```
> time Rsl := SimplifyLength(R);
Time: 0.330
> NumberOfGenerators(Rsl);
48
> PresentationLength(Rsl);
7152
```

A coset enumeration for this presentation produces a finite result in a reasonable amount of time.

```
> time ToddCoxeter(Rsl, sub<Rsl|> : Strategy := "Hard", Style := "C");
32928
Time: 289.410
```

This finally proves that G is finite and has order $32928 * 114 = 3753792$.

80.14 Bibliography

- [AR84] D. G. Arrell and E. F. Robertson. A modified Todd-Coxeter algorithm. In *Computational group theory (Durham, 1982)*, pages 27–32. Academic Press, London, 1984.
- [CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.
- [CHN11] M. Conder, G. Havas, and M. Newman. On one-relator quotients of the modular group. In *Proc. Groups St Andrews 2009 in Bath*, number 387 in London Mathematical Society Lecture Note Series, 183–197. Cambridge University Press, 2011.
- [Fab09] Anna Fabianska. *Algorithmic analysis of presentations of groups and modules.* Dissertation, RWTH Aachen University, 2009.
- [Hav91] G. Havas. Coset enumeration strategies. In *ISSAC'91*, 191–199. ACM Press, 1991.
- [HH10] G. Havas and D.F. Holt. On Coxeter's families of group presentations. *J. Algebra*, 324(5):1076–1082, 2010.
- [HKRR84] George Havas, P. E. Kenne, J. S. Richardson, and E. F. Robertson. A Tietze transformation program. In *Computational group theory (Durham, 1982)*, pages 69–73. Academic Press, London, 1984.
- [HEO05] D. F. Holt, B. Eick, and E. O`Brien. *Handbook of Computational Group Theory.* Chapman and Hall/CRC, 2005.
- [Jam12] Sebastian Jambor. *An L3-U3-quotient algorithm for finitely presented groups.* PhD Thesis, RWTH Aachen University, 2012.

- [Jam15] Sebastian Jambor. An L_2-quotient algorithm for finitely presented groups on arbitrarily many generators. *J. Algebra*, 423:1109–1142, 2015. URL: https://doi.org/10.1016/j.jalgebra.2014.08.058, doi:10.1016/j.jalgebra.2014.08.058.
- [MKS76] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial group theory.* Dover Publications Inc., New York, revised edition, 1976. Presentations of groups in terms of generators and relations.
- [NO96] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.
- [Nic96] Werner Nickel. Computing nilpotent quotients of finitely presented groups. In *Geometric and computational perspectives on infinite groups (Minneapolis, MN and New Brunswick, NJ, 1994)*, pages 175–191. Amer. Math. Soc., Providence, RI, 1996.
- [PF09] W. Plesken and A. Fabianska. An L2-quotient algorithm for finitely presented groups. *J. Algebra*, 322(3):914–935, 2009.
- [Ram] Colin Ramsay. ACE. https://staff.itee.uq.edu.au/havas/.
- [Sim94] Charles C. Sims. *Computation with Finitely Presented Groups.* Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

Chapter 81

Polycyclic Groups

81.1 Introduction

In this chapter, we consider a class of finitely presented groups for which the word problem is solvable, the category of – possibly infinite – polycyclic groups. The corresponding Magma category is called `GrpGPC`. To distinguish this class from finite solvable groups described by a power-conjugate presentation (Magma category `GrpPC`, cf. Chapter Finite Soluble Groups), we use the term *general polycyclic group*.

An introduction to the theory of polycyclic groups and a collection of some basic algorithms can be found in [Sim94, ch. 9]. Unless otherwise mentioned, implementations of Magma functions are mostly based on ideas described in this reference.

81.2 Polycyclic Groups and Polycyclic Presentations

81.2.1 Introduction

A polycyclic group is a group G with a subnormal series $G = G_1 \triangleright G_2 \triangleright \ldots \triangleright G_{n+1} = 1$ in which each of the quotients G_i/G_{i+1} is cyclic. Every polycyclic group G has a presentation of the form

$$\begin{aligned} < a_1, \ldots, a_n \mid\ & a_i^{m_i} = w_{i,i} \quad (i \in I), \\ & a_j^{a_i} = w_{i,j} \quad (1 \leq i < j \leq n), \\ & a_j^{a_i^{-1}} = w_{-i,j} \quad (1 \leq i < j \leq n, i \notin I) > \end{aligned}$$

where

(i) $I \subseteq \{1, \ldots, n\}$,

(ii) $m_i > 1$ for $i \in I$, and

(iii) the words $w_{i,j}$ are of the form $w_{i,j} = a_{|i|+1}^{l(i,j,|i|+1)} \ldots a_n^{l(i,j,n)}$, with $0 \leq l(i,j,k) < m_k$ if $k \in I$.

Such a presentation is called a *polycyclic presentation* for G. For $1 \leq i \leq n$, let G_i be the subgroup of G generated by $a_i, \ldots, a_n$ and define G_{n+1} to be the trivial group. The presentation is called *consistent*, if $|G_i/G_{i+1}| = m_i$ whenever $i \in I$ and G_i/G_{i+1} is infinite whenever $i \notin I$. The generators $a_1, \ldots a_n$ are referred to as *polycyclic generators* (*pc-generators*) for G and the values m_i $(i \in I)$ are called the corresponding *pc-exponents*.

In Magma, the user can define a polycyclic group by providing a consistent polycyclic presentation or by using one of the existing category transfer functions.

Given a consistent polycyclic presentation for G, every element a of G can be written uniquely in the form $a = a_1^{e_1} \ldots a_n^{e_n}$, where the e_i are integers satisfying $0 \leq e_i < m_i$ if $i \in I$. This form is called *normal form*. There exists an algorithm (the *collection algorithm*), which given an arbitrary word in the generators $a_1, \ldots, a_n$, will determine the corresponding normal word. Magma uses a collection algorithm

written by Volker Gebhardt. The cost of collection for this algorithm grows logarithmically in a bound on the absolute values of the exponents e_i occurring during the collection [Geb02].

Infinite polycyclic groups are a comparatively new topic in computational group theory and the number of available algorithms is much smaller than in the case of finite polycyclic groups. For this reason, the data type `GrpPC` (cf. Chapter Finite Soluble Groups) should be preferred for finite polycyclic groups.

81.2.2 Specification of Elements

Elements of polycyclic groups are words. A *word* is defined inductively as follows:

(i) A generator is a word;

(ii) The expression (u) is a word, where u is a word;

(iii) The product $u * v$ of the words u and v is a word;

(iv) The conjugate u^v of the word u by the word v is a word (u^v expands into the word $v^{-1} * u * v$);

(v) The power of a word u^n, where u is a word and n is an integer, is a word;

(vi) The commutator (u, v) of the words u and v is a word ((u, v) expands into the word $u^{-1} * v^{-1} * u * v$).

`G ! Q: GrpGPC, [RngIntElt] -> GrpGPCElt`

Given the polycyclic group G and a sequence Q of length n, containing integers $e_1 \dots e_n$, where $0 \leq e_i < m_i$ if $i \in I$, construct the element x of G given by

$$x = a_1^{e_1} \dots a_n^{e_n}.$$

`Identity(G): GrpGPC -> GrpGPCElt`

`Id(G): GrpGPC -> GrpGPCElt`

`G ! 1: GrpGPC, RngIntElt -> GrpGPCElt`

Construct the identity element of the polycyclic group G.

81.2.3 Access Functions for Elements

Throughout this subsection, G will be a polycyclic group with pc-generators $a_1, \dots, a_n$.

`ElementToSequence(x): GrpGPCElt -> [RngIntElt]`

`Eltseq(x): GrpGPCElt -> [RngIntElt]`

Given an element x belonging to the polycyclic group G, where $x = a_1^{e_1} \dots a_n^{e_n}$ in normal form, return the sequence Q of n integers defined by $Q[i] = e_i$, for $i = 1, \dots, n$.

`LeadingTerm(x): GrpGPCElt -> GrpGPCElt`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return $a_i^{e_i}$ for the smallest i such that $e_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingGenerator(x): GrpGPCElt -> GrpGPCElt`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return a_i for the smallest i such that $e_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingExponent(x): GrpGPCElt -> RngIntElt`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return e_i for the smallest i such that $e_i > 0$. If x is the identity of G, then 0 is returned.

`Depth(x): GrpGPCElt -> RngIntElt`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return the smallest i such that $e_i > 0$. If x is the identity of G, then $n + 1$ is returned.

`Depth` returns the maximal value of i, such that $x \in G_i$.

81.2.4 Arithmetic Operations on Elements

Throughout this subsection, G will be a polycyclic group with pc-generators $a_1, \ldots, a_n$.

`g * h: GrpGPCElt, GrpGPCElt -> GrpGPCElt`

Product of the element g and the element h, where g and h belong to some common subgroup G of a polycyclic group U. If g and h are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h are given as elements belonging to distinct subgroups H and K of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain both elements.

`g *:= h: GrpGPCElt, GrpGPCElt`

Replace g with the product of element g and element h.

`g ^ n: GrpGPCElt, RngIntElt -> GrpGPCElt`

The n-th power of the element g, where n is a positive or negative integer.

`g ^:= n: GrpGPCElt, RngIntElt`

Replace g with the n-th power of the element g.

`g / h: GrpGPCElt, GrpGPCElt -> GrpGPCElt`

Quotient of the element g by the element h, i.e. the element $g * h^{-1}$. Here g and h must belong to some common subgroup G of a polycyclic group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g /:= h: GrpGPCElt, GrpGPCElt`

Replace g with $g * h^{-1}$.

`g ^ h: GrpGPCElt, GrpGPCElt -> GrpGPCElt`

Conjugate of the element g by the element h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to some common subgroup G of a polycyclic group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`g ^:= h: GrpGPCElt, GrpGPCElt`

Replace g with the conjugate of the element g by the element h.

`(g`$_1$`, ..., g`$_n$`): List(GrpGPCElt) -> GrpGPCElt`

Given the n words $g_1, \ldots, g_n$ belonging to some common subgroup G of a polycyclic group U, compute the *left-normed* commutator of $g_1, \ldots, g_n$. This is defined inductively as follows: $(g_1, g_2) = g_1^{-1} * g_2^{-1} * g_1 * g_2$ and $(g_1, \ldots, g_n) = ((g_1, \ldots, g_{n-1}), g_n)$.

If $g_1, \ldots, g_n$ are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if $g_1, \ldots, g_n$ are given as elements belonging to distinct subgroups of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain all elements.

81.2.5 Operators for Elements

`Order(x): GrpGPCElt -> RngIntElt`

The order of the element x.

`IsFinite(x): GrpGPCElt -> BoolElt`

Returns `true` if the order of the element x is finite, `false` otherwise.

`Parent(x): GrpGPCElt -> GrpGPC`

The parent group G of the element x.

81.2.6 Comparison Operators for Elements

`g eq h: GrpGPCElt, GrpGPCElt -> BoolElt`

Given elements g and h belonging to a common polycyclic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h: GrpGPCElt, GrpGPCElt -> BoolElt`

Given elements g and h belonging to a common polycyclic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsIdentity(g): GrpGPCElt -> BoolElt`

`IsId(g): GrpGPCElt -> BoolElt`

Returns `true` if g is the identity element, `false` otherwise.

81.2.7 Specification of a Polycyclic Presentation

`quo< GrpGPC : F | R : parameters >: GrpFP, List(GrpFPRel) -> GrpGPC, Map`

Check: BoolElt	Default: true

Given a free group F of rank n with generating set X, and a collection R of polycyclic relations on X, construct the polycyclic group G defined by the polycyclic presentation $< X|R >$. The construct R denotes a list of polycyclic relations. Thus, an element of R must be one of:

(a) A *power relation* $a_i^{m_i} = w_{i,i}$, $i \in I \subseteq \{1, \ldots, n\}$, where $m_i > 1$ is an integer, and $w_{i,i}$ is $Id(F)$ or a word in the generators $a_{i+1}, \ldots, a_n$ for $i < n$, and $w_{i,i} = Id(F)$ for $i = n$.

(b) A *conjugate relation* $a_j^{a_i^e} = w_{e \cdot i,j}$, $1 \leq i < j \leq n$, where $w_{e \cdot i,j}$ is a word in the generators $a_{i+1}, \ldots, a_n$, $e = 1$ if $i \in I$ and $e \in \{-1, 1\}$ if $i \notin I$.

(c) A *power* $a_i^{m_i}$, $i \in I \subseteq \{1, \ldots, n\}$, where $m_i > 1$ is an integer, which is treated as the *power relation* $a_i^{m_i} = Id(F)$.

(d) A set of (a) – (c).

(e) A sequence of (a) – (c).

Note the following points:

(i) If there is no power relation for some generator a_i, $i = 1, \ldots, n$ (i.e. $i \notin I$), conjugate relations $a_j^{a_i^{-1}} = w_{-i,j}$ must be present for $i < j \leq n$ (except for pairs of commuting generators). If there is a power relation for a_i, on the other hand, conjugate relations involving a_i^{-1} are not permitted.

(ii) Conjugate relations for pairs of commuting generators may be omitted. If no conjugate relations are given for a certain pair of generators, the corresponding generators are assumed to commute.

(iii) The words $w_{i,j}$ must be in normal form.

(iv) The presentation must be consistent. In particular, right hand sides of conjugate relations must not be the empty word.

This constructor returns a polycyclic group because the category `GrpGPC` is stated. If no category were stated, it would return an fp-group. The parameter *Check* may be used to indicate whether or not the presentation should be checked for consistency. Disabling this check can be useful for avoiding runtime errors if the constructor is called in user written loops or functions. The boolean valued function *IsConsistent* is provided to check presentations obtained with disabled consistency check. It should be noted that the results of working with an inconsistent presentation are undefined, hence it is strongly advised to enable the consistency check in the constructor or to use the function *IsConsistent*. The natural homomorphism from $F \to G$ is returned as second return value.

PolycyclicGroup< x_1, ..., x_n | R : parameters >: List(Identifiers), List(GrpFPRel) -> GrpGPC, Map

Check: BoolElt	Default: true
Class: MonStgElt	Default:

Construct the polycyclic group G defined by the consistent polycyclic presentation $< x_1, \ldots, x_n | R >$.

The construct $x_1, \ldots, x_n$ defines names for the generators of G that are local to the constructor, i.e. they are used when writing down the relations to the right of the bar. However, no assignment of names to these generators is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The construct R denotes a list of polycyclic relations. The syntax and semantics for the relations clause is identical to that appearing in the *quo< GrpGPC : F | R : parameters >*-constructor above.

A map f from the free group on $x_1, \ldots, x_n$ to G is returned as second return value.

The parameter **Check** may be used as described for the *quo< GrpGPC : F | R : parameters >*-constructor.

If R is both, a valid power-conjugate presentation for a finite soluble group (cf. Chapter Finite Soluble Groups) and a consistent polycyclic presentation, this constructor by default returns a group in the category GrpPC. To force creation of a group in the category **GrpGPC**, the parameter **Class** must be set to **"GrpGPC"** in these situations.

Example: Specifying a Polycyclic Presentation (ex-600748)

Consider the infinite polycyclic group defined by the presentation

$$< a, b, c \mid b^a = b * c, (a, c), (b, c) > .$$

Starting from a free group and giving the relations in the form of a sequence, this presentation would be specified as follows:

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a = b*c, b^(a^-1) = b*c^-1 ];
> G<a,b,c> := quo< GrpGPC : F | rels >;
> G;
GrpGPC : G of infinite order on 3 PC-generators
PC-Relations:
    b^a = b * c,
    b^(a^-1) = b * c^-1
```

Note, that the relation $b^{a^{-1}} = b * c^{-1}$ has to be included, although it can be derived from the relations $b^a = b * c$ and (a, b).

The infinite dihedral group is obtained as epimorphic image of the free group of rank two as follows:

```
> F<a,b> := FreeGroup(2);
> D<u,v>, pi := quo<GrpGPC: F | a^2, b^a = b^-1>;
> D;
GrpGPC : D of infinite order on 2 PC-generators
PC-Relations:
    u^2 = Id(D),
    v^u = v^-1
> pi;
Mapping from: GrpFP: F to GrpGPC: D
```

We create an element e of the group D defined above from a sequence of coefficients and extract both its leading generator and its leading exponent.

```
> e := D ! [1,42];
> e;
u * v^42
> gen := LeadingGenerator(e);
> gen;
u
> Parent(gen);
GrpGPC : D of infinite order on 2 PC-generators
PC-Relations:
    u^2 = Id(D),
    v^u = v^-1
> exp := LeadingExponent(e);
> exp;
1
```

We obtain an element of depth 2 from e, by replacing e with its quotient by the appropriate power of the leading generator.

```
> e /:= gen^exp;
> Depth(e);
2
> e;
v^-42
> ElementToSequence(e);
[ 0, -42 ]
```

Example: Dihedral Group in Different Categories (ex-a6668b)

Using the constructor *PolycyclicGroup< x_1, ..., x_n | R : parameters >* with different values of the parameter `Class`, we construct the dihedral group of order 10 first as a finite soluble group given by a power-conjugate presentation (`GrpPC`) and next as a general polycyclic group (`GrpGPC`). Note that the presentation $\langle a, b \,|\, a^2, b^5, b^a = b^4 \rangle$ is both a valid power-conjugate presentation and a consistent polycyclic presentation, so we have to set the parameter `Class` to `"GrpGPC"` if we want to construct a group in the category `GrpGPC`.

```
> G1<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 >;
> G1;
GrpPC : G1 of order 10 = 2 * 5
PC-Relations:
    a^2 = Id(G1),
    b^5 = Id(G1),
    b^a = b^4
> G2<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 : Class := "GrpGPC">;
> G2;
GrpGPC : G2 of order 10 = 2 * 5 on 2 PC-generators
PC-Relations:
    a^2 = Id(G2),
    b^5 = Id(G2),
    b^a = b^4
```

We construct the infinite dihedral group as a group in the category `GrpGPC` from a consistent polycyclic presentation. We do not have to use the parameter `Class` in this case.

```
> G3<a,b> := PolycyclicGroup< a,b | a^2, b^a=b^-1>;
> G3;
```

(continues on next page)

(continued from previous page)

```
GrpGPC : G3 of infinite order on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^a = b^-1
```

The presentation $\langle a, b \mid a^2, b^4, b^a = b^3 \rangle$ is not a valid power-conjugate presentation for the dihedral group of order 8, since the exponent of b is not prime. However, it is a consistent polycyclic presentation. Consequently, the constructor `PolycyclicGroup` without specifying a value for the parameter `Class` returns a group in the category `GrpGPC`.

```
> G4<a,b> := PolycyclicGroup< a,b | a^2, b^4, b^a=b^3 >;
> G4;
GrpGPC : G4 of order 2^3 on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^4 = Id(G3),
    b^a = b^3
```

81.2.8 Properties of a Polycyclic Presentation

`IsConsistent(G): GrpGPC -> BoolElt`

Returns `true` if the stored presentation for G is consistent, `false` otherwise.

`IsIdenticalPresentation(G, H): GrpGPC, GrpGPC -> BoolElt`

Returns `true` if the polycyclic presentations for G and H are identical, `false` otherwise.

`PresentationIsSmall(G): GrpGPC -> BoolElt`

Returns `true` if only small integers occur in the presentation of G, `false` otherwise. The category transfer functions *FPGroup* and *PCGroup* currently support only groups with a small presentation.

81.3 Subgroups, Quotient Groups, Homomorphisms, and Extensions

81.3.1 Construction of Subgroups

`sub<G | L>: GrpGPC, List -> GrpGPC, Map`

Construct the subgroup H of the polycyclic group G generated by the elements specified by the terms of the *generator list* L.

A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable to G;

(b) A sequence of integers representing an element of G;

(c) A subgroup of G;

(d) A set or sequence of (a), (b), or (c).

The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

The generators of H consist of the words specified directly by terms of $L[i]$ together with the stored generating words for any groups specified by terms of $L[i]$. Repetitions of an element and occurrences of the identity element are removed.

The inclusion map from H to G is returned as a second value.

`ncl<G | L>: GrpGPC, List -> GrpGPC, Map`

Construct the subgroup N of the polycyclic group G as the normal closure of the subgroup generated by the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the *sub<G | L>*-constructor.

The inclusion map from N to G is returned as a second value.

81.3.2 Coercions Between Groups and Subgroups

`G ! g: GrpGPC, GrpGPCElt -> GrpGPCElt`

Given an element g belonging to the subgroup H of the group G, rewrite g as an element of G.

`H ! g: GrpGPC, GrpGPCElt -> GrpGPCElt`

Given an element g belonging to the group G, and given a subgroup H of G containing g, rewrite g as an element of H.

`K ! g: GrpGPC, GrpGPCElt -> GrpGPCElt`

Given an element g belonging to the group H, and a group K, such that H and K are subgroups of G, and both H and K contain g, rewrite g as an element of K.

`InclusionMap(G, H): GrpGPC, GrpGPC -> Map`

The map from the subgroup H of G to G.

Example: Coercion into a Subgroup (ex-49b53c)

Consider again the infinite polycyclic group G defined by the polycyclic presentation

$$< a, b, c \mid b^a = b * c, (a, c), (b, c) > .$$

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a = b*c, b^(a^-1) = b*c^-1 ];
> G<a,b,c> := quo< GrpGPC : F | rels >;
> G;
GrpGPC : G of infinite order on 3 PC-generators
PC-Relations:
    b^a = b * c,
    b^(a^-1) = b * c^-1
```

Using the function *PCGenerators* which is described later, the groups $G_1, \ldots, G_4$ and the corresponding inclusion maps can be defined as follows:

```
> G_ := []; incl_ := [ PowerStructure(Map) | ];
> for i := 1 to #PCGenerators(G)+1 do
>   G_[i], incl_[i] := sub< G | [ g : g in PCGenerators(G) |
>                                       Depth(g) ge i ] >;
> end for;
> for i := 1 to #G_ do
>   printf "G_%o = <%o>", i, {@ G!x : x in
>                     PCGenerators(G_[i]) @}; print "";
> end for;
G_1 = <{@ a, b, c @}>
G_2 = <{@ b, c @}>
G_3 = <{@ c @}>
G_4 = <{@ @}>
```

Note that we must set the universe of the sequence `incl_` to `PowerStructure(Map)` manually, since we want to store maps which do not have a common domain. If we failed to do this, the universe would be chosen automatically to be the set of all maps from G_1 to G when the first map

is inserted into the sequence. Inserting the second map, which does not have the domain G_1, would then cause a runtime error.

81.3.3 Construction of Quotient Groups

`quo<G | L>: GrpGPC, List -> GrpGPC, Map`

Construct the quotient Q of the polycyclic group G by the normal subgroup N, where N is the smallest normal subgroup of G containing the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the *sub<G | L>*-constructor.

The quotient group Q and the corresponding natural homomorphism $f : G \to Q$ are returned.

`G / N: GrpGPC, GrpGPC -> GrpGPC`

Given a normal subgroup N of the polycyclic group G, construct the quotient of G by N.

81.3.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter Mappings. This section describes some special aspects of homomorphisms the domain of which is a polycyclic group.

General remarks

The kernel of a homomorphism with a domain of type `GrpGPC` can be computed using the function `Kernel(f): Map -> Str`, if the codomain is of one of the types `GrpGPC`, `GrpPC` (cf. Chapter Finite Soluble Groups), `GrpAb` (cf. Chapter *Abelian Groups*), `GrpPerm` (cf. Chapter Permutation Groups), `ModAlg`, or `ModGrp` (cf. Chapter Modules over an Algebra and Group Representations) or if the codomain is of the type `GrpMat` (cf. Chapter Matrix Groups over General Rings) and the image is finite.

In particular, preimages of substructures can be computed in these situations. The kernel of a map will be computed automatically, if the preimage of a substructure of the codomain is to be computed.

The kernel (and hence preimages of substructures) may also be computable, if the codomain is of the type `GrpFP` (cf. Chapter *Finitely Presented Groups*) and the domain is nilpotent.

Construction of Homomorphisms

`hom< P -> G | S : parameters>: Struct, Struct -> Map`

Check: BoolElt	Default: true

Returns the homomorphism from the polycyclic group P to the group G defined by the assignment S. S can be the one of the following:

(i) A list, sequence, or indexed set containing the images of the n polycyclic generators $P.1, \ldots, P.n$ of P. Here, the i-th element of S is interpreted as the image of $P.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set, or indexed set, containing r tuples $< x_i, y_i >$ or arrow pairs x_i `->` y_i, where $x_i \in P$, $y_i \in G$ $(i = 1, \ldots, r)$ and the set $\{x_1, \ldots, x_r\}$ is a generating set for P. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important. Note that the preimages x_i need not be the polycyclic generators of P.

If the data type of the codomain supports element arithmetic and element comparison, by default the constructed homomorphism is checked by verifying that the would-be images of the polycyclic generators satisfy the defining relations of P and that this assignment is consistent with the assignments made by the user. In this case, it is assured that the returned map is a well-defined homomorphism with the desired images. The most important situation in which it is not possible to perform checking is the case in which the domain is a finitely presented group (`FPGroup`;

cf. Chapter *Finitely Presented Groups*) which is not free. Checking may be disabled using the parameter `Check`.

If `Check` is set to `false`, checking of the homomorphism is disabled.

81.3.5 Construction of Extensions

`DirectProduct(G, H): GrpGPC, GrpGPC -> GrpGPC, [Map], [Map]`

The direct product K of the polycyclic groups G and H. The second value returned is a sequence containing the inclusion maps $I_G\colon G \to K$ and $I_H\colon H \to K$. The third value returned is a sequence containing the projection maps $P_G\colon K \to G$ and $P_H\colon K \to H$.

81.3.6 Construction of Standard Groups

A number of functions are provided which construct polycyclic presentations for various standard groups.

`AbelianGroup(GrpGPC, Q): Cat, [RngIntElt] -> GrpGPC`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ as a polycyclic group. The entries n_i may be either zero, indicating an infinite cyclic factor, or integers greater than 1. The function returns the polycyclic group which is the direct product of the cyclic groups $Z_1 \times \cdots \times Z_r$, where Z_i is a cyclic group of infinite order if $n_i = 0$, or a cyclic group of order n_i if $n_i > 1$.

`CyclicGroup(GrpGPC, n): Cat, RngIntElt -> GrpGPC`

For $n > 0$, the cyclic group of order n is returned, while for $n = 0$, the infinite cyclic group is returned as a polycyclic group.

`DihedralGroup(GrpGPC, n): Cat, RngIntElt -> GrpGPC`

For $n \geq 3$, the dihedral group of order $2n$ is returned, while for $n = 0$, the infinite dihedral group is returned as a polycyclic group.

`ElementaryAbelianGroup(GrpGPC, p, n): Cat, RngIntElt, RngIntElt -> GrpGPC`

Given a prime p and a positive integer n, construct the elementary abelian group of order p^n as a polycyclic group.

`ExtraSpecialGroup(GrpGPC, p, n : parameters): Cat, RngIntElt, RngIntElt -> GrpGPC`

Type: MonStgElt	Default: "+"

Given a prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} as a polycyclic group. The isomorphism type of G can be selected using the parameter `Type`.

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

`FreeAbelianGroup(GrpGPC, n): Cat, RngIntElt -> GrpGPC`

Given a positive integer n, construct the free abelian group of rank n as a polycyclic group.

`FreeNilpotentGroup(r, e): RngIntElt, RngIntElt -> GrpGPC`

The free nilpotent group of rank r and class e is returned as a polycyclic group.

Example: Homomorphism of D4 onto Z2 x Z2 `(ex-5a225f)`

Consider the dihedral group D_4.

```
> G<a,b> := DihedralGroup(GrpGPC, 4);
```

We obtain a well-defined homomorphism f of G onto a group $A = \mathbf{Z}_2 \times \mathbf{Z}_2$, by assigning two generators of A as the images of a and ab, respectively. Note that in the definition of f we can use the generating set $\{a, ab\}$ of G, instead of the defining polycyclic generating sequence of G.

```
> A<u,v> := AbelianGroup([2,2]);
> f := hom< G->A | a->u, a*b->v >;
```

We compute the kernel K of f and express the generators of K as elements of G, using the function *PCGenerators* described later.

```
> K := Kernel(f);
> PCGenerators(K, G);
{@ b^2 @}
```

Example: Direct Product with the Infinite Dihedral Group (ex-f4c433)

A polycyclic representation for the group $D_3 \times D_\infty$ may be obtained as follows:

```
> G1<a,b> := DihedralGroup(GrpGPC, 3);
> G2<u,v> := DihedralGroup(GrpGPC, 0);
> D, incl, proj := DirectProduct(G1, G2);
> D;
GrpGPC : D of infinite order on 4 PC-generators
PC-Relations:
    D.1^2 = Id(D),
    D.2^3 = Id(D),
    D.3^2 = Id(D),
    D.2^D.1 = D.2^2,
    D.4^D.3 = D.-4
```

Using the inclusion maps returned by *DirectProduct*, we define a subgroup and compute the quotient by its normal closure:

```
> S := sub<D| incl[1](a)*incl[2](u), incl[1](b)*incl[2](v)>;
> S;
GrpGPC : S of infinite order on 3 PC-generators
PC-Relations:
    S.1^2 = Id(S),
    S.2^3 = S.3,
    S.2^S.1 = S.2^2 * S.-3,
    S.3^S.1 = S.-3
> Q, pi := quo<D|S>;
> Q;
GrpGPC : Q of order 2 on 1 PC-generators
PC-Relations:
    Q.1^2 = Id(Q)
> Q.1 @@ pi;
D.3
```

81.4 Conversion between Categories

This section describes category transfers from polycyclic groups to other Magma categories of groups. For category transfers to permutation groups (cf. Chapter Permutation Groups) the functions *CosetAction* and *CosetImage* can be used. For category transfers to matrix groups (cf. Chapter Matrix Groups over General Rings) we refer to Section *Representation Theory*.

AbelianGroup(G): GrpGPC -> GrpAb, Map

A GrpAb (cf. Chapter *Abelian Groups*) representation A of the abelian polycyclic group G and the isomorphism from G to A.

FPGroup(G): GrpGPC -> GrpFP, Map

A GrpFP (cf. Chapter *Finitely Presented Groups*) representation F of G and the isomorphism from F to G. This category transfer is currently only possible provided that all exponents occurring in the presentation of G are small integers. This can be checked by means of the function *PresentationIsSmall*.

PCGroup(G): GrpGPC -> GrpPC, Map

A GrpPC (cf. Chapter Finite Soluble Groups) representation F of the finite polycyclic group G and the isomorphism from G to F.

This category transfer is currently only possible provided that all exponents occurring in the presentation of G are small integers. This can be checked by means of the function *PresentationIsSmall*.

GPCGroup(G): GrpPerm -> GrpGPC, Map

GPCGroup(G): GrpMat -> GrpGPC, Map

GPCGroup(G): GrpAb -> GrpGPC, Map

GPCGroup(G): GrpPC -> GrpGPC, Map

A GrpGPC representation P of the solvable group G and the isomorphism from G to P. The group G can be of one of the following types: GrpPerm (cf. Chapter Permutation Groups), GrpMat (cf. Chapter Matrix Groups over General Rings), GrpAb (cf. Chapter *Abelian Groups*), GrpPC (cf. Chapter Finite Soluble Groups). Currently G must be finite, if it is of type GrpMat.

Example: Transferring Subgroups and Quotients (ex-f0ed02)

We define a finite, solvable matrix group and convert it to a (general) polycyclic group G.

```
> a := GL(2,3) ! [1,1,0,1];
> b := GL(2,3) ! [0,1,1,0];
> M := sub<Parent(a)|a,b>;
> IsSolvable(M);
true
> IsFinite(M);
true
> G, f := GPCGroup(M);
```

We now compute the direct product D of G and the infinite dihedral group H, define a subgroup S of D, its normal closure N, and construct the quotient Q of D by N.

```
> H<u,v> := DihedralGroup(GrpGPC, 0);
> D, incl, proj := DirectProduct(G, H);
> S := sub<D | incl[1](f(a*b)), incl[2]((u,v)^2)>;
> N := ncl<D|S>;
> Q := D/N;
> Q;
GrpGPC : Q of order 2^3 on 2 PC-generators
PC-Relations:
    Q.1^2 = Id(Q),
    Q.2^4 = Id(Q),
    Q.2^Q.1 = Q.2^3
```

Since Q is finite, it can be transformed into a group of type GrpPC using the function *PCGroup*. This should (in non-trivial examples) be done, if further computations with it are intended.

```
> Q_ := PCGroup(Q);
> Q_;
```

(continues on next page)

(continued from previous page)

```
GrpPC : Q_ of order 8 = 2^3
PC-Relations:
    Q_.2^2 = Q_.3,
    Q_.2^Q_.1 = Q_.2 * Q_.3
```

81.5 Access Functions for Groups

The functions described here provide access to basic information stored for a polycyclic group G.

`G . i: GrpGPC, RngIntElt -> GrpGPCElt`

The i-th polycyclic generator for G if $i > 0$, the inverse of the $|i|$-th polycyclic generator for G if $i < 0$ and the identity element of G if $i = 0$.

`Generators(G): GrpGPC -> {@ GrpGPCElt @}`

`PCGenerators(G): GrpGPC -> {@ GrpGPCElt @}`

An indexed set containing the polycyclic generators of G.

`Generators(H, G): GrpGPC, GrpGPC -> {@ GrpGPCElt @}`

`PCGenerators(H, G): GrpGPC, GrpGPC -> {@ GrpGPCElt @}`

An indexed set containing the polycyclic generators of H as elements of G.

`NumberOfGenerators(G): GrpGPC -> RngIntElt`

`Ngens(G): GrpGPC -> RngIntElt`

`NumberOfPCGenerators(G): GrpGPC -> RngIntElt`

`NPCgens(G): GrpGPC -> RngIntElt`

`NPCGenerators(G): GrpGPC -> RngIntElt`

The number of polycyclic generators for the polycyclic group G.

`PCExponents(G): GrpGPC -> [RngIntElt]`

The orders of the cyclic factors in the polycyclic series defined by the polycyclic presentation of G. The orders are returned in a sequence Q. $|G_i/G_{i+1}| = m_i = Q[i]$ if $Q[i] > 0$ and G_i/G_{i+1} is infinite (i.e. $i \notin I$) if $Q[i] = 0$.

`HirschNumber(G): GrpGPC -> RngIntElt`

The Hirsch number of G, i.e. the number of infinite cyclic factors in the polycyclic series defined by the polycyclic presentation of G.

The Hirsch number of G is equal to $n - |I|$, i.e. to the number of polycyclic generators of G for which there is no power relation. A polycyclic group G is finite if and only if its Hirsch number is 0.

81.6 Set-Theoretic Operations in a Group

81.6.1 Functions Relating to Group Order

`FactoredIndex(G, H): GrpGPC, GrpGPC -> [<RngIntElt, RngIntElt>]`

Given a group G and a subgroup H of G of finite index, return the factored index of H in G

`FactoredOrder(G): GrpGPC -> [<RngIntElt, RngIntElt>]`

The factored order of the finite group G.

`Index(G, H): GrpGPC, GrpGPC -> RngIntElt`

The index of the subgroup H in the group G, returned as an ordinary integer.

`Order(G): GrpGPC -> RngIntElt`

`# G: GrpGPC -> RngIntElt`

The order of the group G, returned as an ordinary integer.

81.6.2 Membership and Equality

`g in G: GrpGPCElt, GrpGPC -> BoolElt`

Given an element g and a group G, return `true` if g is an element of G, `false` otherwise.

`g notin G: GrpGPCElt, GrpGPC -> BoolElt`

Given an element g and a group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G: { GrpGPCElt}, GrpGPC -> BoolElt`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G: { GrpGPCElt}, GrpGPC -> BoolElt`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G: GrpGPC, GrpGPC -> BoolElt`

`IsSubgroup(H, G): GrpGPC, GrpGPC -> BoolElt`

Given groups G and H, having some covering group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G: GrpGPC, GrpGPC -> BoolElt`

Given groups G and H, having some covering group, return `true` if H is not a subgroup of G, `false` otherwise.

`G eq H: GrpGPC, GrpGPC -> BoolElt`

Given groups G and H, having some covering group, return `true` if G and H are the same group, `false` otherwise.

`G ne H: GrpGPC, GrpGPC -> BoolElt`

Given groups G and H, having some covering group, return `true` if G and H are distinct groups, `false` otherwise.

81.6.3 Set Operations

`Representative(G): GrpGPC -> GrpGPCElt`

`Rep(G): GrpGPC -> GrpGPCElt`

A representative element of G.

`RandomProcess(G): GrpGPC -> Process`

```
Slots   : RngIntElt                    Default: 10
Scramble: RngIntElt                    Default: 100
```

Create a process to generate pseudo-randomly chosen elements from the group G. The process uses an 'expansion' procedure to construct a set of elements corresponding to fairly long words in the generators of G [CLGM+95]. At all times, N elements forming a generating set for G are stored. Here, N is the maximum of $n + 1$ and the specified value for *Slots*, where n is the number of generators of G. Initially, these are just the generators of G and products of pairs of generators of G. Random elements are now produced by successive calls to *Random*(P), where P is the process created by this function. Each such call chooses an element previously stored by the process as the new random element. The process then replaces this stored element with the product of this element and another one of the stored elements (on the left or the right). Setting *Scramble*:= m causes m such operations to be performed before the process is returned.

Care should be taken when trying to apply this function to infinite polycyclic groups. Firstly, the computations may take a considerable amount of time and secondly, the quality of the pseudo-random element generator may be extremely poor, depending on the required properties of the sequence of pseudo-random elements.

`Random(P): Process -> GrpGPCElt`

Given a random element process P created by the function *RandomProcess*(G) for the group G, construct a pseudo-random element of G by forming a random product over the expanded generating set currently stored by the process. The remarks concerning random elements of infinite polycyclic groups given in the description of *RandomProcess* apply here.

`Random(G): GrpGPC -> GrpGPCElt`

`Random(G, max): GrpGPC, RngIntElt -> GrpGPCElt`

An element, pseudo-randomly chosen, from the group G. An exponent vector in normal form is chosen at random. Exponents of polycyclic generators for which there is no power relation, are chosen to have absolute value less or equal to `max`. A default value for `max` is 10.

It should be kept in mind that the distribution of the elements returned by `Random` is uniform only in the case that G is finite.

81.7 Coset Spaces

`CosetTable(G, H): GrpGPC, GrpGPC -> Map`

The (right) coset table for G over the subgroup H of finite index, relative to the polycyclic generators. This is defined to be a map

$$\{1, \ldots, |G : H|\} \times G \to \{1, \ldots, |G : H|\}$$

describing the action of G on the enumerated set of right cosets of H in G by right multiplication.

The underlying set of right coset representatives is identical to the right transversal returned by *Transversal* and *RightTransversal* and the same enumeration of the elements is used.

`Transversal(G, H): GrpGPC, GrpGPC -> {@ GrpGPCElt @}, Map`

`RightTransversal(G, H): GrpGPC, GrpGPC -> {@ GrpGPCElt @}, Map`

Given a group G and a subgroup H of G, this function returns:

(a) An indexed set of elements T of G forming a right transversal for G over H. The right transversal and its enumeration are identical to those internally used by the function *CosetTable*.

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and g in G, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

Example: Right Transversal in the Infinite Dihedral Group (ex-fcfe92)

We compute a right transversal of a subgroup H of the infinite dihedral group G.

```
> G<a,b> := DihedralGroup(GrpGPC, 0);
> H := sub<G|a*b, b^10>;
> Index(G, H);
10
> RT, transmap := Transversal(G, H);
> RT;
{@ Id(G), b^-1, b^-2, b^-3, b^-4, b^-5, b^-6, b^-7, b^-8, b^-9 @}
> transmap;
Mapping from: GrpGPC: G to SetIndx: RT
```

From this a left transversal is easily obtained:

```
> LT := {@ x^-1 : x in RT @};
> LT;
{@ Id(G), b, b^2, b^3, b^4, b^5, b^6, b^7, b^8, b^9 @}
```

We construct the coset table and define a function $RT \times G \to RT$, describing the action of G on the set of right cosets of H in G.

```
> ct := CosetTable(G, H);
> action := func< r, g | RT[ct(Index(RT, r), g)] >;
> action(Id(G), b);
b^-9
```

i.e. $H * b = Hb^{-9}$.

```
> action(b^-4, a*b);
b^-6
```

i.e. $Hb^{-4} * (ab) = Hb^{-6}$.

Note that the definition of the function `action` relies on the fact that the computed right transversal and its enumeration are identical to those internally used by the function *CosetTable*.

CosetAction(G, H): GrpGPC, GrpGPC -> Map, GrpPerm, GrpGPC

Given a subgroup H of the group G of finite index, construct the permutation representation of G, induced by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The permutation representation $f : G \to L \leq \mathrm{Sym}(|G:H|)$, induced by the action of G on the set of (right) cosets of H in G;

(b) The epimorphic image L of G under the representation f;

(c) The kernel K of the representation f.

CosetImage(G, H): GrpGPC, GrpGPC -> GrpPerm

Given a subgroup H of the group G of finite index, construct the permutation group, induced by the action of G on the set of (right) cosets of H in G. The returned group is the epimorphic image L of G under the permutation representation $f : G \to L \leq \mathrm{Sym}(|G:H|)$, induced by the action of G on the set of (right) cosets of H in G.

CosetKernel(G, H): GrpGPC, GrpGPC -> GrpGPC

Given a subgroup H of the group G of finite index, construct the kernel of the permutation representation $f : G \to L \leq \mathrm{Sym}(|G:H|)$, induced by the action of G on the set of (right) cosets of H in G.

Example: Coset Action of a Polycyclic Group (ex-bf9acf)

We use the function *CosetAction* to construct a (non-faithful) permutation representation of the group G defined by the polycyclic presentation

$$< a, b, c \mid b^a = b * c, (a, c), (b, c) > .$$

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a=b*c, b^(a^-1)=b*c^-1 ];
> G<a,b,c> := quo<GrpGPC: F | rels>;
>
> S := sub<G|(a*b)^3, c^7, b^21>;
> Index(G, S);
441
> pi, P, K := CosetAction(G, S);
> P;
Permutation group P acting on a set of cardinality 441
> K;
GrpGPC : K of infinite order on 3 PC-generators
PC-Relations:
    K.2^K.1 = K.2 * K.3^63,
    K.2^(K.-1) = K.2 * K.-3^63
```

(continues on next page)

(continued from previous page)

```
> Index(G, K);
3087
```

We express the generators of the kernel K in terms of the generators of G:

```
> {@ G!x : x in PCGenerators(K) @};
{@ a^21, b^21, c^7 @}
```

$pi(S)$ is a point stabiliser in the transitive permutation group P of degree 441 and hence should have index 441 in P:

```
> pi(S);
Permutation group acting on a set of cardinality 441
> Index(P, pi(S));
441
```

81.8 The Subgroup Structure

81.8.1 General Subgroup Constructions

The operators and functions which construct a subgroup of a polycyclic group always return the subgroup as a polycyclic group.

`H ^ g: GrpGPC, GrpGPCElt -> GrpGPC`

`Conjugate``(H, g): GrpGPC, GrpGPCElt -> GrpGPC`

Construct the conjugate $g^{-1} * H * g$ of the group H under the action of the element g. The group H and the element g must belong to a common group.

`H ^ G: GrpGPC, GrpGPC -> GrpGPC`

`ncl<` `G | H >: GrpGPC, GrpGPC -> GrpGPC`

`NormalClosure``(G, H): GrpGPC, GrpGPC -> GrpGPC`

Given a subgroup H of the group G, construct the normal closure of H in G.

`CommutatorSubgroup``(G, H, K): GrpGPC, GrpGPC, GrpGPC -> GrpGPC`

`CommutatorSubgroup``(H, K): GrpGPC, GrpGPC -> GrpGPC`

Construct the commutator subgroup of groups H and K, where H and K are subgroups of a common group G.

81.8.2 Subgroup Constructions Requiring a Nilpotent Covering Group

The operators and functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98]. Again, the constructed subgroup is returned as a polycyclic group.

`H` **`meet`** `K: GrpGPC, GrpGPC -> GrpGPC`

Given two groups H and K, contained in some common group G which is nilpotent, construct the intersection of H and K.

`H` **`meet:=`** `K: GrpGPC, GrpGPC -> GrpGPC`

Given two groups H and K, contained in some common group G which is nilpotent, replace H with the intersection of H and K.

`Centraliser``(G, g): GrpGPC, GrpGPCElt -> GrpGPC`

`Centralizer``(G, g): GrpGPC, GrpGPCElt -> GrpGPC`

The subgroup of G centralising g. Both g and G must be contained in some common nilpotent group.

`Centraliser``(G, H): GrpGPC, GrpGPC -> GrpGPC`

`Centralizer(G, H): GrpGPC, GrpGPC -> GrpGPC`

The subgroup of G centralising H. Both H and G must be subgroups of some common nilpotent group.

`Core(G, H): GrpGPC, GrpGPC -> GrpGPC`

The maximal normal subgroup of the nilpotent group G that is contained in the subgroup H of G.

`Normaliser(G, H): GrpGPC, GrpGPC -> GrpGPC`

`Normalizer(G, H): GrpGPC, GrpGPC -> GrpGPC`

The subgroup of G normalising H. Both H and G must be subgroups of some common nilpotent group.

81.9 General Group Properties

`IsAbelian(G): GrpGPC -> BoolElt`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G): GrpGPC -> BoolElt`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G): GrpGPC -> BoolElt`

Returns `true` if the group G is elementary abelian, `false` otherwise. The following definition is used:

A group G is called elementary abelian if it is an abelian p-group of exponent p for some prime p.

`IsFinite(G): GrpGPC -> BoolElt`

Returns `true` if the group G is finite, `false` otherwise.

`IsNilpotent(G): GrpGPC -> BoolElt`

Returns `true` if the group G is nilpotent, `false` otherwise. This function uses an algorithm described in [Lo98].

`IsPerfect(G): GrpGPC -> BoolElt`

Returns `true` if the group G is perfect, `false` otherwise. A polycyclic group G is perfect, if and only if it is trivial.

`IsSimple(G): GrpGPC -> BoolElt`

Returns `true` if the group G is simple, `false` otherwise. A polycyclic group is simple, if and only if it is cyclic of prime order.

`IsSoluble(G): GrpGPC -> BoolElt`

`IsSolvable(G): GrpGPC -> BoolElt`

Returns `true` if the group G is solvable, `false` otherwise. Every polycyclic group is solvable.

81.9.1 General Properties of Subgroups

`IsCentral(G, H): GrpGPC, GrpGPC -> BoolElt`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsNormal(G, H): GrpGPC, GrpGPC -> BoolElt`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

81.9.2 Properties of Subgroups Requiring a Nilpotent Covering Group

The functions described in this section require the existence of a nilpotent covering group. The are based on algorithms published in [Lo98].

`IsConjugate(G, H, K): GrpGPC, GrpGPC, GrpGPC -> BoolElt, GrpGPCElt`

Given groups G, H, and K with a nilpotent common covering group, return the value `true` if there exists $c \in G$ such that $H^c = K$. If so, the function returns such a conjugating element as second value.

`IsSelfNormalising(G, H): GrpGPC, GrpGPC -> BoolElt`

`IsSelfNormalizing(G, H): GrpGPC, GrpGPC -> BoolElt`

Returns `true` if the subgroup H of the nilpotent group G is self-normalising in G, `false` otherwise.

Example: Intersections of Subgroups of a Nilpotent Group `(ex-905fd9)`

We define a group G on 5 generators $a, \ldots, e$ of infinite order by fixing the commutators of the generators:

$$(b, a) = e^2, \quad (d, c) = e^3$$

All other pairs of generators commute.

```
> F<a,b,c,d,e> := FreeGroup(5);
> rels := [ b^a = b*e^2, b^(a^-1) = b*e^-2, d^c = d*e^3,
>           d^(c^-1) = d*e^-3 ];
> G<a,b,c,d,e> := quo< GrpGPC: F | rels >;
> IsNilpotent(G);
true
```

Since G is nilpotent, we can compute intersections of subgroups of G.

We define the subgroups generated by $a, \ldots, e$ and their nontrivial commutator groups as subgroups of G.

```
> H1 := sub<G|a>;
> H2 := sub<G|b>;
> H3 := sub<G|c>;
> H4 := sub<G|d>;
> H5 := sub<G|e>;
>
> C12 := CommutatorSubgroup(H1, H2);
> {@ G!x : x in PCGenerators(C12) @};
{@ e^2 @}
> C12 subset H5;
true
>
> C34 := CommutatorSubgroup(H3, H4);
> {@ G!x : x in PCGenerators(C34) @};
{@ e^3 @}
> C34 subset H5;
true
```

Finally, we compute the intersection C of $C12$ and $C13$.

```
> C := C12 meet C34;
> {@ G!x : x in PCGenerators(C) @};
{@ e^6 @}
```

This intersection C is cyclic and central in G.

```
> IsCyclic(C);
true
> IsCentral(G, C);
true
```

Example: Subgroup Structure of a Wreath Product (ex-bc5c0c)

Consider the nilpotent group $G := D_{16} \wr 2$ generated by the 5 generators a, b, c, d, t with the relations

$$a^2 = 1, \quad b^{16} = 1, \quad b^a = b^{15}$$

$$c^2 = 1, \quad d^{16} = 1, \quad d^c = d^{15}$$

$$t^2 = 1, \quad a^t = c, \quad b^t = d, \quad c^t = a, \quad d^t = b$$

(All other pairs of generators commute.)

```
> F<t, a,b, c,d> := FreeGroup(5);
> G<t, a,b, c,d> := quo<GrpGPC: F | a^2, b^16, b^a=b^15,
>                                   c^2, d^16, d^c=d^15,
>                       t^2, a^t=c, b^t=d, c^t=a, d^t=b>;
> IsNilpotent(G);
true
```

Since G is nilpotent, we can compute normalisers and centralisers in G.

We define the (dihedral) subgroup $D3$ of G generated by ac and bd and compute its normaliser in G and its centraliser in the (dihedral) subgroup $D2$ of G generated by c and d.

```
> D2 := sub<G|c,d>;
>
> D3<u,v> := sub<G|a*c, b*d>;
> D3;
GrpGPC : D3 of order 2^5 on 2 PC-generators
PC-Relations:
    u^2 = Id(D3),
    v^16 = Id(D3),
    v^u = v^15
>
> N3 := Normaliser(G, D3);
> PCGenerators(N3, G);
{@ t, a * c, b * d, d^8 @}
>
> C3 := Centraliser(D2, D3);
> PCGenerators(C3, G);
{@ d^8 @}
```

Finally we compute the centraliser of the element t in G.

```
> Ct := Centraliser(G, t);
> PCGenerators(Ct, G);
{@ t, a * c, b * d @}
```

81.10 Normal Structure and Characteristic Subgroups

81.10.1 Characteristic Subgroups and Subgroup Series

`Centre(G): GrpGPC -> GrpGPC`

`Center(G): GrpGPC -> GrpGPC`

The centre of the group G. For nilpotent groups the centre is computed using the centraliser algorithm [Lo98]. Otherwise, it is computed as the simultaneous fixed point space of the action of the generators of G on the centre of the Fitting subgroup of G [Eic01].

`DerivedLength(G): GrpGPC -> RngIntElt`

The derived length of the group G.

`DerivedSeries(G): GrpGPC -> [GrpGPC]`

The derived series of the group G. The series is returned as a sequence of subgroups.

`DerivedSubgroup(G): GrpGPC -> GrpGPC`

`DerivedGroup(G): GrpGPC -> GrpGPC`

The derived subgroup of the group G.

`EFASeries(G): GrpGPC -> [GrpGPC]`

Returns a normal series of G, the factors of which are either elementary abelian p-groups or free abelian groups.

`FittingLength(G): GrpGPC -> RngIntElt`

The Fitting length of the group G, i.e. the smallest integer k such that $F_k = G$ where the groups F_i are defined recursively by $F_0 := 1$ and $F_i/F_{i-1} := \mathrm{Fit}(G/F_{i-1})$ $(i > 0)$. Note that such a k exists for every polycyclic group G.

`FittingSeries(G): GrpGPC -> [GrpGPC]`

The Fitting series of the group G, where the groups F_i are defined recursively by $F_0 := 1$ and $F_i/F_{i-1} := \mathrm{Fit}(G/F_{i-1})$ $(i > 0)$. The series is returned as the sequence $[F_0, \ldots, F_k]$ of subgroups of G. Note that every polycyclic group G has a finite Fitting series ending in G.

`FittingSubgroup(G): GrpGPC -> GrpGPC`

`FittingGroup(G): GrpGPC -> GrpGPC`

The Fitting subgroup of the group G, i.e. the maximal nilpotent normal subgroup of G. This function uses an algorithm described in [Eic01].

`HasComputableLCS(G): GrpGPC -> BoolElt`

This function returns the value `true` if the lower central series of G is computable, otherwise it returns the value `false`. This is useful to avoid runtime errors, when *LowerCentralSeries* is called in user written loops or functions.

`LowerCentralSeries(G): GrpGPC -> [GrpGPC]`

The lower central series for the group G. The series is returned as a sequence of subgroups. Since infinite polycyclic groups need not satisfy the descending chain condition for subgroups, computation of the lower central series may fail. To determine if the series can be computed and thereby avoid runtime errors, the function *HasComputableLCS* may be used. This function uses an algorithm described in [Lo98].

`NilpotencyClass(G): GrpGPC -> RngIntElt`

The nilpotency class of the group G. If G is not nilpotent, then -1 is returned.

`NilpotentPresentation(G): GrpGPC -> GrpGPC, Map`

A polycyclic presentation is called *nilpotent*, if for all $i = 1, \ldots, n$, G_{i+1} is normal in G and G_i/G_{i+1} is central in G/G_{i+1}. Every nilpotent polycyclic group has a nilpotent polycyclic presentation. A suitable polycyclic series can be obtained by refining the lower central series.

The function `NilpotentPresentation` computes a group N isomorphic to G, given by a nilpotent polycyclic presentation and the isomorphism from G to N.

`SemisimpleEFASeries(G): GrpGPC -> [GrpGPC]`

Returns a normal series of G, the factors of which are either elementary abelian p-groups which are semisimple as $\mathbf{F}_p[G]$-modules or free abelian groups which are semisimple as $\mathbb{Q}[G]$-modules.

The normal series returned by the function `SemisimpleEFASeries` is a refinement of the normal series returned by the function *EFASeries*.

`UpperCentralSeries(G): GrpGPC -> [GrpGPC]`

The upper central series $[Z_0, \ldots, Z_k]$ of the group G, where the groups Z_k are defined recursively by $Z_0 := 1$ and $Z_i/Z_{i-1} := \mathrm{Z}(G/Z_{i-1})$ $(i > 0)$. The series is returned as a sequence of subgroups of G. Note that since polycyclic groups satisfy the ascending chain condition for subgroups, every polycyclic group G has a finite upper central series.

Example: Lower Central Series of a Dihedral Group (ex-c7b9de)

The dihedral group of order 32 is nilpotent; we compute its lower central series.

```
> D16<a,b> := DihedralGroup(GrpGPC, 16);
> IsNilpotent(D16);
true
> NilpotencyClass(D16);
4
> L := LowerCentralSeries(D16);
```

The generators of the subgroups in the lower central series expressed as elements of $D16$ are:

```
> for i := 1 to 1+NilpotencyClass(D16) do
>   print i, ":", {@ D16!x : x in PCGenerators(L[i]) @};
> end for;
1 : {@ a, b @}
2 : {@ b^2 @}
3 : {@ b^4 @}
4 : {@ b^8 @}
5 : {@ @}
```

We compute a nilpotent presentation and express the new generators in terms of a and b:

```
> N<p,q,r,s,t>, f := NilpotentPresentation(D16);
> N;
GrpGPC : N of order 2^5 on 5 PC-generators
PC-Relations:
    p^2 = Id(N),
    q^2 = r,
    r^2 = s,
    s^2 = t,
    t^2 = Id(N),
    q^p = q * r * s * t,
    r^p = r * s * t,
    s^p = s * t
> {@ x@@f : x in PCGenerators(N) @};
{@ a, b, b^2, b^4, b^8 @}
```

The infinite dihedral group has an infinite, strictly descending, lower central series which cannot be computed:

```
> D := DihedralGroup(GrpGPC, 0);
> HasComputableLCS(D);
false
```

It is easy to see, that $b^{2^{i-1}}$ would be a generator of L_i.

The symmetric group on 3 letters is not nilpotent, but has a lower central series which becomes stationary and which can be computed:

```
> F2<a,b> := FreeGroup(2);
> rels := [ a^2 = F2!1, b^3 = F2!1, b^a = b^2 ];
```

(continues on next page)

(continued from previous page)

```
> G<a,b> := quo<GrpGPC : F2 | rels>;
> G;
GrpGPC : G of order 6 = 2 * 3 on 2 PC-generators
PC-Relations:
    a^2 = Id(G),
    b^3 = Id(G),
    b^a = b^2
> IsNilpotent(G);
false
> HasComputableLCS(G);
true
> L := LowerCentralSeries(G);
> for i := 1 to #L do
>   print i, ":", {@ G!x : x in PCGenerators(L[i]) @};
> end for;
1 : {@ a, b @}
2 : {@ b @}
```

81.10.2 The Abelian Quotient Structure of a Group

AbelianQuotient(G): GrpGPC -> GrpAb, Map

The maximal abelian quotient G/G' of the group G as GrpAb (cf. Chapter *Abelian Groups*). The natural epimorphism is returned as second return value.

AbelianQuotientInvariants(G): GrpGPC -> [RngIntElt]

AQInvariants(G): GrpGPC -> [RngIntElt]

Returns a sequence containing the invariants of the maximal abelian quotient G/G' of the group G. Each infinite cyclic factor of G/G' is represented by zero.

ElementaryAbelianQuotient(G, p): GrpGPC, RngIntElt -> GrpAb, Map

The maximal p-elementary abelian quotient of the group G as GrpAb (cf. Chapter *Abelian Groups*). The natural epimorphism is returned as second return value.

FreeAbelianQuotient(G): GrpGPC -> GrpAb, Map

The maximal free abelian quotient of the group G as GrpAb (cf. Chapter *Abelian Groups*). The natural epimorphism is returned as second return value.

81.11 Conjugacy

The functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98].

IsConjugate(G, g, h): GrpGPC, GrpGPCElt, GrpGPCElt -> BoolElt, GrpGPCElt

Given elements g and h and a group G, which are contained in some nilpotent common group, return the value true if there exists $c \in G$ such that $g^c = h$. If so, the function returns such a conjugating element as second value.

IsConjugate(G, H, K): GrpGPC, GrpGPC, GrpGPC -> BoolElt, GrpGPCElt

Given groups G, H, and K with a nilpotent common covering group, return the value true if there exists $c \in G$ such that $H^c = K$. If so, the function returns such a conjugating element as second value.

Example: Conjugacy in a Nilpotent Wreath Product (ex-f79f1e)

We again consider the nilpotent group $G := D_{16} \wr 2$.

```
> F<t, a,b, c,d> := FreeGroup(5);
> G<t, a,b, c,d> := quo<GrpGPC: F | a^2, b^16, b^a=b^15,
>                                   c^2, d^16, d^c=d^15,
>                       t^2, a^t=c, b^t=d, c^t=a, d^t=b>;
> IsNilpotent(G);
true
```

Since G is nilpotent, a test for conjugacy in G is available.

We define the following subgroups of G: $D1$ generated by a and b, $D2$ generated by c and d, and $D3$ generated by ac and bd.

```
> D1 := sub<G|a,b>;
> D2 := sub<G|c,d>;
> D3<u,v> := sub<G|a*c, b*d>;
>
```

$D1$ and $D2$ are, of course, conjugate in G; t is a conjugating element.

```
> IsConjugate(G, D1, D2);
true t
```

The elements b and d^{-1} are conjugate in G; we compute a conjugating element.

```
> IsConjugate(G, b, d^-1);
true t * a * c
```

However, neither the subgroups $D1$ and $D2$ nor the elements b and d^{-1}, are conjugate in the subgroup $D3$.

```
> IsConjugate(D3, D1, D2);
false
> IsConjugate(D3, b, d^-1);
false
```

81.12 Representation Theory

This section describes some functions for creating $R[G]$-modules for a polycyclic group G, which are unique for this category or have special properties when called for polycyclic groups. For a complete description of the functions available for creating and working with $R[G]$-modules we refer to Chapter Modules over an Algebra and Group Representations.

Note that the function `GModuleAction(M): ModGrp → Map(Hom)` can be used to extract the matrix representation associated to an $R[G]$-module.

`EFAModuleMaps(G): GrpGPC -> [ModGrp]`

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \cdots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian or p-elementary abelian for some prime p. The action of G by conjugation induces a $\mathbb{Z}[G]$-module structure on M_i if M_i is free abelian and an $\mathbf{F}_p[G]$-module structure if M_i is p-elementary abelian.

This function returns a sequence $[f_1, \ldots, f_r]$, where $f_i : N_i \to M_i$ is the natural epimorphism onto the additive group of an $R_i[G]$-module M_i ($R_i \in \{\mathbf{F}_p, \mathbb{Z}\}$), constructed as above.

The functions **`EFAModuleMaps`** and *`EFAModules`* use the normal series returned by the function *`EFASeries`*.

Note that the kernels of the epimorphisms f_i can be computed and hence it is possible to form preimages of submodules of M_i, which are normal subgroups of G contained in N_i and containing N_{i+1}.

EFAModules(G): GrpGPC -> [ModGrp]

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \cdots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian or p-elementary abelian for some prime p. The action of G by conjugation induces a $\mathbb{Z}[G]$-module structure on M_i if M_i is free abelian and an $\mathbf{F}_p[G]$-module structure if M_i is p-elementary abelian.

This function returns a sequence $[M_1, \ldots, M_r]$ of $R_i[G]$-modules (where $R_i \in \{\mathbf{F}_p, \mathbb{Z}\}$), constructed as above.

The functions *EFAModuleMaps* and **EFAModules** use the normal series returned by the function *EFASeries*.

GModule(G, A, p): GrpGPC, GrpGPC, RngIntElt -> ModGrp, Map

GModule(G, A): GrpGPC, GrpGPC -> ModGrp, Map

Let A be a normal subgroup of the polycyclic group G. If $p = 0$, the function returns the $\mathbb{Z}[G]$-module corresponding to the conjugation action of G on the maximal free abelian quotient of A. If p is a prime, it returns the $\mathbf{F}_p[G]$-module corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A. The epimorphism $\pi : A \to M$ onto the additive group of the constructed module M is returned as second return value. Note that the kernel of π can be computed and hence it is possible to form preimages of submodules of M, which are normal subgroups of G contained in A.

If the maximal abelian quotient A/A' of A is either free abelian or p-elementary abelian for some prime p, p can be omitted in the function call.

Note that it is the user's responsibility to ensure that A is in fact normal in G.

GModule(G, A, B, p): GrpGPC, GrpGPC, GrpGPC, RngIntElt -> ModGrp, Map

GModule(G, A, B): GrpGPC, GrpGPC, GrpGPC -> ModGrp, Map

Let A and $B < A$ be normal subgroups of the polycyclic group G. If $p = 0$, the function returns the $\mathbb{Z}[G]$-module corresponding to the conjugation action of G on the maximal free abelian quotient of A/B. If p is a prime, it returns the $\mathbf{F}_p[G]$-module corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A/B. The epimorphism $\pi : A \to M$ onto the additive group of the constructed module M is returned as second return value. Note that the kernel of π can be computed and hence it is possible to form preimages of submodules of M, which are normal subgroups of G contained in A and containing B.

If the maximal abelian quotient of A/B is either free abelian or p-elementary abelian for some prime p, p can be omitted in the function call.

Note that it is the user's responsibility to ensure that A and B are in fact normal in G.

GModulePrimes(G, A): GrpGPC, GrpGPC -> SetMulti

Let G be a polycyclic group and A a normal subgroup of G. Given any prime p, the maximal p-elementary abelian quotient of A can be viewed as an $\mathbf{F}_p[G]$-module M_p. The maximal free abelian quotient of A can be viewed as a $\mathbf{Z}[G]$-module M_0. This function determines those primes p for which the module M_p is non-trivial (i.e. not zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A contains m copies of $\mathbb{Z}$. In this case, the rank of M_0 is m and M_p is non-trivial for every prime p and its rank is the sum of m and the multiplicity of p in S.

GModulePrimes(G, A, B): GrpGPC, GrpGPC, GrpGPC -> SetMulti

Let G be a polycyclic group, A a normal subgroup of G, and B a normal subgroup of G contained in A. Given any prime p, the maximal p-elementary abelian quotient of A/B can be viewed as an $\mathbf{F}_p[G]$-module M_p. The maximal free abelian quotient of A/B can be viewed as a $\mathbf{Z}[G]$-module M_0. This function determines those primes p for which the module M_p is non-trivial (i.e. not zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A/B is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A/B

contains m copies of $\mathbb{Z}$. In this case, the rank of M_0 is m and M_p is non-trivial for every prime p and its rank is the sum of m and the multiplicity of p in S.

SemisimpleEFAModuleMaps(G): GrpGPC -> [ModGrp]

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \cdots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian and semisimple as a $\mathbb{Q}[G]$-module or p-elementary abelian and semisimple as an $\mathbf{F}_p[G]$-module for some prime p.

This function returns a sequence $[f_1, \ldots, f_r]$, where $f_i : N_i \to M_i$ is the natural epimorphism onto the additive group of an $R_i[G]$-module M_i ($R_i \in \{\mathbf{F}_p, \mathbb{Z}\}$), constructed as above.

The functions *SemisimpleEFAModules* and SemisimpleEFAModuleMaps use the normal series returned by the function *SemisimpleEFASeries*. Moreover, this normal series is a refinement of the normal series returned by the function *EFASeries*.

Note that the kernels of the epimorphisms f_i can be computed and hence it is possible to form preimages of submodules of M_i, which are normal subgroups of G contained in N_i and containing N_{i+1}.

SemisimpleEFAModules(G): GrpGPC -> [ModGrp]

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \cdots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian and semisimple as a $\mathbb{Q}[G]$-module or p-elementary abelian and semisimple as an $\mathbf{F}_p[G]$-module for some prime p.

This function returns a sequence $[M_1, \ldots, M_r]$ of $R_i[G]$-modules (where $R_i \in \{\mathbf{F}_p, \mathbb{Z}\}$), constructed as above.

The functions SemisimpleEFAModules and *SemisimpleEFAModuleMaps* use the normal series returned by the function *SemisimpleEFASeries*. Moreover, this normal series is a refinement of the normal series returned by the function *EFASeries*.

Example: Semisimple Modules of a Polycyclic Group (ex-e27464)

Consider the group G defined by the polycyclic presentation

$$\langle a, b, c, d, e \mid c^6, e^3, d^c = de, e^c = e^2, b^a = b^{-1}, b^{a^{-1}} = b^{-1},$$
$$c^b = ce, c^{b^{-1}} = ce, d^b = d^{-1}, d^{b^{-1}} = d^{-1}, e^b = e^2, e^{b^{-1}} = e^2\rangle.$$

(Trivial commutator relations have been omitted.)

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                                   d^c=d*e, e^c=e^2,
>                                   b^a=b^-1,
>                                   b^(a^-1)=b^-1,
>                                   c^b=c*e,
>                                   c^(b^-1)=c*e,
>                                   d^b=d^-1,
>                                   d^(b^-1)=d^-1,
>                                   e^b=e^2,
>                                   e^(b^-1)=e^2 >;
```

The subgroup H of G generated by c, d, e is normal in G.

```
> H := sub< G | c,d,e >;
> IsNormal(G, H);
true
```

We determine the primes such that the action of G on H yields non-trivial modules.

```
> GModulePrimes(G, H);
{* 0, 2, 3 *}
```

We construct the $\mathbf{F}_3[G]$-module M given by the action of G on the maximal 3-elementary abelian quotient of H and the natural epimorphism π from H onto the additive group of M.

```
> M, pi := GModule(G, H, 3);
> M;
GModule M of dimension 2 over GF(3)
```

Using the function `Submodules(M): ModRng → [ModRng]`, we obtain the submodules of M. Their preimages under π are precisely the normal subgroups of G which are contained in H and contain $\ker(\pi)$.

```
> submod := Submodules(M);
> nsgs := [ m @@ pi : m in submod ];
> [ PCGenerators(s, G) : s in nsgs ];
[
    {@ c^3, d^3, e @},
    {@ c, d^3, e @},
    {@ c^3, d, e @},
    {@ c, d, e @}
]
```

Example: Primes of Semisimple G-Modules `(ex-e34f40)`

Consider the group defined by the polycyclic presentation

$$< a, b, c, d, e \mid a^5, b^5, c^6, d^5, e^3, b^a = bd > .$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F |
>                       a^5, b^5, c^6, d^5, e^3, b^a = b*d >;
```

Obviously the subgroup of G generated by b, c, d, e is normal in G.

```
> H := sub< G | b,c,d,e >;
> IsNormal(G, H);
true
```

We use the function *GModulePrimes* to determine the set of primes p for which the action of G on the maximal p-elementary abelian quotient of H induces a nontrivial $\mathbf{F}_p[G]$-module.

```
> P := GModulePrimes(G, H);
> 0 in P;
false
```

0 is not contained in P, i.e. the maximal free abelian quotient of H is trivial. Hence, there are only finitely many primes, satisfying the condition above.

We loop over the distinct elements of P and for each element p we construct the induced $\mathbf{F}_p[G]$-module, print its dimension, and check whether it is decomposable. Note that the dimension of the module for p must be equal to the multiplicity of p in P.

```
> for p in MultisetToSet(P) do
>     M := GModule(G, H, p);
>     dim := Dimension(M);
>     decomp := IsDecomposable(M);
>
>     assert dim eq Multiplicity(P, p);
>
>     print "prime", p, ": module of dimension", dim;
```

(continues on next page)

(continued from previous page)

```
>    if decomp then
>       print "  has a nontrivial decomposition";
>     else
>       print "  is indecomposable";
>    end if;
> end for;
prime 2 : module of dimension 1
  is indecomposable
prime 3 : module of dimension 2
  has a nontrivial decomposition
prime 5 : module of dimension 2
  is indecomposable
```

Example: Fitting Subgroup via Semisimple Modules (ex-407bf7)

The Fitting subgroup of a polycyclic group G can be characterised as the intersection of the centralisers in G of the semisimple G-modules defined by the action of G on the factors of a semisimple EFA-series of G. The centraliser in G of a G-module is just the kernel of the action map.

We illustrate this with the group G defined in the example above.

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                                   b^a = b * d,
>                                   b^(a^-1) = b * d^-1 >;
```

We first construct the G-modules defined by the action of G on the factors of a semisimple EFA-series of G.

```
> modules := SemisimpleEFAModules(G);
> modules;
[
    GModule of dimension 2 over Integer Ring,
    GModule of dimension 1 over Integer Ring,
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(3)
]
```

Now, we compute the intersection of the kernels of the module action maps, which can be obtained using the function `GModuleAction(M): ModGrp → Map(Hom)`.

```
> S := G;
> for m in modules do
>    S meet:= Kernel(GModuleAction(m));
> end for;
```

Finally, we compare the result with the Fitting subgroup of G, returned by the Magma function *FittingSubgroup*.

```
> S eq FittingSubgroup(G);
true
```

Example: Refining an EFA-Series with Module Maps (ex-5b5ab4)

The functions *EFAModuleMaps* and *SemisimpleEFAModuleMaps* are useful whenever it is desired to refine an EFA-series or a semisimple EFA-series by computing the subgroups corresponding to submodules of the G-modules given by the factors of the series. Consider again the group defined above.

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                                   b^a = b * d,
>                                   b^(a^-1) = b * d^-1 >;
```

We extract the map f from G (the first group in any EFA-series of G) onto the module given by the first factor of an EFA-series of G.

```
> f := EFAModuleMaps(G)[1];
> f;
Mapping from: GrpGPC: G to GModule of dimension 2 over Integer
Ring
```

The module itself can be accessed as the codomain of f.

```
> M := Codomain(f);
> M;
GModule M of dimension 2 over Integer Ring
```

Spinning up random elements, we try to construct a submodule S of M.

```
> repeat
>    S := sub<M|[Random(-1, 1): i in [1 .. Dimension(M)]]>;
>    until Dimension(S) gt 0 and S ne M;
> S;
GModule S of dimension 1 over Integer Ring
```

The preimage N of S under f is a normal subgroup of G, which lies between the first and the second subgroup of the original EFA-series for G.

```
> N := S @@ f;
> PCGenerators(N, G);
{@ a^2 * b^4, c, d, e @}
```

81.13 Power Groups

`Parent(G): GrpGPC -> PowStr`

The PowerStructure of category `GrpGPC`.

`PowerGroup(G): GrpPC -> PowerGroup`

The set of all subgroups of G. This is very useful when constructing sets of polycyclic groups. If the user will be building a set of subgroups of a polycyclic group G, then it is best to specify the set's universe to be `PowerGroup(G)`. If the set's universe is not specified it will be the parent structure of G as returned by *Parent*`(G)`.

81.14 Bibliography

- [CLGM+95] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.
- [Eic01] Bettina Eick. On the Fitting subgroup of a polycyclic-by-finite group and its applications. *J. Algebra*, 242(1):176–187, 2001.
- [Geb02] Volker Gebhardt. Efficient collection in infinite polycyclic groups. *J. Symbolic Comput.*, 34(3):213–228, 2002.

- [Lo98] Eddie H. Lo. Finding intersections and normalizers in finitely generated nilpotent groups. *J. Symbolic Comput.*, 25(1):45–59, 1998.
- [Sim94] Charles C. Sims. *Computation with Finitely Presented Groups.* Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

Chapter 82

Braid Groups

82.1 Introduction

This chapter deals with another, quite specialised, class of finitely presented groups for which the word problem is solvable, the category of braid groups. The corresponding Magma category is called `GrpBrd`.

The notion of braid groups was introduced by Artin [Art47], who considered a sequence B_n $(n = 1, 2, \dots)$ of groups, where B_n is presented on $n-1$ generators $\sigma_1, \dots, \sigma_{n-1}$ with the defining relations

$$\begin{aligned} \sigma_i\sigma_j &= \sigma_j\sigma_i \qquad (1 \le i < j < n, \quad j - i > 1), \\ \sigma_i\sigma_{i+1}\sigma_i &= \sigma_{i+1}\sigma_i\sigma_{i+1} \quad (1 \le i < n-1) \quad . \end{aligned}$$

B_n is called the *braid group on n strings*. In the sequel, we refer to the above presentation as *Artin presentation* and to $\sigma_1, \dots, \sigma_{n-1}$ as *Artin generators* of B_n.

Birman, Ko, and Lee introduced an alternative way of presenting braid groups [BKL98]. Here, B_n is presented on $n(n-1)/2$ generators $a_{r,t}$ $(n \ge r > t \ge 1)$ with the defining relations

$$\begin{aligned} a_{t,s}a_{r,q} &= a_{r,q}a_{t,s} \quad \text{for } n \ge t > s \ge 1,\ n \ge r > q \ge 1,\ (t-r)(t-q)(s-r)(s-q) > 0 \\ a_{t,s}a_{s,r} &= a_{t,r}a_{t,s} = a_{s,r}a_{t,r} \quad \text{for } n \ge t > s > r > 0 \quad . \end{aligned}$$

We refer to this presentation as *BKL presentation* and to $a_{r,t}$ $(n \ge r > t \ge 1)$ as *BKL generators* of B_n.

A possible choice for the BKL generators in terms of Artin generators is $a_{r,t} = (\sigma_{r-1} \cdots \sigma_{t+1})\sigma_t(\sigma_{t+1}^{-1} \cdots \sigma_{r-1}^{-1})$. This identification is used in Magma.

Recently, braid groups came under consideration as possible sources for public key cryptosystems [AAG99, KLC+00]. The features of braid groups which make them interesting for public key cryptography are the following.

- The basic group operations in braid groups can be implemented efficiently on a computer.
- The word problem in braid groups is solvable, that is, there is a normal form for elements of a braid group and elements can be compared. Moreover, there are algorithms which are able to compute the normal form of an element efficiently.
- There are several problems in braid groups which are believed to be mathematically hard and whose use for cryptographic purposes has been suggested. The most important examples are variations of the conjugacy problem.

However, both recent attacks on particular cryptosystems [GKT+02, HS03, Hug02, LP03, LL02] and advances in the analysis of the conjugacy problem [Geb03, GM02] in general shed some doubts on the security of braid group cryptosystems. At the time of this writing it is an open question whether braid group cryptosystems can be made secure by an appropriate choice of parameters and keys or whether they have to be considered as insecure. More research into these issues is necessary.

The Magma category `GrpBrd` was introduced mainly with these applications in mind. Focus was put on providing fast operations with elements and on giving the user as much control over the details of computations as possible.

82.1.1 Lattice Structure and Simple Elements

In this section we briefly recall the basic terminology used for describing elements of braid groups. More detailed descriptions can be found in [ECH+92] for the Artin presentation and in [BKL98] for the BKL presentation.

We remark that both Artin presentation and BKL presentation are special cases of so-called *Garside groups* [Deh02].

In the sequel, let M be either the Artin presentation or the BKL presentation of the braid group B on n strings, let X denote the generators of M and let R denote the relations of M. As the relations in R do not contain inverses of generators, we can interpret M as monoid presentation. We denote the finitely presented monoid defined by M by B^+. The natural homomorphism from B^+ to B can be shown to be injective. We identify its image with B^+ and call it the set of *positive* elements of B. Finally, we denote the identity of B by 1.

We can now define two partial orderings on B. For elements $u, v \in B$ we say $u \preceq v$, if there exists a positive element a such that $ua = v$, and we say $v \succeq u$, if there exists a positive element a such that $v = au$. Note that these partial orderings are different; $u \preceq v$ is not equivalent to $v \succeq u$.

B can be shown to be a lattice with respect to both partial orderings, that is, for elements $u, v \in B$ there are elements $d_l, m_l, d_r, m_r \in B$ such that

$$\begin{aligned}
d_l \preceq u,\ d_l \preceq v \quad &\text{and} \quad d \preceq u,\ d \preceq v \ \text{ implies } \ d \preceq d_l \ \text{ for all } \ d \in B \\
u \preceq m_l,\ v \preceq m_l \quad &\text{and} \quad u \preceq m,\ v \preceq m \ \text{ implies } \ m_l \preceq m \ \text{ for all } \ m \in B \\
u \succeq d_r,\ v \succeq d_r \quad &\text{and} \quad u \succeq d,\ v \succeq d \ \text{ implies } \ d_r \succeq d \ \text{ for all } \ d \in B \\
m_r \succeq u,\ m_r \succeq v \quad &\text{and} \quad m \succeq u,\ m \succeq v \ \text{ implies } \ m \succeq m_r \ \text{ for all } \ m \in B \quad .
\end{aligned}$$

We call d_l, m_l, d_r and m_r the *left-gcd*, the *left-lcm*, the *right-gcd*, and the *right-lcm*, respectively, of u and v.

It can be shown that the left-lcm of the elements of X and the right-lcm of the elements of X are equal; we call this element the *fundamental element* of the presentation M and denote it by D. The fundamental element is crucial for the study of braid groups. One of its most important properties is that a certain power D^N of D generates the centre of B. ($N = 2$ for the Artin presentation and $N = n$ for the BKL presentation.) Moreover, $u \preceq D^k$ is equivalent to $D^k \succeq u$ and $D^k \preceq u$ is equivalent to $u \succeq D^k$ for all $k \in \mathbb{Z}, u \in B$.

In Magma, the partial ordering $\preceq$ is provided as operator `u le v` and the partial ordering $\succeq$ is provided as operator `u ge v`; see Section *Boolean Predicates for Elements*. For a description of the functions computing lcm and gcd of elements, see Section *Lattice Operations*.

The positive elements c of B satisfying $c \preceq D$ are called *simple elements*; we denote the set of simple elements by C. simple elements can be uniquely described by permutations on n points. In Magma, a simple element c inducing a permutation π on the strings on which B is defined, is represented by the permutation π^{-1}.

If M is the Artin presentation, every permutation on n points corresponds to a simple element, that is, $|C| = n!$.

If M is the BKL presentation, $|C| = (2n)!/(n!(n+1)!)$ and only permutations on n points which are products of parallel, descending cycles correspond to simple elements. Here, a cycle $(i_1, \ldots, i_r)$ is called *descending* if $i_1 > \cdots > i_r$ and two descending cycles $(i_1, \ldots, i_r)$ and $(j_1, \ldots, j_s)$ are called *parallel* if $(i_k - j_l)(i_k - j_{l'})(i_{k'} - j_l)(i_{k'} - j_{l'}) > 0$ for all $1 \le k, k' \le r$ and $1 \le l, l' \le s$. The descending cycle $(i_1, \ldots, i_r)$ corresponds to the element $a_{i_1,i_2} a_{i_2,i_3} \cdots a_{i_{r-1},i_r}$ of B. It is obvious from the defining relations that the simple elements defined by two parallel descending cycles commute.

Every element u of B can be written in the form $u = D^l c_1 \cdots c_k$, where l is a suitable integer and $c_1, \ldots c_k$ are simple elements. We call representations of this form *simple element representations* or *canonical factor products* (CFP).

82.1.2 Representing Elements of a Braid Group

This section describes the ways in which elements of a braid group can be represented internally by Magma. From the user's point of view, this mainly affects input and printing of elements. This section is intended to be a concise overview; for a detailed description of functions and for examples we refer to Section *Constructing and Accessing Braid Groups*, Section *Creating Elements of a Braid Group*, and Section *Accessing Information*.

Since an element of a braid group B can be represented either as word in the generators or as product of simple elements (see Section *Lattice Structure and Simple Elements*) with respect to either the Artin presentation or the BKL presentation of B, there are four different ways of representing elements of B, which can be used for entering or printing elements and for computing with elements.

Automatic Conversions

Magma can work with all the above representations and conversions are done automatically when necessary, for example, when multiplying an element defined as word in the Artin generators with an element given as product of simple elements for the BKL presentation. Hence, the user normally does not have do give too much thought about how elements are represented. It should be noted, however, that automatic conversions can affect performance and that in time critical situations, the best results in general are obtained if automatic conversions are avoided.

Default Presentations

When creating a braid group B using the command *BraidGroup*, the user can specify whether the Artin presentation or the BKL presentation should be used as *default presentation* for B. Unless specified otherwise by the user, this presentation is used in all subsequent operations with B or with elements of B. In particular, group operations and printing of elements are performed with respect to this presentation. It is possible to change the default presentation using the command *SetPresentation*. Certain commands accept a parameter `Presentation`, which can be used to perform that command with respect to a presentation other than the default presentation.

Representation Used for Group Operations

By default, group operations with elements of a braid group B are performed using representations of the elements as products of simple elements for the default presentation of B. Experienced users can change this behaviour using the command *SetForceCFP*. If this flag is set to `false`, arguments of a group operation are not automatically converted into CFP representation if both arguments are represented as words in the generators of the default presentation of B, but the operation is performed, if possible, using the word representations instead.

Printing of Elements

The default printing format for an element u of a braid group B is that both a representation of u as word in the generators of the default presentation of B and a representation of u as product of simple elements for the default presentation of B are printed.

Depending on the application, the user may wish to change the print format so that only one of the above representations of u is printed. This can be achieved using the command *SetElementPrintFormat*.

82.1.3 Normal Form for Elements of a Braid Group

This section briefly describes the normal form for elements of braid groups. For details we refer to [ECH+92] and [BKL98]. The Magma commands for computing normal forms are described in Section *Computing Normal Forms of Elements*.

Let B be the braid group on n strings and fix a presentation M for B, either the Artin presentation or the BKL presentation. A product of simple elements $D^l c_1 \cdots c_k$ is said to be in *left normal form* with respect to M, if $c_1 \neq D$, $c_k \neq 1$, and $(c_i^{-1} D) \wedge_l c_{i+1} = 1$ for $i = 1, \ldots, k-1$, where $(c_i^{-1} D) \wedge_l c_{i+1}$ denotes the left-gcd of $c_i^{-1} D$ and c_{i+1} with respect to the presentation M.

Similarly, we define $c_1 \cdots c_k D^l$ to be in *right normal form* with respect to M, if $c_k \neq D$, $c_1 \neq 1$, and $c_i \wedge_r (D c_{i+1}^{-1}) = 1$ for $i = 1, \ldots, k-1$, where $\wedge_r$ denotes right-gcd with respect to the presentation M.

It can be shown that the numbers of simple elements and the powers of D in the left and right normal forms of an element are equal, that is, if $x \in B$ has left normal form $D^l c_1 \cdots c_k$ and right normal form $\bar{c}_1 \cdots \bar{c}_{k'} D^{l'}$ then $k' = k$ and $l' = l$. In this situation we call l the *infimum* of x, denoted by $\inf(x)$, k the *canonical length* of x, denoted by $\mathrm{len}(x)$, and $l + k$ the *supremum* of x, denoted by $\sup(x)$. l is the maximal integer d satisfying $D^d \preceq x$ and $l + k$ is the minimal integer d satisfying $x \preceq D^d$.

To bring a product $D^l c_1 \cdots c_k$ of simple elements into left normal form, we proceed by induction, assuming that $D^l c_1 \cdots c_{k-1}$ is in left normal form. For $i = k-1, \ldots, 1$ we now compute $d = (c_i^{-1} D) \wedge_l c_{i+1}$ and, if $d \neq 1$, replace c_i by $c_i d$ and c_{i+1} by $d^{-1} c_{i+1}$. Finally, we delete trailing trivial simple elements and absorb simple elements equal to D into the leading power of D. The result can be shown to be in left normal form [BKL98, ECH+92].

Both the theoretical complexity of this algorithm and its performance in practice are determined by the gcd computations.

For the Artin presentation, the cost of computing the left-gcd of two simple elements is $O(n \log n)$ [ECH+92], whence the complexity of bringing a product of simple elements as above into left normal form is $O(k^2 n \log n)$.

For the Artin presentation, the cost of computing the left-gcd of two simple elements is $O(n)$ [BKL98], whence the complexity of bringing a product of simple elements as above into left normal form is $O(k^2 n)$.

Computing right normal forms is analogous.

82.1.4 Mixed Canonical Form and Lattice Operations

This section outlines the algorithms used for lattice operations in a braid group. Let u and v be elements of a braid group B and let M be either the Artin presentation or the BKL presentation of B. The Magma commands for computing mixed canonical forms are described in Section *Computing Normal Forms of Elements* and the commands providing lattice operations are described in Section *Lattice Operations*.

Evaluating partial orderings for u and v with respect to M is straightforward. $u \preceq v$ if and only if $u^{-1}v$ is a positive element with respect to M. The latter can be decided by computing the left normal form $D^l c_1 \cdots c_k$ of $u^{-1}v$ with respect to M: $u^{-1}v$ is positive if and only if $l \geq 0$. Evaluating the partial ordering $\succeq$ is analogous.

We call the tuple $< a, b >$ the *left-mixed canonical form* of an element $x \in B$, if $a = a_1 \cdots a_k$ and $b = b_1 \cdots b_s$ are positive elements in left normal form ($a_1 = D$, $b_1 = D$ is permitted), $x = a^{-1}b$ and the left-gcd of a_1 and b_1 is trivial.

Similarly, we call the tuple $< a, b >$ the *right-mixed canonical form* of x, if $a = a_1 \cdots a_k$ and $b = b_1 \cdots b_s$ are positive elements in right normal form ($a_k = D$, $b_s = D$ is permitted), $x = ab^{-1}$ and the right-gcd of a_k and b_s is trivial.

It is not difficult to show that the left-gcd of u and v is given by ua^{-1}, where $< a, b >$ is the left-mixed canonical form of $u^{-1}v$, and that the right-gcd of u and v is given by $a^{-1}u$, where $< a, b >$ is the right-mixed canonical form of uv^{-1} [ECH+92].

Similarly, the left-lcm of u and v is given by ua, where $< a, b >$ is the right-mixed canonical form of $u^{-1}v$ and the right-lcm of u and v is given by au, where $< a, b >$ is the left-mixed canonical form of uv^{-1}

Computing the left-mixed canonical form of an element x can, after writing $x = a^{-1}b$ with two positive elements a and b, easily be reduced to computing repeatedly the left-normal forms of a and b and cancelling the left-gcd of the leading simple elements. Computing the right-mixed canonical form is analogous.

82.1.5 Conjugacy Testing and Conjugacy Search

Conjugacy testing, that is, deciding whether two given braids are conjugate, and conjugacy search, that is, computing a conjugating element for a pair of conjugate braids, are of particular importance to public key cryptosystems based on braid groups. Known algorithms for both conjugacy testing and conjugacy search require the (at least partial) computation of an invariant of the conjugacy classes of the elements in question, either the *super summit set* [ERM94, Gar69] or the *ultra summit set* [Geb03].

This section recalls the definition of these invariants and sketches the algorithms used for computing them, for conjugacy testing and for conjugacy search. The relevant Magma commands are described in Section *Invariants of Conjugacy Classes*.

For this section let B be a braid group and let M be either the Artin presentation or the BKL presentation of B.

Definition of the Class Invariants

We define two operations, the *cycling* operation $\mathbf{c}$ and the *decycling* operation $\mathbf{d}$, each mapping an arbitrary element $x \in B$ to a conjugate of x as follows. Let $x \in B$ be a braid with left normal form $x = D^l c_1 \cdots c_k$ as defined in Section *Normal Form for Elements of a Braid Group*. If $k = 0$, we define $\mathbf{c}(x) = x$ and $\mathbf{d}(x) = x$. Otherwise, we define $\mathbf{c}(x) = D^l c_2 \cdots c_k (c_1^{D^{-l}})$ and $\mathbf{d}(x) = D^l (c_k^{D^l}) c_1 \cdots c_{k-1}$.

We now fix an element $x \in B$ and consider the set C_x of all conjugates of x. Proofs for the following facts can be found in [ECH+92] or [BKL98].

- The set $\{\inf(y) : y \in C_x\}$ is bounded above; we denote its maximum by ss-inf(x).
- The set $\{\sup(y) : y \in C_x\}$ is bounded below; we denote its minimum by ss-sup(x).
- The maximum of inf on C_x and the minimum of sup on C_x can be achieved simultaneously.

We define three sets of conjugates of x as follows.

- The set $P_x = \{y \in C_x : y \in B^+\}$, containing the positive conjugates of x.
- The set $S_x = \{y \in C_x : \inf(y) = \text{ss-inf}(x), \sup(y) = \text{ss-sup}(x)\}$, called the *super summit set* of x.
- The set $U_x = \{y \in S_x : \exists\, i > 0 : \mathbf{c}^i(y) = y\}$, called the *ultra summit set* of x.

Clearly, the sets P_x, S_x, and U_x only depend on the conjugacy class of x. Moreover, the set P_x is empty if ss-inf$(x) < 0$ and it contains S_x if ss-inf$(x) \geq 0$.

Proofs of the following properties can be found in [ECH+92] and [BKL98] for the sets P_x and S_x and in [Geb03] for the set U_x.

- The sets P_x, S_x, and U_x are finite.
- The sets S_x and U_x are non-empty.
- Representatives of P_x, S_x, and U_x, respectively, can be obtained from x by a finite number of cycling and decycling operations.

Computing the Class Invariants

The main tools for computing the class invariants introduced in Section *Definition of the Class Invariants* are the following "convexity" results established in [ERM94] and [FGM03] for the sets P_x and S_x and in [Geb03] for the set U_x. Let $I_x \in \{P_x, S_x, U_x\}$.

- For $y, z \in I_x$, there exists a finite sequence $y = y_0, \ldots, y_r = z$ such that for $i = 1, \ldots, r$, $y_i \in I_x$ and $y_i = y_{i-1}^{c_i}$ for a simple element c_i.
- For $y \in I_x$ and a simple element c, there exists a unique $\preceq$-minimal element $\iota_y(c)$ such that $c \preceq \iota_y(c)$ and $y^{\iota_y(c)} \in I_x$. Moreover, $\iota_y(c)$ is simple.

By the above results, any non-empty subset $I \subseteq I_x$ with the property that $y^{\iota_y(s)} \in I$ for all $y \in I$ and all generators s of the presentation M is equal to I_x. In particular, I_x can be computed, starting from a single representative, as closure under conjugation with minimal simple elements.

Algorithms for computing the minimal simple elements $\iota_y(c)$ are given in [FGM03] for the case $I_x \in \{P_x, S_x\}$ and in [Geb03] for the case $I_x = U_x$.

The Magma commands for computing the class invariants P_x, S_x, and U_x as well as corresponding minimal simple elements $\iota_y(c)$ are described in Section *Invariants of Conjugacy Classes*.

Conjugacy Testing and Conjugacy Search

Testing conjugacy of two braids $x, y \in B$ can be performed using either super summit sets or ultra summit sets. It is obvious from the results cited in Section *Definition of the Class Invariants* that the following are equivalent.

- x and y are conjugate in B.
- $S_x = S_y$.
- $U_x = U_y$.
- $S_x \cap S_y \neq \emptyset$.
- $U_x \cap U_y \neq \emptyset$.

If x and y are conjugate, a conjugating element can be obtained by establishing an element $z \in S_x \cap S_y$ or $z \in U_x \cap U_y$ both as conjugate of x and of y and keeping track of the conjugating elements in each step.

The size of super summit sets grows rapidly with increasing values of braid index n and canonical length. In general, computing super summit sets is difficult or infeasible for braids on more than 5-10 strings, except for very short canonical lengths. Ultra summit sets, on the other hand tend to be much smaller and can frequently be computed for braids on up to 100 strings and canonical length up to 1000, provided sufficient memory is available [Geb03]. Conjugacy search may be successful even in situations where the entire class invariant is too large to be computed.

In Magma, conjugacy testing and conjugacy search based on ultra summit sets is provided by the function *IsConjugate*.

82.2 Constructing and Accessing Braid Groups

This section describes the facilities for creating a braid group and for accessing and changing its basic properties.

`BraidGroup(n: parameters): RngIntElt -> GrpBrd`

`BraidGroup(GrpBrd, n: parameters): Cat, RngIntElt -> GrpBrd`

`Presentation: MonStgElt`	`Default: "Artin"`

Given a small integer $n > 0$, return the braid group on n strings, that is, $n - 1$ Artin generators.

The presentation can be selected using the parameter `Presentation`. Possible values for this parameter are `"Artin"` (default) and `"BKL"`.

`GetPresentation(B): GrpBrd -> MonStgElt`

Returns a string s, either `"Artin"` or `"BKL"`, indicating the presentation currently used for B.

`SetPresentation(~B, s): GrpBrd, MonStgElt`

Set the presentation used for B to the presentation indicated by s. Possible values for s are `"Artin"` and `"BKL"`.

`GetForceCFP(B): GrpBrd -> BoolElt`

Returns whether arithmetic operations with elements of B are always done using representations of the elements as products of simple elements.

`SetForceCFP(~B, b): GrpBrd, BoolElt`

By default, arithmetic operations with elements of B are always done using representations of the elements as products of simple elements. If necessary, such representations are computed. Experienced users can turn this feature off for a braid group B using the procedure `SetForceCFP` with b set to `false`.

`GetElementPrintFormat(B): GrpBrd -> MonStgElt`

Returns a string s indicating the format currently used for printing elements of B. s is one of `"Word"`, `"CFP"`, or `"Both"`.

`SetElementPrintFormat(~B, s): GrpBrd, MonStgElt`

When printing an element of a braid group B, by default both the representation as word in the generators and the representation as product of simple elements in the presentation selected for B are printed. Experienced users can use the procedure `SetElementPrintFormat` for changing the print format. Possible values for s are `"Word"`, `"CFP"`, and `"Both"`.

`NumberOfStrings(B): GrpBrd -> RngIntElt`

Given a braid group B, return the number of strings on which B is defined.

`NumberOfGenerators(B): GrpBrd -> RngIntElt`

`Ngens(B): GrpBrd -> RngIntElt`

Given a braid group B, return the number of Artin generators of B. Note that the number of Artin generators is returned regardless of the presentation selected for B.

82.3 Creating Elements of a Braid Group

This section describes the facilities for creating elements of a braid group.

`Representative(B): GrpBrd -> GrpBrdElt`

`Rep(B): GrpBrd -> GrpBrdElt`

Given a braid group B, return a representative of B.

`Identity(B): GrpBrd -> GrpBrdElt`

`Id(B): GrpBrd -> GrpBrdElt`

`B ! 1: GrpBrd, RngIntElt -> GrpBrdElt`

Given a braid group B, return the identity element of B.

`FundamentalElement(B: parameters): GrpBrd -> GrpBrdElt`

`Presentation: MonStgElt`	`Default:`

Return the fundamental element for the presentation of B indicated by the parameter `Presentation`. Possible values for this parameter are `"Artin"` and `"BKL"`. If the parameter `Presentation` is not used, the fundamental element for the presentation currently selected for B is returned.

`Generators(B: parameters): GrpBrd -> [ GrpBrd ]`

`Presentation: MonStgElt`	`Default:`

Return a sequence containing the generators for the presentation of B indicated by the parameter `Presentation`. Possible values for this parameter are `"Artin"` and `"BKL"`. If the parameter `Presentation` is not used, a sequence containing the generators for the presentation currently selected for B is returned.

`B . i: GrpBrd, RngIntElt -> GrpBrdElt`

Given a braid group B on n strings and an integer i, where $0 < |i| < n$, return the $|i|$-th Artin generator σ_i, if $i > 0$, or its inverse $\sigma_{|i|}^{-1}$, if $i < 0$.

`B . T: GrpBrd, Tup -> GrpBrdElt`

Given a braid group B on n strings and a tuple $T =< r, t >$, where $1 \leq |t| < |r| \leq n$, return the BKL generator $a_{|r|,|t|}$, if $r, t > 0$, or its inverse $a^{-1}_{|r|,|t|}$ otherwise.

`B ! [ i1, ..., ik ]: GrpBrd, [ RngIntElt ] -> GrpBrdElt`

Given a braid group B on n strings and a sequence $[i_1, \cdots, i_k]$ of integers satisfying $0 < |i_j| < n$ $(j = 1, \ldots, k)$, return the element of B given by the product

$$\sigma_{|i_1|}^{\mathrm{sgn}(i_1)} \cdots \sigma_{|i_k|}^{\mathrm{sgn}(i_k)}.$$

`B ! [ T1, ..., Tk ]: GrpBrd, [ Tup ] -> GrpBrdElt`

Given a braid group B on n strings and a sequence $[T_1, \cdots, T_k]$ of tuples satisfying $T_j =< r_j, t_j >$, $1 \leq |t_j| < |r_j| \leq n$ $(j = 1, \ldots, k)$, return the element of B given by the product

$$a^{e_1}_{|r_1|,|t_1|} \cdots a^{e_k}_{|r_k|,|t_k|}$$

where $e_j = 1$ if $r_j, t_j > 0$ and $e_j = -1$ otherwise $(j = 1, \ldots, k)$.

`B ! p: GrpBrd, GrpPermElt -> GrpBrdElt`

Given a braid group B on n strings and a permutation p on n points, return the simple element defined by p in the presentation currently selected for B as new element of B. Note that the result in general depends on the presentation selected for B. Note further that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; attempting to coerce an invalid permutation will result in a runtime error. The function *IsProductOfParallelDescendingCycles* can be used to test whether a given permutation corresponds to a BKL simple element.

`B ! [ p1, ...,pk ]: GrpBrd, [ GrpPermElt ] -> GrpBrdElt`

Given a braid group B on n strings and a sequence $[p_1, \ldots, p_k]$ of permutations on n points, return the product $c_1 \cdots c_k$ as new element of B, where c_j is the simple element defined by p_j in the presentation currently selected for B $(j = 1, \ldots, k)$. Note that the result in general depends on the presentation selected for B. Note further that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; attempting to coerce a sequence containing an invalid permutation will result in a runtime error. The function *IsProductOfParallelDescendingCycles* can be used to test whether a given permutation corresponds to a BKL simple element.

`B ! T: GrpBrd, Tup -> GrpBrdElt`

Given a braid group B on n strings and a tuple $T =< s, l, S, r >$, where s is either the string `"Artin"` or the string `"BKL"`, l and r are integers, and S is a sequence $[p_1, \ldots, p_k]$ of permutations on n points, return the product $D^l c_1 \cdots c_k D^r$ as new element of B, where D is the fundamental element and c_j is the simple element defined by p_j $(j = 1, \ldots, k)$ in the presentation indicated by s. Note that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; if S contains an invalid permutation, a runtime error will result. Whether the elements of a given sequence S correspond to BKL simple elements can be tested using the function *IsProductOfParallelDescendingCycles*.

`IsProductOfParallelDescendingCycles(p): GrpPermElt -> BoolElt`

Given a permutation p on n points, return whether p is a product of parallel descending cycles, that is, whether p defines a simple element in the BKL monoid on n strings.

`Random(B, r, s, m, n: parameters): GrpBrd, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> GrpBrdElt`

`RandomCFP(B, r, s, m, n: parameters): GrpBrd, RngIntElt, RngIntElt -> GrpBrdElt`

`Random(B: parameters): GrpBrd -> GrpBrdElt`

`RandomCFP(B: parameters): GrpBrd -> GrpBrdElt`

Presentation: MonStgElt	Default:

Given a braid group B and integers r, s, m, n, satisfying $r \leq s$ and $0 \leq m \leq n$, a pseudo-random element of B is constructed as follows. Let D be the fundamental element and C the set of simple elements for the presentation indicated by the parameter `Presentation`. First, integers $e \in [r, s]$ and $l \in [m, n]$ are chosen using uniform distributions on these sets. Then, for $i = 1, \ldots, l$, $c_i \in C$ is chosen using a uniform distribution on C and the element $D^e c_1 \cdots c_l$ is returned. If no value is given for the parameter `Presentation`, the presentation selected for B is used. The versions with a single argument are short for `Random(B, 0, 0, 0, 42)`.

`Random(B, m, n: parameters): GrpBrd, RngIntElt, RngIntElt -> GrpBrdElt`

`RandomWord(B, m, n: parameters): GrpBrd, RngIntElt, RngIntElt -> GrpBrdElt`

`RandomWord(B: parameters): GrpBrd -> GrpBrdElt`

Presentation: MonStgElt	Default:

Given a braid group B and two integers $0 \leq m \leq n$, `Random(B, m, n)` returns a pseudo-random element of B constructed as follows. First, a length $l \in [m, n]$ is chosen using a uniform distribution. Then, for $i = 1, \ldots, l$, $g_i \in X \cup X^{-1} \setminus \{g_{i-1}^{-1}\}$ is chosen using a uniform distribution on this set. Here, X is the set of generators of the presentation indicated by the parameter `Presentation` and X^{-1} is the set of generator inverses. If no value is given for the parameter `Presentation`, the presentation selected for B is used. The signature `RandomWord(B)` is short for `RandomWord(B, 0, 42)`.

Example: Creating Braid Group Elements (ex-ddc2bb)

We construct the braid group B on 6 strings and the symmetric group S on 6 points.

```
> S := Sym(6);
> B := BraidGroup(6);
> B;
GrpBrd : B on 6 strings
```

By default, B is created using the Artin presentation.

```
> GetPresentation(B);
Artin
```

We now define the fundamental element with respect to the BKL presentation of B and print this element with respect to the presentation currently used for B, that is, with respect to the Artin presentation. Note that both a word in the Artin generators and a representation of the element in terms of Artin simple elements are printed.

```
> D_BKL := FundamentalElement(B : Presentation := "BKL");
> D_BKL;
B.5 * B.4 * B.3 * B.2 * B.1
<Artin, 0, [
    (1, 6, 5, 4, 3, 2)
], 0>
> GetElementPrintFormat(B);
Both
```

We print the BKL generator $a_{3,1}$.

```
> B.<3,1>;
B.2 * B.1 * B.2^-1
<Artin, 0, [
    (1, 3, 2),
```

(continues on next page)

(continued from previous page)

```
    (1, 6)(2, 5, 3, 4)
], -1>
```

Next we change the format for printing elements of B using the function *SetElementPrintFormat* so that only a representation in terms of simple elements is printed.

```
> SetElementPrintFormat(~B, "CFP");
```

We now define and print several elements of B, illustrating the use of some of the functions described in the previous section. First we create a pseudo-random element of B as product of 3 random simple elements for the Artin presentation.

```
> u := Random(B, 0, 0, 3, 3);
> u;
<Artin, 0, [
    (1, 6)(3, 5, 4),
    (1, 3)(2, 6)(4, 5),
    (2, 3)
], 0>
```

Next we define an element of B by a product of simple elements for the BKL presentation using the coercion operator '`B ! T`'. Note that printing of this element is still done with respect to the Artin presentation.

```
> v := B ! <"BKL", 0, [ S | (1,6)(3,5,4), (1,3)(4,5)], 0>;
> v;
<Artin, 0, [
    (1, 6, 5, 4, 3, 2),
    (1, 2, 6)(3, 5),
    (2, 4, 5, 6),
    (1, 6)(2, 4, 3, 5)
], -2>
```

Finally, we look at the simple elements defined by the permutations $p = (1,3)(4,2)$ and $q = (1,4,3)$ on 6 points.

```
> p := S ! (1,3)(4,2);
> q := S ! (1,4,3);
```

Creating the simple elements for the Artin presentation defined by p and q is straightforward using the coercion operator '`B ! p`'.

```
> p_Artin := B!p;
> p_Artin;
<Artin, 0, [
    (1, 3)(2, 4)
], 0>

> q_Artin := B!q;
> q_Artin;
<Artin, 0, [
    (1, 4, 3)
], 0>
```

We now change the presentation used for B to the BKL presentation. Note that this also changes the presentation with respect to which elements are printed.

```
> SetPresentation(~B, "BKL");
> GetPresentation(B);
BKL
```

The attempt to define a simple element for the BKL presentation using the permutation p fails.

```
> p_BKL := B!p;

>> p_BKL := B!p;
            ^
Runtime error in '!': Illegal coercion
LHS: GrpBrd
RHS: GrpPermElt
```

We should have been more careful: using the function *IsProductOfParallelDescendingCycles* we see that p is not a product of parallel descending cycles and hence does not define a simple element for the BKL presentation.

```
> IsProductOfParallelDescendingCycles(p);
false
```

q, on the other hand, does define a simple element for the BKL presentation and we can coerce q to an element of B using the operator '`B ! p`'.

```
> IsProductOfParallelDescendingCycles(q);
true
> q_BKL := B!q;
> q_BKL;
<BKL, 0, [
    (1, 4, 3)
], 0>
```

Note however, that the simple element for the BKL presentation defined by q and the simple element for the Artin presentation defined by q are different elements of B! (The comparison operator `u eq v` is described in Section *Boolean Predicates for Elements*.)

```
> q_BKL eq q_Artin;
false
```

The representations of the Artin simple elements defined by p and q in terms of BKL simple elements have no obvious connection to p and q, respectively.

```
> p_Artin;
<BKL, 0, [
    (1, 3, 2),
    (2, 4, 3)
], 0>

> q_Artin;
<BKL, 0, [
    (1, 4, 3, 2),
    (2, 3)
], 0>
```

82.4 Working with Elements of a Braid Group

82.4.1 Accessing Information

This sections describes how the internal representations of an element and a number of basic invariants can be accessed.

`Parent(u): GrpBrdElt -> GrpBrd`

Given an element u of a braid group B, return the parent group of u, that is B.

`# u: GrpBrdElt -> RngIntElt`

Given an element u of a braid group B, return the length of the representing word in the generators corresponding to the presentation selected for B. Note that this is not an invariant of u.

`CanonicalFactorRepresentation(u: parameters): GrpBrdElt -> Tup`

`CFP(u: parameters): GrpBrdElt -> Tup`

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return a tuple $T =< s, l, S, r >$ describing the representation in terms of simple elements for the presentation indicated by the value of the parameter `Presentation`. If no value for `Presentation` is given, the presentation selected for B is used. The interpretation of the components of T is as follows: s is a string, either equal to `"Artin"` or equal to `"BKL"` indicating the presentation, l and r are integers, and S is a sequence $[p_1, \ldots, p_k]$ of permutations on n points, such that

$$D^l c_1 \cdots c_k D^r$$

is the representation of u in terms of simple elements, where D is the fundamental element and c_j is the simple element defined by p_j $(j = 1, \ldots, k)$ in the presentation indicated by s.

`WordToSequence(u: parameters): GrpBrdElt -> SeqEnum`

`ElementToSequence(u: parameters): GrpBrdElt -> SeqEnum`

`Eltseq(u: parameters): GrpBrdElt -> SeqEnum`

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return a sequence describing the representing word in the generators corresponding to the presentation indicated by the value of the parameter `Presentation`. If no value for `Presentation` is given, the presentation selected for B is used. For a representing word

$$\sigma_{i_1}^{e_1} \cdots \sigma_{i_k}^{e_k}$$

in the Artin generators with $0 < i_j < n$ and $e_j \in \{-1, 1\}$ for $j = 1, \ldots, k$, the sequence of integers

$$[e_1 i_1, \ldots, e_k i_k]$$

is returned. For a representing word

$$a_{r_1,t_1}^{e_1} \ldots a_{r_k,t_k}^{e_k}$$

in the BKL generators with $1 \leq t_j < r_j \leq n$ and $e_j \in \{-1, 1\}$ for $j = 1, \ldots, k$, the sequence of tuples

$$[< e_1 r_1, e_1 t_1 >, \ldots, < e_k r_k, e_k t_k >]$$

is returned.

InducedPermutation(u): GrpBrdElt -> GrpPermElt

Given an element u of a braid group B on n strings, return the permutation on n points induced by u acting on the strings on which B is defined.

In the following descriptions of the functions *CanonicalLength*, *Infimum*, and *Supremum*, let D be the fundamental element and let $D^l c_1 \cdots c_k$ be the left normal form of the element u of the braid group B in the presentation indicated by the value of the parameter **Presentation**. If no value for **Presentation** is given, the presentation selected for B is used.

CanonicalLength(u: parameters): GrpBrdElt -> RngIntElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the canonical length k of u for the appropriate presentation of B. The argument is converted into left normal form.

Infimum(u: parameters): GrpBrdElt -> RngIntElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the infimum l of u for the appropriate presentation of B. The argument is converted into left normal form.

Supremum(u: parameters): GrpBrdElt -> RngIntElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the supremum $l + k$ of u for the appropriate presentation of B. The argument is converted into left normal form.

In the descriptions of the functions *SuperSummitCanonicalLength*, *SuperSummitInfimum*, and *SuperSummitSupremum* which follow, let D be the fundamental element and let $D^l c_1 \cdots c_k$ be the normal form of a representative of the super summit set the element u of the braid group B with respect to the presentation indicated by the value of the parameter **Presentation**. If no value for **Presentation** is given, the presentation selected for B is used.

SuperSummitCanonicalLength(u: parameters): GrpBrdElt -> RngIntElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the canonical length k of a representative of the super summit set of u with respect to the appropriate presentation of B, that is, the minimal canonical length among all conjugates of u. The argument is converted into left normal form.

SuperSummitInfimum(u: parameters): GrpBrdElt -> RngIntElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the infimum l of a representative of the super summit set of u with respect to the appropriate presentation of B, that is, the maximal infimum among all conjugates of u. The argument is converted into left normal form.

SuperSummitSupremum(u: parameters): GrpBrdElt -> RngIntElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the supremum $l + k$ of a representative of the super summit set of u with respect to the appropriate presentation of B, that is, the minimal supremum among all conjugates of u. The argument is converted into left normal form.

Example: Access Functions for BKL Braids (ex-0cc874)

We define the braid group B on 6 strings using the BKL presentation and create an element u as a random word of length between 5 and 10 in the BKL generators.

```
> B := BraidGroup(6 : Presentation := "BKL");
> u := RandomWord(B, 5, 10);
```

The parent group of u can be accessed using the function *Parent*.

```
> Parent(u);
GrpBrd : B on 6 strings
```

As word in the BKL generators, u has length 5. We define a sequence describing the representation of u as word in the BKL generators.

```
> #u;
5
> seq_BKL := WordToSequence(u);
> seq_BKL;
[ <5, 1>, <5, 2>, <5, 4>, <4, 1>, <2, 1> ]
```

When we ask for a representation as word in the Artin generators, such a representation is created automatically.

```
> seq_Artin := WordToSequence(u : Presentation := "Artin");
> seq_Artin;
[ 4, 3, 2, 1, 2, 1, -2, -3, 1 ]
```

We now define the permutation p induced by u on the strings on which B is defined.

```
> p := InducedPermutation(u);
> p;
(4, 5)
```

The representation of u in terms of simple elements for the Artin presentation can be obtained using the function *CanonicalFactorRepresentation*.

```
> CanonicalFactorRepresentation(u : Presentation := "Artin");
<Artin, 0, [
    (1, 5, 4, 3),
    (1, 2),
    (1, 6)(2, 5, 3),
    (5, 6)
], -1>
```

We now compute the canonical lengths of u with respect to the Artin presentation and with respect to the BKL presentation. Note that these lengths are different.

```
> CanonicalLength(u : Presentation := "Artin");
4
> CanonicalLength(u : Presentation := "BKL");
3
```

Finally, we compute for both presentations the canonical lengths of a super summit representative of u.

```
> SuperSummitCanonicalLength(u : Presentation := "Artin");
2
```

(continues on next page)

(continued from previous page)

```
> SuperSummitCanonicalLength(u : Presentation := "BKL");
3
```

Obviously, u does not belong to its super summit set with respect to the Artin presentation. (We cannot tell for the BKL presentation from the information we have computed.)

82.4.2 Computing Normal Forms of Elements

This section describes functions and procedures for computing various normal forms of elements of a braid group B. All normal forms are defined in terms of representations of elements as products of simple elements and depend on the presentation of B which is used. The functions documented in this section all accept a parameter `Presentation` which can be used to specify the presentation of B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"` If no value is given for `Presentation`, the presentation selected for B is used.

LeftNormalForm(u: parameters): GrpBrdElt -> GrpBrdElt

NormalForm(u: parameters): GrpBrdElt -> GrpBrdElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return a new element of B defined by the left normal form of u with respect to the indicated presentation.

LeftNormalForm(~u: parameters): GrpBrdElt

NormalForm(~u: parameters): GrpBrdElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, bring u into left normal form with respect to the indicated presentation.

RightNormalForm(u: parameters): GrpBrdElt -> GrpBrdElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return a new element of B defined by the right normal form of u with respect to the indicated presentation.

RightNormalForm(~u: parameters): GrpBrdElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, bring u into right normal form with respect to the indicated presentation.

LeftMixedCanonicalForm(u: parameters): GrpBrdElt -> Tup, Tup

MixedCanonicalForm(u: parameters): GrpBrdElt -> Tup, Tup

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return two tuples T_1 and T_2 defining products $v_1 \cdots v_k$ and $w_1 \cdots w_l$, respectively, of simple elements for the indicated presentation, such that $v_1 \cdots v_k$ and $w_1 \cdots w_l$ are in left normal form, the left-gcd of v_1 and w_1 is trivial and

$$u = (v_1 \cdots v_k)^{-1}(w_1 \cdots w_l).$$

See the entry for *CanonicalFactorRepresentation* for a description of the tuple format. Note that the tuples can be coerced into elements of B using the coercion operator '`B ! T`'.

`RightMixedCanonicalForm(u: parameters): GrpBrdElt -> Tup, Tup`

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return two tuples T_1 and T_2 defining products $v_1 \cdots v_k$ and $w_1 \cdots w_l$, respectively, of simple elements for the indicated presentation, such that $v_1 \cdots v_k$ and $w_1 \cdots w_l$ are in right normal form, the right-gcd of v_1 and w_1 is trivial and

$$u = (v_1 \cdots v_k)(w_1 \cdots w_l)^{-1}.$$

See the entry for *CanonicalFactorRepresentation* for a description of the tuple format. Note that the tuples can be coerced into elements of B using the coercion operator '`B ! T`'.

Example: Normal Forms of Braids `(ex-9c32e6)`

We define the braid group B on 6 strings using the Artin presentation, set the print format for elements to `"CFP"` and define an element u of B.

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
>
> u := B ! <"Artin",
>          0,
>          [ Sym(6) | (1,6)(3,5,4), (1,3)(2,6)(4,5), (2,3)],
>          0>;
> u;
<Artin, 0, [
    (1, 6)(3, 5, 4),
    (1, 3)(2, 6)(4, 5),
    (2, 3)
], 0>
```

We compute and print the left normal form of u with respect to the Artin presentation of B.

```
> u_Artin := LeftNormalForm(u);
> u_Artin;
<Artin, 1, [
    (1, 2, 6, 5, 4, 3),
    (2, 3)
], 0>
```

We now compute the left normal form of u with respect to the BKL presentation of B. Since elements are printed with respect to the presentation selected for the parent group, that is, in the Artin presentation in our example, we use the function *CanonicalFactorRepresentation* to print the representation in terms of simple elements for the BKL presentation.

```
> u_BKL := LeftNormalForm(u : Presentation := "BKL");
> CFP(u_BKL : Presentation := "BKL");
<BKL, 2, [
    (2, 6, 5, 4, 3),
    (2, 6, 5, 4),
    (2, 6, 5),
    (2, 6),
    (2, 3)
], 0>
```

We define another element v of B in left normal form.

```
> v := LeftNormalForm(B.5*B.2^-2*B.4*B.3^-1*B.5^-1*B.3^-1*B.5);
> v;
<Artin, -3, [
    (1, 6)(2, 4, 3, 5),
    (1, 2, 6)(3, 5),
    (1, 6, 2, 5)(3, 4),
    (3, 5)(4, 6)
], 0>
```

As can easily be read off the representation of v in terms of simple elements which is in left normal form, v has infimum -3, canonical length 4, and supremum 1 with respect to the Artin presentation.

```
> Infimum(v);
-3
> CanonicalLength(v);
4
> Supremum(v);
1
```

Note that infimum, canonical length, and supremum of an element can also be obtained from its right normal form.

```
> RightNormalForm(v);
<Artin, 0, [
    (4, 6, 5),
    (1, 6)(2, 5, 3, 4),
    (1, 6)(2, 4, 5),
    (1, 6)(2, 5)
], -3>
```

82.4.3 Arithmetic Operators and Functions for Elements

This section describes the basic arithmetic operations for elements of a braid group. Strictly speaking, all functions should be considered as functions on *representatives* of elements, that is, words in the generators or products of simple elements. Unless stated otherwise, arithmetic operations with elements of a braid group B are performed using representations with respect to the presentation selected for B. This presentation can be changed using the function *SetPresentation*. By default, arithmetic operations with elements of B are performed using representations in terms of simple elements; such representations are created if necessary. Experienced users can change this behaviour, if desired, using the function *SetForceCFP*. The complexity of all basic arithmetic operations is linear in the length of the representations of the input elements. No normalisations are performed automatically, as doing so would restrict the user's control of operations with elements. It is, however, recommended to use the function *NormalForm* or its procedural version in time critical situations to limit the length of representations of elements; see Example *Arithmetic with Braids in Normal Form*.

`u * v: GrpBrdElt, GrpBrdElt -> GrpBrdElt`

Given elements u and v belonging to the same braid group B, return the product uv as a new element of B.

`u *:= v: GrpBrdElt, GrpBrdElt`

Given elements u and v belonging to the same braid group B, replace u with the product uv.

`u / v: GrpBrdElt, GrpBrdElt -> GrpBrdElt`

Given elements u and v belonging to the same braid group B, return the product uv^{-1} as a new element of B.

`u /:= v: GrpBrdElt, GrpBrdElt`

Given elements u and v belonging to the same braid group B, replace u with the product uv^{-1}.

`u ^ n: GrpBrdElt, RngIntElt -> GrpBrdElt`
Given an element u of a braid group B and an integer n, return the power u^n as a new element of B.

`u ^:= n: GrpBrdElt, RngIntElt`
Given an element u of a braid group B and an integer n, replace u with the power u^n.

`u ^ v: GrpBrdElt, GrpBrdElt -> GrpBrdElt`
Given elements u and v belonging to the same braid group B, return the conjugate $u^v = v^{-1}uv$ as a new element of B.

`u ^:= v: GrpBrdElt, GrpBrdElt`
Given elements u and v belonging to the same braid group B, replace u with the conjugate $u^v = v^{-1}uv$.

`Inverse(u): GrpBrdElt -> GrpBrdElt`
Given an element u of a braid group B, return its inverse u^{-1} as a new element of B.

`Inverse(~u): GrpBrdElt`
Given an element u of a braid group B, replace u with its inverse u^{-1}.

`LeftConjugate(u, v): GrpBrdElt, GrpBrdElt -> GrpBrdElt`
Given elements u and v belonging to the same braid group B, return the "left conjugate" vuv^{-1} as a new element of B.

`LeftConjugate(~u, v): GrpBrdElt, GrpBrdElt -> GrpBrdElt`
Given elements u and v belonging to the same braid group B, replace u with the "left conjugate" vuv^{-1}.

`LeftDiv(u, v): GrpBrdElt, GrpBrdElt -> GrpBrdElt`
Given elements u and v belonging to the same braid group B, return the product $u^{-1}v$ as a new element of B.

`LeftDiv(u, ~v): GrpBrdElt, GrpBrdElt -> GrpBrdElt`
Given elements u and v belonging to the same braid group B, replace v with the product $u^{-1}v$.

The following functions *Cycle* and *Decycle* accept a parameter `Presentation` which can be set either to `"Artin"` or to `"BKL"`. The results of the cycling and decycling operations are defined in terms of the left normal form $D^l c_1 \cdots c_k$ of the argument $u \in B$ in terms of simple elements for a presentation of B and the results in general depend on the presentation used. The results of these functions are returned in left normal form. If no value for the parameter `Presentation` is given, the presentation selected for the parent group of the argument will be used.

`Cycle(u: parameters): GrpBrdElt -> GrpBrdElt`

Presentation: MonStgElt	Default:

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, return the result of a cycling operation on u, that is,

$$u^{(c_1^{D^{-l}})}$$

as new element of B in left normal form.

`Cycle(~u: parameters): GrpBrdElt`

Presentation: MonStgElt	Default:

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, replace u by the result of a cycling operation on u, that is, by

$$u^{(c_1^{D^{-l}})}$$

in left normal form.

`Decycle(u: parameters): GrpBrdElt -> GrpBrdElt`

Presentation: MonStgElt	Default:

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, return the result of a decycling operation on u, that is,

$$u^{(c_k^{-1})}$$

as new element of B in left normal form.

`Decycle(~u: parameters): GrpBrdElt`

Presentation: MonStgElt	Default:

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, replace u by the result of a decycling operation on u, that is, by

$$u^{(c_k^{-1})}$$

in left normal form.

Example: Arithmetic with Braids in Normal Form (ex-be378e)

We illustrate the importance of limiting the length of representations of elements using the function *NormalForm* when performing a sequence of arithmetic operations on an element. Consider the following computation in the braid group B on 6 strings. Starting with an element w, we repeatedly replace w by the product ww^{σ_1} where σ_1 is the first Artin generator of B. A naive way of implementing this computation would be as follows.

```
> B := BraidGroup(6);
> u := B.5*B.2^-2*B.4*B.3^-1;
> v := B.1;
> N := 14;
>
> T := Cputime();
> w := u;
> for i := 1 to N do
>    w := w * w^v;
> end for;
```

This, however, yields an extremely complicated representation for the result; the representation in terms of simple elements has the length 114686.

```
> #CFP(w)[3];
114686
```

Performing subsequent computations with the result, for example computing its normal form, is very expensive.

```
> NormalForm(~w);
> print "total time used: ", Cputime()-T;
total time used:  149.229
```

One might be tempted to solve this problem by working only with elements in normal form, that is, by bringing every result of an arithmetic operation into normal form after computing it. Using this approach, computing the result of the above iteration is indeed much faster.

```
> T := Cputime();
> w := u;
> for i := 1 to N do
>    t := w^v;
>    NormalForm(~t);
>    w := w * t;
>    NormalForm(~w);
> end for;
> print "total time used: ", Cputime()-T;
total time used:  0.53
```

However, this strategy is not optimal either. For the above example, the optimal performance is obtained if the result is normalised every third pass through the iteration.

```
> T := Cputime();
> w := u;
> for i := 1 to N do
>    w := w * w^v;
>    if i mod 3 eq 0 then
>       NormalForm(~w);
>    end if;
> end for;
> NormalForm(~w);
> print "total time used: ", Cputime()-T;
total time used:  0.171
```

Unfortunately, the frequency of normalisation giving best results depends heavily on the situation, that is, both on the arithmetic operations and on the characteristics of the arguments. As a rule of thumb, the effects of normalising results too frequently are less of a problem than normalising results not often enough or not at all.

82.4.4 Boolean Predicates for Elements

This section describes the tests for membership, equality, and partial orderings which are available for elements of a braid group B. Unless stated otherwise, all computations are performed in the presentation selected for B or in the presentation specified by the value of the parameter `Presentation`, either `"Artin"` or `"BKL"` if a value for this parameter is given.

`u in B: GrpBrdElt, GrpBrd -> BoolElt`

Given an element u of a braid group and a braid group B, return `true` if $u \in B$ and `false` otherwise.

`u notin B: GrpBrdElt, GrpBrd -> BoolElt`

Given an element u of a braid group and a braid group B, return `false` if $u \in B$ and `true` otherwise.

`IsEmptyWord(u: parameters): GrpBrdElt -> BoolElt`

```
Presentation: MonStgElt                    Default:
```

Given an element u of a braid group, return `true` if u is the represented by the empty word in the specified presentation and `false` otherwise.

`AreIdentical(u, v: parameters): GrpBrdElt, GrpBrdElt -> BoolElt`

```
Presentation: MonStgElt                    Default:
```

Given elements u and v belonging to the same braid group B, return `true` if u and v are represented by identical words in the specified presentation and `false` otherwise.

`IsSimple(u: parameters): GrpBrdElt -> BoolElt`

Presentation: MonStgElt	Default:

Given an element u of a braid group, return `true` if u is a simple element with respect to the specified presentation and `false` otherwise. The argument is converted into normal form.

`IsSuperSummitRepresentative(u: parameters): GrpBrdElt -> BoolElt`

Presentation: MonStgElt	Default:

Given an element u of a braid group, return `true` if u is an element of its super summit set with respect to the specified presentation and `false` otherwise. The argument is converted into normal form.

`IsUltraSummitRepresentative(u: parameters): GrpBrdElt -> BoolElt`

Presentation: MonStgElt	Default:

Given an element u of a braid group, return `true` if u is an element of its ultra summit set with respect to the specified presentation and `false` otherwise. The argument is converted into normal form.

`IsIdentity(u: parameters): GrpBrdElt -> BoolElt`
`IsId(u: parameters): GrpBrdElt -> BoolElt`

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return `true` if u is the identity element of B and `false` otherwise. The argument is converted into normal form.

`u eq v: GrpBrdElt, GrpBrdElt -> BoolElt`

Given elements u and v belonging to the same braid group B, return `true` if $u = v$ and `false` otherwise. Both arguments are converted into normal form.

`u ne v: GrpBrdElt, GrpBrdElt -> BoolElt`

Given elements u and v belonging to the same braid group B, return `false` if $u = v$ and `true` otherwise. Both arguments are converted into normal form.

`u le v: GrpBrdElt, GrpBrdElt -> BoolElt`
`IsLE(u, v: parameters): GrpBrdElt, GrpBrdElt -> BoolElt`
`IsLe(u, v: parameters): GrpBrdElt, GrpBrdElt -> BoolElt`

Presentation: MonStgElt	Default:

Given elements u and v belonging to the same braid group B, return `true` if $u \preceq v$, that is, if $u^{-1}v$ is a positive element, with respect to the specified presentation and `false` otherwise. Note that the parameter `Presentation` is not available for the operator version of this predicate.

`u ge v: GrpBrdElt, GrpBrdElt -> BoolElt`
`IsGE(u, v: parameters): GrpBrdElt, GrpBrdElt -> BoolElt`
`IsGe(u, v: parameters): GrpBrdElt, GrpBrdElt -> BoolElt`

Presentation: MonStgElt	Default:

Given elements u and v belonging to the same braid group B, return `true` if $u \succeq v$, that is, if uv^{-1} is a positive element, with respect to the specified presentation and `false` otherwise. Note that the parameter `Presentation` is not available for the operator version of this predicate.

`IsConjugate(u, v: parameters): GrpBrdElt, GrpBrdElt -> BoolElt, GrpBrdElt`

`Presentation: MonStgElt` `Default:`

Given elements u and v belonging to the same braid group B, return `true` and an element $c \in B$ satisfying $u^c = v$ if u and v are conjugate and return `false` otherwise. The function first computes representatives u_s and v_s of the ultra summit sets of u and v, respectively, with respect to the specified presentation. If this does not prove that the elements are not conjugate, the function tries to compute elements of the ultra summit set of u until either the element v_s is found, proving that u and v are conjugate, or the ultra summit set of u is seen not to contain v_s, proving that u and v are not conjugate. See Example *Interactive Braid Conjugacy Testing* for a more detailed description. Note that testing elements for conjugacy is a hard problem and may require significant amounts of memory and CPU time.

Example: Boolean Predicates for Braids (ex-58fe4b)

We define the braid group B on 6 strings using the Artin presentation and set the print format for elements to `"CFP"`.

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
```

We create pseudo-random elements of B until we find an element u which is contained in its super summit set with respect to the Artin presentation of B.

```
> repeat
>    u := Random(B, 5, 10);
> until IsSuperSummitRepresentative(u);
> NormalForm(u);
<Artin, -2, [
    (1, 5)(2, 3, 6),
    (1, 5, 6, 3, 2, 4),
    (2, 6)(3, 4, 5)
], 0>
```

u is not contained in its super summit set with respect to the BKL presentation of B, showing that the super summit set of an element in general depends on the presentation with respect to which it is defined.

```
> IsSuperSummitRepresentative(u : Presentation := "Artin");
true
> IsSuperSummitRepresentative(u : Presentation := "BKL");
false
```

This example shows that the Artin presentation and the BKL presentation give rise to distinct partial orderings on B. $\sigma_1^{-1}\sigma_2\sigma_1$ has negative infimum with respect to the Artin presentation and hence is not a positive element with respect to this presentation.

```
> Infimum(B.1^-1*B.2*B.1 : Presentation := "Artin");
-1
```

Consequently, $\sigma_1 \not\preceq \sigma_2\sigma_1$ in the partial ordering defined with respect to the Artin presentation.

```
> B.1 le B.2*B.1;
false
```

We can also use the function version to check this.

```
> IsLE(B.1, B.2*B.1 : Presentation := "Artin");
false
```

However, $\sigma_1^{-1}\sigma_2\sigma_1$ is equal to the BKL generator $a_{3,1}$ and hence is, in particular, a positive element with respect to the BKL presentation. in the BKL generators.

```
> B.1^-1*B.2*B.1 eq B.<3,1>;
true
```

Hence, $\sigma_1 \preceq \sigma_2\sigma_1$ in the partial ordering defined with respect to the BKL presentation.

```
> IsLE(B.1, B.2*B.1 : Presentation := "BKL");
true
```

We change the print format for elements of B so that only words in the Artin generators are printed.

```
> SetElementPrintFormat(~B, "Word");
```

Inducing permutations with different cycle structure, σ_1 and $\sigma_1\sigma_2$ cannot be conjugate in B.

```
> InducedPermutation(B.1);
(1, 2)
> InducedPermutation(B.2*B.1);
(1, 2, 3)
> IsConjugate(B.1, B.2*B.1);
false
```

σ_1 and σ_2, however, are conjugate in B. We compute a conjugating element c.

```
> res, c := IsConjugate(B.1, B.2);
> res;
true
> NormalForm(c);
B.2 * B.1
```

c, as desired, conjugates σ_1 to σ_2.

```
> B.1^c eq B.2;
true
```

82.4.5 Lattice Operations

This section describes the functions available for computing lattice operations, least common multiple and greatest common divisor, for elements of a braid group B. The results of all lattice operations depend on the presentation used for B and on the partial ordering considered. The functions documented in this section all accept a parameter `Presentation` which can be used to specify the presentation of B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"`. If no value is given for `Presentation`, the presentation selected for B is used.

LeftGCD(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt

LeftGcd(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt

LeftGreatestCommonDivisor(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt

GCD(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt

Gcd(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt

GreatestCommonDivisor(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt

```
Presentation: MonStgElt                    Default:
```

Given elements u and v belonging to the same braid group B, return the left-gcd of u and v, that is, the with respect to $\preceq$ maximal element d of B satisfying $d \preceq u$ and $d \preceq v$. Here, $\preceq$ is the partial ordering on B defined as follows: $a \preceq b$ iff $a^{-1}b$ is representable as a positive word in the specified presentation of B.

```
RightGCD(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
RightGcd(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
RightGreatestCommonDivisor(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
```

```
Presentation: MonStgElt                    Default:
```

Given elements u and v belonging to the same braid group B, return the right-gcd of u and v, that is, the with respect to $\succeq$ maximal element d of B satisfying $u \succeq d$ and $v \succeq d$. Here, $\succeq$ is the partial ordering on B defined as follows: $a \succeq b$ iff ab^{-1} is representable as a positive word in the specified presentation of B.

```
LeftGCD(S: parameters): Setq -> GrpBrdElt
LeftGcd(S: parameters): Setq -> GrpBrdElt
LeftGreatestCommonDivisor(S: parameters): Setq -> GrpBrdElt
GCD(S: parameters): Setq -> GrpBrdElt
Gcd(S: parameters): Setq -> GrpBrdElt
GreatestCommonDivisor(S: parameters): Setq -> GrpBrdElt
```

```
Presentation: MonStgElt                    Default:
```

Given a set or a sequence S containing elements of a braid group B, return the left-gcd of the elements of S, that is, the with respect to $\preceq$ maximal element d of B satisfying $d \preceq s$ for all $s \in S$, where $\preceq$ is defined as above.

```
RightGCD(S: parameters): Setq -> GrpBrdElt
RightGcd(S: parameters): Setq -> GrpBrdElt
RightGreatestCommonDivisor(S: parameters): Setq -> GrpBrdElt
```

```
Presentation: MonStgElt                    Default:
```

Given a set or a sequence S containing elements of a braid group B, return the right-gcd of the elements of S, that is, the with respect to $\succeq$ maximal element d of B satisfying $s \succeq d$ for all $s \in S$, where $\succeq$ is defined as above.

```
LeftLCM(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
LeftLcm(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
LeftLeastCommonMultiple(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
LCM(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
Lcm(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
LeastCommonMultiple(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
```

```
Presentation: MonStgElt                    Default:
```

Given elements u and v belonging to the same braid group B, return the left-lcm of u and v, that is, the with respect to $\preceq$ minimal element d of B satisfying $u \preceq d$ and $v \preceq d$, where $\preceq$ is defined as above.

```
RightLCM(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
RightLcm(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
RightLeastCommonMultiple(u, v: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt
```

Presentation: MonStgElt Default:

Given elements u and v belonging to the same braid group B, return the right-lcm of u and v, that is, the with respect to $\succeq$ minimal element d of B satisfying $d \succeq u$ and $d \succeq v$, where $\succeq$ is defined as above.

LeftLCM(S: parameters): Setq -> GrpBrdElt

LeftLcm(S: parameters): Setq -> GrpBrdElt

LeftLeastCommonMultiple(S: parameters): Setq -> GrpBrdElt

LCM(S: parameters): Setq -> GrpBrdElt

Lcm(S: parameters): Setq -> GrpBrdElt

LeastCommonMultiple(S: parameters): Setq -> GrpBrdElt

Presentation: MonStgElt Default:

Given a set or a sequence S containing elements of a braid group B, return the left-gcd of the elements of S, that is, the with respect to $\preceq$ minimal element d of B satisfying $s \preceq d$ for all $s \in S$, where $\preceq$ is defined as above.

RightLCM(S: parameters): Setq -> GrpBrdElt

RightLcm(S: parameters): Setq -> GrpBrdElt

RightLeastCommonMultiple(S: parameters): Setq -> GrpBrdElt

Presentation: MonStgElt Default:

Given a set or a sequence S containing elements of a braid group B, return the right-lcm of the elements of S, that is, the with respect to $\succeq$ minimal element d of B satisfying $d \succeq s$ for all $s \in S$, where $\succeq$ is defined as above.

Example: Lattice Operations on Braids (ex-3e6267)

We define the braid group B on 6 strings.

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
```

For both Artin and BKL presentation, the fundamental element is the (left or right) least common multiple of the generators. We check this for the Artin presentation…

```
> D_Artin := LeftLCM({B.i : i in [1..NumberOfGenerators(B)]});
> D_Artin eq FundamentalElement(B);
true
> D_Artin eq RightLCM({B.i : i in [1..NumberOfGenerators(B)]});
true
```

…and for the BKL presentation.

```
> idx := { <r,t> : r,t in {1..NumberOfStrings(B)} | r gt t };
> D_BKL := LeftLCM({B.T : T in idx} : Presentation := "BKL");
> D_BKL eq FundamentalElement(B : Presentation := "BKL");
true
> D_BKL eq RightLCM({B.T : T in idx} : Presentation := "BKL");
true
```

In general, left and right least common multiple of elements are different.

```
> LeftLCM(B.1,B.1*B.2) eq RightLCM(B.1,B.1*B.2);
false
```

For both Artin and BKL presentation, the following hold. Let D denote the fundamental element.

- The simple elements are those positive elements s satisfying $s \preceq D$ (or $D \succeq s$).
- A product $u_1 \cdots u_r$ of simple elements is in left normal form, if and only if the left greatest common divisor of $u_i^{-1}D$ and u_{i+1} is trivial for all $i = 1, \ldots, r-1$. We illustrate this for the Artin presentation.

```
> D := FundamentalElement(B);
> forall{ s : s in Sym(6) | B!1 le B!s and B!s le D };
true
> forall{ s : s in Sym(6) | D ge B!s and B!s ge B!1 };
true
```

We create an element u as product of random simple elements.

```
> u := Random(B, 0, 0, 3, 5);
> u;
<Artin, 0, [
    (1, 5, 2)(3, 6),
    (1, 6, 5, 3),
    (1, 6, 5, 3, 2)
], 0>
```

We define a sequence of elements of B, containing the simple elements of the above representation of u using the function *CanonicalFactorRepresentation* and the coercion operator `B ! p`.

```
> cfu := [ B!x : x in CFP(u)[3] ];
```

This representation is not in left normal form, as the above condition is violated for $i = 1$.

```
> IsId(LeftGCD(cfu[1]^-1*D, cfu[2]));
false
```

We now bring u into left normal form and extract again the sequence of simple elements.

```
> n := NormalForm(u);
> n;
<Artin, 0, [
    (1, 5, 3, 6, 2),
    (1, 6, 3, 2, 5)
], 0>
> cfn := [ B!x : x in CFP(n)[3] ];
```

This time, the above condition is satisfied.

```
> IsId(LeftGCD(cfn[1]^-1*D, cfn[2]));
true
```

82.4.6 Invariants of Conjugacy Classes

This section describes the functions for computing the set of positive conjugates, the super summit set, and the ultra summit set for an element of a braid group B as defined in Section *Conjugacy Testing and Conjugacy Search*, as well as related Magma functions. All the class invariants in general depend on the presentation of B used for their definition. Many functions documented in this section accept a parameter `Presentation` which can be used to specify the presentation of B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"`. If no value is given for `Presentation`, the presentation selected for B is used. For any given element $u \in B$, all the invariants defined in Section *Conjugacy Testing and Conjugacy Search* are finite and can be computed in principle. In practice, however, computations may fail because the sets can get

very large with increasing canonical length of u or with increasing braid index of B. This is in particular the case for sets of positive conjugates and for super summit sets.

PositiveConjugates(u: parameters): GrpBrdElt -> SetIndx

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return an indexed set containing the conjugates of u which can be represented as positive words in the specified presentation of B.

SuperSummitRepresentative(u: parameters): GrpBrdElt -> GrpBrdElt, GrpBrdElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return an element u_s of the super summit set of u with respect to the specified presentation of B and an element c of B satisfying $u^c = u_s$. Note that u_s is a positive conjugate of u, if u_s has non-negative infimum and that u does not have any positive conjugates if the infimum of u_s is negative.

SuperSummitSet(u: parameters): GrpBrdElt -> SetIndx

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the super summit set of u with respect to the specified presentation as indexed set of elements of B.

UltraSummitRepresentative(u: parameters): GrpBrdElt -> GrpBrdElt, GrpBrdElt

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return an element u_s of the ultra summit set of u with respect to the specified presentation of B and an element c of B satisfying $u^c = u_s$. Note that u_s is an element of the super summit set of u, that u_s is a positive conjugate of u, if u_s has non-negative infimum, and that u does not have any positive conjugates if the infimum of u_s is negative.

UltraSummitSet(u: parameters): GrpBrdElt -> SetIndx

Presentation: MonStgElt	Default:

Given an element u of a braid group B, return the ultra summit set of u with respect to the specified presentation as indexed set of elements of B.

Example: Super Summit Sets of Braids (ex-ea29be)

(1) In the braid group B on 4 strings we compute the sets of positive conjugates and the super summit sets of $u = \sigma_1\sigma_2\sigma_1$ with respect to both Artin presentation and BKL presentation.

```
> B := BraidGroup(4);
> u := B.1*B.2*B.1;
> p_Artin := PositiveConjugates(u : Presentation := "Artin");
> p_BKL := PositiveConjugates(u : Presentation := "BKL");
> s_Artin := SuperSummitSet(u : Presentation := "Artin");
> s_BKL := SuperSummitSet(u : Presentation := "BKL");
```

Since the Artin generators form a subset of the BKL generators, every element which is positive with respect to the Artin presentation is also positive with respect to the BKL presentation. In particular, `p_Artin` is a subset of `p_BKL`.

```
> p_Artin subset p_BKL;
true
```

The converse inclusion does not hold.

```
> #p_Artin;
10
> #p_BKL;
36
```

For both presentations the super summit set is a subset of the set of positive conjugates, as u is positive. The converse inclusions do not hold.

```
> s_Artin subset p_Artin;
true
> s_BKL subset p_BKL;
true
> #s_Artin;
2
> #s_BKL;
12
```

(2) As we have seen in Section *Conjugacy Testing and Conjugacy Search*, we can decide whether two braids are conjugate by checking whether their super summit sets are equal. We illustrate this approach with two elements of B, using the Artin presentation of B.

```
> u := B.2 * B.1 * B.2^2 * B.1 * B.2;
> v := B.2^2 * B.1 * B.3 * B.1 * B.3;
```

Suppose we want to prove that u and v are not conjugate in B. We could start by checking the cycle structure of the induced permutations on the strings on which B acts.

```
> CycleStructure(InducedPermutation(u));
[ <1, 4> ]
> CycleStructure(InducedPermutation(v));
[ <1, 4> ]
```

This does not help. Next we can check the infima and the suprema of super summit representatives.

```
> SuperSummitInfimum(u) eq SuperSummitInfimum(v);
true
> SuperSummitSupremum(u) eq SuperSummitSupremum(v);
true
```

Again, we cannot conclude anything. We decide to compare the super summit sets of u and v.

```
> SuperSummitSet(u) eq SuperSummitSet(v);
false
```

Success! The super summit sets of u and v are different, proving that u and v are not conjugate. For a more efficient version of conjugacy testing see Example *Interactive Braid Conjugacy Testing*.

(3) Finally, we illustrate the significant difference in the sizes of super summit sets and ultra summit sets for slightly larger values of braid index and canonical length.

```
> B := BraidGroup(8);
```

We create a pseudo-random element of B as product of 5 simple elements independently chosen at random.

```
> x := B.4 * B.3 * B.2 * B.1 * B.5 * B.4 * B.5 *
> B.6 * B.7 * B.6 * B.5;
> x := x^2;
> Sx := SuperSummitSet(x);
> #Sx;
10972
> Ux := UltraSummitSet(x);
> #Ux;
36
```

The ultra summit set is much smaller than the super summit set. We try again.

```
> x := B.4 * B.3 * B.2 * B.1 * B.5 * B.4 * B.5;
> x := x^3;
> Sx := SuperSummitSet(x);
> #Sx;
882
> Ux := UltraSummitSet(x);
> #Ux;
18
```

The difference in sizes is still large. The behaviour exhibited by these examples is quite typical. In particular, the sizes of super summit sets for braids on a given number of strings and with a given canonical length show much larger fluctuations than the sizes of ultra summit sets. For a more detailed analysis we refer to [Geb03].

Computing Class Invariants Interactively

This section describes the functions relevant for interactive computation of the set of positive conjugates, the super summit set, and the ultra summit set for a given element of a braid group B as defined in Section *Conjugacy Testing and Conjugacy Search*.

Process versions of the algorithms used by the functions *PositiveConjugates*, *SuperSummitSet*, and *UltraSummitSet* are available for computing these invariants one element at a time.

PositiveConjugatesProcess(u: parameters): GrpBrdElt -> GrpBrdClassProc

```
Presentation: MonStgElt                    Default:
```

Given an element u of a braid group B, return a process for constructing the conjugates of u which can be represented as positive words in the specified presentation of B. The returned process contains the first positive conjugate of u if positive conjugates exist and is *empty* otherwise.

SuperSummitProcess(u: parameters): GrpBrdElt -> GrpBrdClassProc

```
Presentation: MonStgElt                    Default:
```

Given an element u of a braid group B, return a process for constructing the super summit elements of u with respect to the specified presentation of B. The returned process contains the first super summit element of u.

UltraSummitProcess(u: parameters): GrpBrdElt -> GrpBrdClassProc

```
Presentation: MonStgElt                    Default:
```

Given an element u of a braid group B, return a process for constructing the ultra summit elements of u with respect to the specified presentation of B. The returned process contains the first ultra summit element of u.

`BaseElement(P): GrpBrdClassProc -> GrpBrdElt`

Return the element used for the construction of the process P.

`# P: GrpBrdClassProc -> RngIntElt`

Return the number of elements that have been found by the process P.

`Representative(P): GrpBrdClassProc -> GrpBrdElt`

`Rep(P): GrpBrdClassProc -> GrpBrdElt`

Given a non-empty process P, return the element most recently found by P. If P is empty, a runtime error will occur. The function *IsEmpty* can be used for checking whether a process is empty, in order to avoid runtime errors in loops and user written functions.

`IsEmpty(P): GrpBrdClassProc -> BoolElt`

Return `true` if P is empty and `false` otherwise. This function can be used to check whether *Representative* can be called for a process P.

`Elements(P): GrpBrdClassProc -> SetIndx`

Return an indexed set containing the elements found so far by the process P.

`u in P: GrpBrdElt, GrpBrdClassProc -> BoolElt, GrpBrdElt`

Given an element u of a braid group and a process P for computing positive conjugates or super summit elements of the element b, return `true` and an element c satisfying $b^c = u$ if u is one of the elements that have been constructed by P and `false` otherwise.

`u notin P: GrpBrdElt, GrpBrdClassProc -> BoolElt`

Given an element u of a braid group and a process P, return `false` if u is one of the elements that have been constructed by P and `true` otherwise.

`NextElement(~P): GrpBrdClassProc`

Given a process P, continue searching for elements until the next element is found or the search completes without finding a new element. If a new element if found, it can subsequently be accessed using the function *Representative*. If the search completes without finding a new element, P is marked as *empty*. Calling `NextElement` on an empty process has no effect.

`Complete(~P): GrpBrdClassProc`

Given a process P, complete the search for elements. After executing this procedure, P is *empty* and the set of all elements found by P can be accessed using the function *Elements*. Calling `Complete` on an empty process has no effect.

Example: Interactive Braid Conjugacy Testing (ex-a3dd2a)

We sketch how the functions described in the preceding section could be used for testing whether two elements are conjugate and for computing a conjugating element if they are. The approach outlined here is basically the algorithm used by the function *IsConjugate*.

```
> function MyIsConjugate(u, v)
>
>  // check obvious invariants
>  infu := SuperSummitInfimum(u);
>  infv := SuperSummitInfimum(v);
>  supu := SuperSummitSupremum(u);
>  supv := SuperSummitSupremum(v);
>  if infu ne infv or supu ne supv then
>     return false, _;
>  end if;
>
>  // compute an ultra summit element for v
>  sv, cv := UltraSummitRepresentative(v);
>
>  // set up a process for computing the ultra summit set of u
```

(continues on next page)

(continued from previous page)

```
>  P := UltraSummitProcess(u);
>
>  // compute ultra summit elements of u until sv is found
>  //    or sv is seen not to be in the ultra summit set of u
>  while sv notin P and not IsEmpty(P) do
>     NextElement(~P);
>  end while;
>
>  print #P, "elements computed";
>  isconj, c := sv in P;
>  if isconj then
>     // return true and an element conjugating u to v
>     return true, c*cv^-1;
>  else
>     return false, _;
>  end if;
>
> end function;
```

We test our function using two pairs of elements of the braid group B on 4 strings.

```
> B := BraidGroup(4);
```

As we have seen in Example *Super Summit Sets of Braids*, the following elements u and v are not conjugate.

```
> u := B.2 * B.1 * B.2^2 * B.1 * B.2;
> v := B.2^2 * B.1 * B.3 * B.1 * B.3;
```

To prove this, our function has to compute the whole ultra summit set of u.

```
> MyIsConjugate(u,v);
2 elements computed
false
> #UltraSummitSet(u);
2
```

We try our function on another pair of elements.

```
> r := B.3*B.2*B.3*B.2^2*B.1*B.3*B.1*B.2;
> s := B.3^-1*B.2^-1*B.3*B.2*B.3*B.2^2*B.1*B.3*B.1*B.2^2*B.3;
> isconj, c := MyIsConjugate(r,s);
3 elements computed
> isconj;
true
> r^c eq s;
true
```

The ultra summit representative of s was the 3rd ultra summit element of r found. Note that the function did not have to compute the whole ultra summit set of r to find the answer.

```
> #UltraSummitSet(r);
6
```

In this small example, we could also have used super summit sets for conjugacy testing, as the super summit set of r is not much larger than its ultra summit set.

```
> #SuperSummitSet(r);
22
```

A more challenging application of the function `MyIsConjugate` from above will be presented in Example *Braid Group onto Symmetric Group*.

Computing Minimal Simple Elements

This section describes the functions for computing minimal simple elements as introduced in Section *Computing the Class Invariants* and functions for computing the *transport* and the *pullback* as defined in [Geb03].

All functions documented in this section accept two parameters, `Presentation` and `CheckArguments`. The parameter `Presentation` can be used to specify the presentation of a braid group B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"`. If no value is given for `Presentation`, the presentation selected for B is used. The parameter `CheckArguments` can be used to turn off argument checking for performance reasons. It should be noted that the results are undefined if functions are called with invalid arguments and argument checking is disabled.

`MinimalElementConjugatingToPositive(x, s: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt`

```
Presentation   : MonStgElt                    Default:
CheckArguments: BoolElt                       Default: true
```

Given a positive element x of a braid group B and a simple element s, return the minimal simple element $r_x(s)$ satisfying $s \preceq r_x(s)$ and $x^{r_x(s)} \in B^+$.

`MinimalElementConjugatingToSuperSummit(x, s: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt`

```
Presentation   : MonStgElt                    Default:
CheckArguments: BoolElt                       Default: true
```

Given an element x of a braid group B which is contained in its super summit set S_x and a simple element s, return the minimal simple element $\rho_x(s)$ satisfying $s \preceq \rho_x(s)$ and $x^{\rho_x(s)} \in S_x$.

`MinimalElementConjugatingToUltraSummit(x, s: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt`

```
Presentation   : MonStgElt                    Default:
CheckArguments: BoolElt                       Default: true
```

Given an element x of a braid group B which is contained in its ultra summit set U_x and a simple element s, return the minimal simple element $c_x(s)$ satisfying $s \preceq c_x(s)$ and $x^{c_x(s)} \in U_x$.

`Transport(x, s: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt`

```
Presentation   : MonStgElt                    Default:
CheckArguments: BoolElt                       Default: true
```

Given an element x of a braid group B and a simple element s such that both x and x^s are super summit elements, return the *transport* of s along $x \to \mathbf{c}(x)$, that is, the element $\phi_x(s) = (D \wedge_l xD^{-\inf(x)})^{-1} \cdot s \cdot (D \wedge_l x^s D^{-\inf(x)})$, where D is the fundamental element of B. The transport is a simple element satisfying $\mathbf{c}(x^s) = \mathbf{c}(x)^{\phi_x(s)}$ [Geb03].

`Pullback(x, s: parameters): GrpBrdElt, GrpBrdElt -> GrpBrdElt`

```
Presentation  : MonStgElt                    Default:
CheckArguments: BoolElt                      Default: true
```

Given an element x of a braid group B which is contained in its super summit set S_x and a simple element s, return the *pullback* of s along $x \to \mathbf{c}(x)$, that is, the unique $\preceq$-minimal element $\pi_x(s)$ satisfying $x^{\pi_x(s)} \in S_x$ and $s \preceq \phi_x(\pi_x(s))$ [Geb03].

Example: Ultra Summit Sets from Minimal Simple Elements (ex-3ccfd3)

The following function uses the technique sketched in Section *Conjugacy Testing and Conjugacy Search* for computing the ultra summit set of a given braid. This is basically the algorithm used by the Magma function `UltraSummitSet`.

```
> function MyUltraSummitSet(x)
>
> // create a subset of the ultra summit set of x
> U := {@ UltraSummitRepresentative(x) @};
> gens := Generators(Parent(x));
> pos := 1;
>
> // close U under conjugation with minimal simple elements
> while pos le #U do
>    y := U[pos];
>    // add missing conjugates of y
>    for z in { y^MinimalElementConjugatingToUltraSummit(y, s)
>                  : s in gens } do
>       if z notin U then
>          Include(~U, z);
>       end if;
>    end for;
>    pos +:= 1;
> end while;
>
> return U;
>
> end function;
```

82.5 Homomorphisms

For a general description of homomorphisms, we refer to Chapter Mappings. This section describes some special aspects of homomorphisms whose domain is in the category *GrpBrd*.

82.5.1 General Remarks

An important special case of homomorphisms with domain in the category *GrpBrd* is the following. A homomorphism $f : B \to G$, where B and B' are braid groups on n and m strings, respectively, and f is an embedding of B in G induced by

$$\sigma_i \to \sigma'^{\epsilon i}_{k} \quad (i = 1, \ldots, n-1)$$

where the σ_i are the Artin generators of B, the σ'_j are the Artin generators of B', $|\epsilon| = 1$, and k is a suitable constant.

The Magma implementation uses special optimisation techniques, if a homomorphism with domain in the category *GrpBrd* has the additional properties listed above. Compared to the general case, this results in faster evaluation of such homomorphisms, in particular in the case $\epsilon = 1$.

Computing preimages under homomorphisms with domain in the category *GrpBrd* currently is only supported for the special case described above. Moreover, computing the preimage of an element u under a map f may fail, even if u is contained in the image of f.

82.5.2 Constructing Homomorphisms

`hom< B -> G | S : parameters >: Struct, Struct -> Map`

Check: BoolElt	Default: true

Returns the homomorphism from the braid group B to the group G defined by the assignment S. S can be the one of the following:

(i) A list, sequence, or indexed set containing the images of all k Artin generators $B.1, \ldots, B.k$ of B. Here, the i-th element of S is interpreted as the image of $B.i$, that is, the order of the elements in S is important.

(ii) A list, sequence, enumerated set, or indexed set, containing k tuples $< x_i, y_i >$ or arrow pairs $x_i \text{ -> } y_i$, where x_i is a generator of B and $y_i \in G$ $(i = 1, \ldots, k)$ and the set $\{x_1, \ldots, x_k\}$ is the full set of Artin generators of B. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

Note, that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of B.

If the category of the codomain supports element arithmetic and element comparison, by default the constructed homomorphism is checked by verifying that the would-be images of the Artin generators satisfy the braid relations of B. In this case, it is assured that the returned map is a well-defined homomorphism. The most important situation in which it is not possible to perform checking is the case in which the domain is a finitely presented group (`FPGroup`; cf. Chapter *Introduction to Fp-Groups*) which is not free. Checking may be disabled by setting the parameter `Check` to `false`.

82.5.3 Accessing Homomorphisms

`e @ f: GrpBrdElt, Map -> GrpElt`

`f(e): Map, GrpBrdElt -> GrpElt`

Given a homomorphism whose domain is a braid group B and an element e of B, return the image of e under f as element of the codomain of f.

`B @ f: GrpBrd, Map -> Grp`

`f(B): Map, GrpBrd -> Grp`

Given a homomorphism whose domain is a braid group B, return the image of B under f as a subgroup of the codomain of f.

This function is not supported for all codomain categories.

`u @@ f: GrpElt, Map -> GrpBrdElt`

Given a homomorphism whose domain is a braid group B and an element u of the image of f, return the preimage of u under f as an element of B.

This function currently is only supported if f is an embedding of one braid group into another as described in Section *General Remarks*. Note, moreover, that computing the preimage of u may fail, even if u is contained in the image of f.

`Domain(f): Map -> Grp`

The domain of the homomorphism f.

`Codomain(f): Map -> Grp`

The codomain of the homomorphism f.

`Image(f): Map -> Grp`

The image or range of the homomorphism f as a subgroup of the codomain of f.

This function is not supported for all codomain categories.

Example: Braid Group onto Symmetric Group (ex-5eaf4d)

(1) The symmetric group on n letters is an epimorphic image of the braid group on n strings, where for $0 < i < n$ the image of the Artin generator σ_i is given by the transposition $(i, i+1)$.

We construct this homomorphism for the case $n = 10$.

```
> Bn := BraidGroup(10);
> Sn := Sym(10);
> f := hom< Bn->Sn | [ Sn!(i,i+1) : i in [1..Ngens(Bn)] ] >;
```

Of course, the image of `f` is the full symmetric group.

```
> Image(f) eq Sn;
true
```

Now we compute the image of a pseudo-random element of `Bn` under `f`.

```
> f(Random(Bn));
(1, 5, 8)(2, 4, 9, 7, 6, 3)
```

(2) (Key exchange as proposed in [KLC+00])

Consider a collection of $l + r$ strings $t_1, \ldots, t_{l+r}$ and the braid group B acting on $t_1, \ldots, t_{l+r}$ with Artin generators $\sigma_1, \ldots, \sigma_{l+r-1}$. The subgroups of B fixing the strings $t_{l+1}, \ldots, t_{l+r}$ and $t_1, \ldots, t_l$ may be identified with braid groups L on l strings and R on r strings, respectively, with the Artin generators of L and R corresponding to $\sigma_1, \ldots, \sigma_{l-1}$ and $\sigma_{l+1}, \ldots, \sigma_{l+r-1}$, respectively.

We set up these groups for $l = 6$ and $r = 7$ using the BKL presentations.

```
> l := 6;
> r := 7;
> B := BraidGroup(l+r : Presentation := "BKL");
> L := BraidGroup(l   : Presentation := "BKL");
> R := BraidGroup(r   : Presentation := "BKL");
```

We now construct the embeddings $f : L \to B$ and $g : R \to B$.

```
> f := hom< L-> B | [ L.i -> B.i : i in [1..Ngens(L)] ] >;
> g := hom< R-> B | [ R.i -> B.(l+i) : i in [1..Ngens(R)] ] >;
```

To complete the preparatory steps, we choose a random element of B which is not too short.

```
> x := Random(B, 15, 25);
```

The data constructed so far is assumed to be publicly available. Each time two users A and B require a shared key, the following steps are performed.

(a) A chooses a random secret element $a \in L$ and sends the normal form of $y_1 := x^a$ to B.

(b) B chooses a random secret element $b \in R$ and sends the normal form of $y_2 := x^b$ to A.

(c) A receives y_2 and computes the normal form of y_2^a.

(d) B receives y_1 and computes the normal form of y_1^b.

Note the following.

- Transmitting y_1 and y_2 in normal form disguises their structure as products $a^{-1}xa$ and $b^{-1}xb$, provided the words used are long enough and prevents simply reading off the conjugating elements a and b.

- Since the subgroups L and R of B commute, we have $ab = ba$, which implies $y_2^a = x^{ba} = x^{ab} = y_1^b$. Thus, the normal forms computed by A and B in steps (c) and (d), respectively, must be identical and can be used to extract a secret shared key.

We illustrate this, using the groups set up above. For step (a):

```
> a := Random(L, 15, 25);
> y1 := NormalForm(x^f(a));
```

Now for step (b):

```
> b := Random(R, 15, 25);
> y2 := NormalForm(x^g(b));
```

We now verify that A and B arrive at the same information in steps (c) and (d).

```
> K_A := NormalForm(y2^f(a));
> K_B := NormalForm(y1^g(b));
> AreIdentical(K_A, K_B);
true
```

We see that the information computed by A and B in steps (c) and (d) is indeed identical and hence can be used (in suitable form) as a common secret. Note, however, that the number of strings and the lengths of the elements used in the example above are much smaller than the values suggested for real cryptographic purposes.

(3) (Attack on key exchange)

We now show an attack on the key exchange outlined above using conjugacy search based on ultra summit sets.

An eavesdropper can try to compute an element c conjugating x to y_1. While this is not guaranteed to reproduce the braid a, the chances for a successful key recovery are quite good.

We decide to change to the Artin presentation of B and use the function `MyIsConjugate` from Example *Interactive Braid Conjugacy Testing* for computing a conjugating element c as above.

```
> SetPresentation(~B, "Artin");
> time _, c := MyIsConjugate(x, y1);
42 elements computed
Time: 0.020
```

Finding a conjugating element is no problem at all. Using the conjugating element, we can try to recover the shared secret. In this example we are lucky.

```
> NormalForm(y2^c) eq K_A;
true
```

In the conjugacy search above, a conjugating element was found after computing 42 ultra summit elements. The ultra summit set itself is larger, but can still be computed very easily.

```
> time Ux := UltraSummitSet(x);
Time: 3.150
> #Ux;
1584
```

The super summit set is, even in this small example, too large to be computed; conjugacy search based on super summit sets would quite likely fail.

```
> time Sx := SuperSummitSet(x);
Current total memory usage: 4055.1MB
System error: Out of memory.
```

Finally, we show that the attack using conjugacy search based on ultra summit sets is also applicable to larger examples. We try to recover a key, which is generated using elements with canonical lengths between 500 and 1000 in a braid group on 100 strings.

```
> l := 50;
> r := 50;
> B := BraidGroup(l+r);
> L := BraidGroup(l);
> R := BraidGroup(r);
>
> f := hom< L-> B | [ L.i -> B.i : i in [1..Ngens(L)] ] >;
> g := hom< R-> B | [ R.i -> B.(l+i) : i in [1..Ngens(R)] ] >;
>
> x := Random(B, 0, 1, 500, 1000);
>
> a := Random(L, 0, 1, 500, 1000);
> y1 := NormalForm(x^f(a));
>
> b := Random(R, 0, 1, 500, 1000);
> y2 := NormalForm(x^g(b));
>
> K_A := NormalForm(y2^f(a));
> K_B := NormalForm(y1^g(b));
> AreIdentical(K_A, K_B);
true
```

We again try to recover the key by computing an element conjugating x to y_1. This time, we use the built-in Magma function for efficiency reasons.

```
> time _, c := IsConjugate(x, y1);
Time: 18.350
> K_A eq NormalForm(y2^c);
false
```

Bad luck. – We managed to compute a conjugating element, but this failed to recover the key.

We try with an element conjugating x to y_2.

```
> time _, c := IsConjugate(x, y2);
Time: 3.800
> K_B eq NormalForm(y1^c);
true
```

Success! – Good that we didn't use this key to encrypt our credit card number!

82.5.4 Representations of Braid Groups

This section describes the functions available for creating a number of well known representations of braid groups.

SymmetricRepresentation(B): GrpBrd -> Map

Given a braid group B on n strings, return the natural epimorphism from B onto the symmetric group on n points, induced by the action of B on the strings by which B is defined.

BurauRepresentation(B): GrpBrd -> Map

Given a braid group B on n strings, return the Burau representation of B as homomorphism from B to the matrix algebra of degree n over the rational function field over the integers.

BurauRepresentation(B, p): GrpBrd, RngIntElt -> Map

Given a braid group B on n strings and a prime p, return the p-modular Burau representation of B as homomorphism from B to the matrix algebra of degree n over the rational function field over the field with p elements.

Example: Burau Representation of a Braid Group (ex-14a9f2)

We construct the Burau representation of the braid group on 4 strings.

```
> B := BraidGroup(4);
> f := BurauRepresentation(B);
```

Its codomain is a matrix algebra of degree 4 over the rational function field over the integers.

```
> A := Codomain(f);
> A;
GL(4, FunctionField(IntegerRing()))
> F := BaseRing(A);
> F;
Univariate rational function field over Integer Ring
Variables: $.1
```

To obtain nicer printing, we assign the name t to the generator of the function field F.

```
> AssignNames(~F, ["t"]);
```

Now we can check easily whether we remembered the definition of the Burau representation correctly.

```
> f(B.1);
[-t + 1      t      0      0]
[     1      0      0      0]
[     0      0      1      0]
[     0      0      0      1]

> f(B.2);
[     1      0      0      0]
[     0 -t + 1      t      0]
[     0      1      0      0]
[     0      0      0      1]

> f(B.3);
[     1      0      0      0]
[     0      1      0      0]
[     0      0 -t + 1      t]
[     0      0      1      0]
```

82.6 Bibliography

- [AAG99] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Math. Res. Lett.*, 6(3-4):287–291, 1999.
- [Art47] E. Artin. Theory of braids. *Ann. of Math. (2)*, 48:101–126, 1947.
- [BKL98] Joan Birman, Ki Hyoung Ko, and Sang Jin Lee. A new approach to the word and conjugacy problems in the braid groups. *Adv. Math.*, 139(2):322–353, 1998.

- [Deh02] Patrick Dehornoy. Groupes de Garside. *Ann. Sci. École Norm. Sup. (4)*, 35(2):267–306, 2002.
- [ERM94] Elsayed A. El-Rifai and H. R. Morton. Algorithms for positive braids. *Quart. J. Math. Oxford Ser. (2)*, 45(180):479–497, 1994.
- [ECH+92] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word processing in groups*. Jones and Bartlett Publishers, Boston, MA, 1992. ISBN 0-86720-244-0.
- [FGM03] Nuno Franco and Juan González-Meneses. Conjugacy problem for braid groups and Garside groups. *J. Algebra*, 266(1):112–132, 2003.
- [GKT+02] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, and U. Vishne. Length-based conjugacy search in the braid group. Preprint; available at http://www.arxiv.org/math.GR/0209267, 2002.
- [Gar69] F. A. Garside. The braid group and other groups. *Quart. J. Math. Oxford Ser. (2)*, 20:235–254, 1969.
- [Geb03] Volker Gebhardt. A new approach to the conjugacy problem in Garside groups. Preprint; available at http://www.arxiv.org/math.GT/0306199, 2003.
- [GM02] Juan González-Meneses. Improving an algorithm to solve Multiple Simultaneous Conjugacy Problems in braid groups. Preprint; available at http://www.arxiv.org/math.GT/0212150, 2002.
- [HS03] D. Hofheinz and R. Steinwandt. A practical attack on some braid group based cryptographic primitives. In *Public Key Cryptography, 6th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2003*, volume 2567 of LNCS, pages 187–198. Springer, 2003.
- [Hug02] J. Hughes. A linear algebraic attack on the AAFG1 braid group cryptosystem. In *The 7th Australasian Conference on Information Security and Privacy, ACISP 2002*, volume 2384 of LNCS, pages 176–189. Springer, 2002.
- [KLC+00] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park. New public-key cryptosystem using braid groups. In *Advances in cryptology—CRYPTO 2000*, pages 166–183. Springer, Berlin, 2000.
- [LP03] E. Lee and J. H. Park. Cryptanalysis of the public key encryption based on braid groups. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of LNCS, pages 477–490. Springer, 2003.
- [LL02] Sang Jin Lee and Eonkyung Lee. Potential Weakness of the Commutator Key Agreement Protocol Based on Braid Groups. In *Advances in Cryptology—EuroCrypt 2002*, volume 2332 of LNCS, pages 14–28. Springer, 2002.

Chapter 83

Groups Defined by Rewrite Systems

83.1 Introduction

The class of finitely presented groups defined by finite rewrite systems provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to the Knuth–Bendix completion procedure for groups defined by a finite (monoid) presentation. Much of the material in this chapter is taken from the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure is assumed. Some familiarity with KBMAG would be beneficial.

83.1.1 Terminology

A rewrite group G is a finitely presented group in which equality between elements of G, called *words* or *strings*, may be decidable via a sequence of rewriting equations, called *reduction relations, rules, or equations*. In the interests of efficiency the reduction rules are codified into a finite state automaton called a *reduction machine*. The words in a rewrite group G are ordered, as are the reduction relations of G. Several possible orderings of words are supported, namely short-lex, recursive, weighted short-lex, and wreath-product orderings. A rewrite group can be *confluent* or *non-confluent*. If a rewrite group G is confluent its reduction relations, or more specifically its reduction machine, can be used to reduce words in G to their irreducible normal forms under the given ordering, and so the word problem for G can be efficiently solved.

83.1.2 The Category of Rewrite Groups

The family of all rewrite groups forms a category. The objects are the rewrite groups and the morphisms are group homomorphisms. The Magma designation for this category of groups is `GrpRWS`. Elements of a rewrite group are designated as `GrpRWSElt`.

83.1.3 The Construction of a Rewrite Group

A rewrite group G is constructed in a three-step process:

(i) We construct a free group FG.

(ii) We construct a quotient F of FG.

(iii) We create a monoid presentation of F and then run a Knuth–Bendix completion procedure on this presentation to create a rewrite group G.

The Knuth–Bendix procedure may or may not succeed. If it fails the user may need to perform the above steps several times, manually adjusting the parameters to the Knuth–Bendix procedure. If it succeeds then the rewrite system constructed will be confluent.

83.2 Constructing Confluent Presentations

83.2.1 The Knuth-Bendix Procedure

`RWSGroup(F: parameters): MonFP -> MonRWS`

Suppose F is a finitely presented group. Internally, the first step is to construct a presentation for a monoid M. By default, the generators of M are taken to be $g_1, g_1^{-1}, \ldots, g_n, g_n^{-1}$, where $g_1, \ldots, g_n$ are the generators of F. The relations for M are taken to be the relations of F together with the trivial relations $g_1 g_1^{-1} = g_1^{-1} g_1 = 1$. The Knuth–Bendix completion procedure for monoids is now applied to M. Regardless of whether or not the completion procedure succeeds, the result will be a rewrite monoid, M, containing a reduction machine and a sequence of reduction relations. If the procedure succeeds M will be marked as confluent, and the word problem for M is therefore decidable. If, as is very likely, the procedure fails then M will be marked as non-confluent. In this case M will contain both the reduction relations and the reduction machine computed up to the point of failure. Reductions made using these relations will be correct in F, but words that are equal in F are not guaranteed to reduce to the same word.

The Knuth–Bendix procedure requires ordering to be defined on both the generators and the words. The default generator ordering is that induced by the ordering of the generators of F while the default ordering on strings is the `ShortLex` order. We give a simple example and then discuss the parameters that allow the user to specify these two orderings.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent rewrite monoid. The optimal values for these limits vary from example to example. The various parameters that allow the user to specify the limits for these variables will be described in a subsequent section.

Example: Rewrite Group for the Von Dyck (2,3,5) Group (ex-8622c9)

We construct the Von Dyck $(2, 3, 5)$ group. Since a string ordering is not specified the default `ShortLex` ordering is used. Similarly, since a generator ordering is not specified, the default generator ordering, in this case $[a, a^{-1}, b, b^{-1}]$, is used.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | a*a=1, b*b=b^-1, a*b^-1*a*b^-1*a=b*a*b*a*b>;
> G := RWSGroup(F);
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = ShortLex.
The reduction machine has 39 states.
The rewrite relations are:
    a^2 = Id(F)
    b * b^-1 = Id(F)
    b^-1 * b = Id(F)
    b^2 = b^-1
    b * a * b * a * b = a * b^-1 * a * b^-1 * a
    b^-2 = b
    b^-1 * a * b^-1 * a * b^-1 = a * b * a * b * a
    a^-1 = a
    a * b^-1 * a * b^-1 * a * b = b * a * b * a * b^-1
    b * a * b^-1 * a * b^-1 * a = b^-1 * a * b * a * b
    a * b * a * b * a * b^-1 = b^-1 * a * b^-1 * a * b
    b^-1 * a * b * a * b * a = b * a * b^-1 * a * b^-1
    b * a * b * a * b^-1 * a * b * a = a * b^-1 * a * b * a * b^-1 * a * b^-1
    b^-1 * a * b^-1 * a * b * a * b^-1 * a = a * b * a * b^-1 * a * b * a * b
    b^-1 * a * b * a * b^-1 * a * b * a * b^-1 = b * a * b^-1 * a * b * a * b^-1
```

(continues on next page)

(continued from previous page)

```
        * a * b
    b * a * b^-1 * a * b * a * b^-1 * a * b^-1 = (b^-1 * a * b * a)^2
    b^-1 * a * b * a * b^-1 * a * b * a * b = (b * a * b^-1 * a)^2
    b * a * b^-1 * a * b * a * b^-1 * a * b * a = a * b * a * b^-1 * a * b * a *
        b^-1 * a * b
```

83.2.2 Defining Orderings

`RWSGroup(F: parameters): MonFP -> MonRWS`

```
GeneratorOrder: SeqEnum                    Default:
Ordering      : MonStgElt                  Default: "ShortLex"
Levels        : SeqEnum                    Default:
Weights       : SeqEnum                    Default:
```

Attempt to construct a confluent presentation for the finitely presented group F using the Knuth-Bendix completion algorithm. In this section we describe how the user can specify the generator order and the ordering on strings.

Give an ordering for the generators. This ordering affects the ordering of words in the alphabet. If not specified the ordering defaults to the order induced by F's generators, that is $[g_1, \ldots, g_n]$ where $g_1, \ldots, g_n$ are the generators of F.

`Ordering := "ShortLex"`: Use the short-lex ordering on strings. Shorter words come before longer, and for words of equal length lexicographical ordering is used, using the given ordering of the generators.

`Ordering := "Recursive" | "RtRecursive"`: Use a recursive ordering on strings. There are various ways to define this. Perhaps the quickest is as follows. Let u and v be strings in the generators. If one of u and v, say v, is empty, then $u \geq v$. Otherwise, let $u = u'a$ and $v = v'b$, where a and b are generators. Then $u > v$ if and only if one of the following holds:

(i) $a = b$ and $u' > v'$;

(ii) $a > b$ and $u > v'$;

(iii) $b > a$ and $u' > v$.

The `RtRecursive` ordering is similar to the `Recursive` ordering, but with $u = au'$ and $v = bv'$. Occasionally one or the other runs significantly quicker, but usually they perform similarly.

`Ordering := "WtLex"`: Use a weighted-lex ordering. `Weights` should be a sequence of non-negative integers, with the i-th element of `Weights` giving the weight of the i-th generator. The length of `Weights` must equal the number of generators. The length of words in the generators is then computed by adding up the weights of the generators in the words. Otherwise, ordering is as for short-lex.

`Ordering := "Wreath"`: Use a wreath-product ordering. `Levels` should be a sequence of non-negative integers, with the i-th element of `Levels` giving the level of the i-th generator. The length of `Levels` must equal the number of generators. In this ordering, two strings involving generators of the same level are ordered using short-lex, but all strings in generators of a higher level are larger than those involving generators of a lower level. That is not a complete definition; one can be found in [Sim94, pp. 46--50]. Note that the recursive ordering is the special case in which the level of generator number i is i.

Example: Recursive Ordering for a Non-Hopfian Group `(ex-e4565e)`

A confluent presentation is constructed for an infinite non-hopfian group using the `Recursive` ordering.

```
> F<a, b> := Group< a, b | b^-1*a^2*b=a^3>;
> G := RWSGroup(F : Ordering :="Recursive");
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = Recursive.
The reduction machine has 7 states.
The rewrite relations are:
    a * a^-1 = Id(FG)
    a^-1 * a = Id(FG)
    b * b^-1 = Id(FG)
    b^-1 * b = Id(FG)
    a^3 * b^-1 = b^-1 * a^2
    a^2 * b = b * a^3
    a^-1 * b^-1 = a^2 * b^-1 * a^-2
    a^-1 * b = a * b * a^-3
```

Example: Rewrite System for a Free Nilpotent Group (ex-8d0d83)

A confluent presentation of a free nilpotent group of rank 2 and class 2 is constructed by the following code. Note that the lower weight generators (in the sense of nilpotency class) need to come first in the ordering of generators.

```
> FG<a,b,c> := FreeGroup(3);
> F := quo< FG | b*a=a*b*c, c*a=a*c, c*b=b*c>;
> G := RWSGroup(F : Ordering :="Recursive",
>                 GeneratorOrder := [c,c^-1,b,b^-1,a,a^-1]);
> G;
A confluent rewrite group.
Generator Ordering = [ c, c^-1, b, b^-1, a, a^-1 ]
Ordering = Recursive.
The reduction machine has 7 states.
The rewrite relations are:
    c * c^-1 = Id(FG)
    c^-1 * c = Id(FG)
    b * b^-1 = Id(FG)
    b^-1 * b = Id(FG)
    a * a^-1 = Id(FG)
    a^-1 * a = Id(FG)
    b * a = a * b * c
    c * a = a * c
    c * b = b * c
    c^-1 * a = a * c^-1
    c * a^-1 = a^-1 * c
    c^-1 * b = b * c^-1
    c * b^-1 = b^-1 * c
    b^-1 * a = a * b^-1 * c^-1
    b^-1 * a^-1 = a^-1 * b^-1 * c
    c^-1 * a^-1 = a^-1 * c^-1
    c^-1 * b^-1 = b^-1 * c^-1
    b * a^-1 = a^-1 * b * c^-1
```

83.2.3 Setting Limits

In this section we introduce the various parameters used to control the execution of the Knuth-Bendix procedure.

`RWSMonoid(F: parameters): MonFP -> MonRWS`

```
MaxRelations : RngIntElt                    Default: 32767
TidyInt      : RngIntElt                    Default: 100
RabinKarp    : Tup                          Default:
MaxStates    : RngIntElt                    Default:
MaxReduceLen : RngIntElt                    Default: 32767
ConfNum      : RngIntElt                    Default: 500
MaxStoredLen : Tup                          Default:
MaxOverlapLen: RngIntElt                    Default:
Sort         : BoolElt                      Default: false
MaxOpLen     : RngIntElt                    Default: 0
```

Attempt to construct a confluent presentation for the finitely presented group F using the Knuth-Bendix completion algorithm. We present details of the various parameters used to control the execution of the Knuth-Bendix.

Limit the maximum number of reduction equations to `MaxRelations`.

After finding `TidyInt` new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.)

Use the Rabin-Karp algorithm for word-reduction on words having length at least l, provided that there are at least n equations, where `RabinKarp := <l, n>`. This uses less space than the default reduction automaton, but it is distinctly slower, so it should only be used when seriously short of memory. Indeed this option is only really useful for examples in which collapse occurs - i.e. at some intermediate stage of the calculation there is a very large set of equations, which later reduces to a much smaller confluent set. Collapse is not uncommon when analysing pathological presentations of finite groups, and this is one situation where the performance of the Knuth–Bendix algorithm can be superior to that of Todd-Coxeter coset enumeration. The best setting for `RabinKarp` varies from example to example - generally speaking, the smaller l is, the slower things will be, so set it as high as possible subject to not running out of memory. The number of equations n should be set to a value greater than the expected final number of equations.

Limit the maximum number of states of the finite state automaton used for word reduction to `MaxStates`. By default there is no limit, and the space allocated is increased dynamically as required. The space needed for the reduction automaton can also be restricted by using the `RabinKarp` parameter. This limit is not usually needed.

Limit the maximum allowed length that a word can reach during reduction to `MaxReduceLen`. It is only likely to be exceeded when using the recursive ordering on words. This limit is usually not needed.

If `ConfNum` overlaps are processed and no new equations are discovered, then the overlap searching process is interrupted, and a fast check for confluence performed on the existing set of equations. Doing this too often wastes time, but doing it at the right moment can also save a lot of time. If `ConfNum` $= 0$, then the fast confluence check is performed only when the search for overlaps is complete.

Warning: Changing the default setting on any of the following parameters may either cause the procedure to terminate without having found a confluent presentation or may change the underlying group.

Only equations in which the left and right hand sides have lengths at most l and r, respectively, where `MaxStoredLen := <l, r>` are kept. Of course this may cause the overlap search to complete on a set of equations that is not confluent. In some examples, particularly those involving collapse (i.e. a large intermediate set of equations, which later simplifies to a small set), it can result in a confluent set being found much more quickly. It is most often useful when using a recursive ordering on words. Another danger with this option is that sometimes discarding equations can result in information being lost, and the monoid defined by the equations changes.

Only overlaps of total length at most `MaxOverlapLen` are processed. Of course this may cause the overlap search to complete on a set of equations that is not confluent.

If `Sort` is set to `true` then the equations will be sorted in order of increasing length of their left hand sides, rather than the default, which is to leave them in the order in which they were found. `MaxOpLen` should be a non-negative integer. If `MaxOpLen` is positive, then only equations with left hand sides having length at most `MaxOpLen` are output. If `MaxOpLen` is zero, then all equations are sorted by length. Of course, if `MaxOpLen` is positive, there is a danger that the monoid defined by the output equations may be different from the original.

`SetVerbose("KBMAG", v): MonStgElt, RngIntElt`

Set the verbose printing level for the Knuth-Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for v are 0 to 3 inclusive. Setting v to 0 corresponds to the '-silent' option of KBMAG in which no extra output is printed. Setting v to 2 corresponds to the '-v' (verbose) option of KBMAG in which a small amount of extra output is printed. Setting v to 3 corresponds to the '-vv' (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

83.2.4 Accessing Group Information

The functions in this group provide access to basic information stored for a rewrite group G.

`G . i: GrpRWS, RngIntElt -> GrpRWSElt`

The i-th defining generator for G. The integer i must lie in the range $[-r, r]$, where r is the number of group G.

`Generators(G): GrpRWS -> [GrpRWSElt]`

A sequence containing the defining generators for G.

`NumberOfGenerators(G): GrpRWS -> RngIntElt`

`Ngens(G): GrpRWS -> RngIntElt`

The number of defining generators for G.

`Relations(G): GrpRWS -> [GrpFPRel]`

A sequence containing the defining relations for G. The relations will be given between elements of the free group of which G is a quotient. In these relations the (image of the) left hand side (in G) will always be greater than the (image of the) right hand side (in G) in the ordering on words used to construct G.

`NumberOfRelations(G): GrpRWS -> RngIntElt`

`Nrels(G): GrpRWS -> RngIntElt`

The number of relations in G.

`Ordering(G): GrpRWS -> String`

The ordering of G.

Example: Accessing Rewrite Group Information (ex-13860e)

We illustrate the access operations using the following presentation of $\mathbb{Z} \wr C_2$.

```
> FG<a,b,t> := FreeGroup(3);
> F := quo< FG | t^2=1, b*a=a*b, t*a*t=b>;
> G<x,y,z> := RWSGroup(F);
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1, t, t^-1 ]
Ordering = ShortLex.
The reduction machine has 6 states.
The rewrite relations are:
```

(continues on next page)

(continued from previous page)

```
    a * a^-1 = Id(F)
    a^-1 * a = Id(F)
    b * b^-1 = Id(F)
    b^-1 * b = Id(F)
    t^2 = Id(F)
    b * a = a * b
    t * a = b * t
    b^-1 * a = a * b^-1
    t * b = a * t
    b * a^-1 = a^-1 * b
    t * a^-1 = b^-1 * t
    t^-1 = t
    b^-1 * a^-1 = a^-1 * b^-1
    t * b^-1 = a^-1 * t
> G.1;
x
> G.1*G.2;
x * y
> Generators(G);
[ x, y, z ]
> Ngens(G);
3
> Relations(G);
[ a * a^-1 = Id(F), a^-1 * a = Id(F), b * b^-1 = Id(F), b^-1 * b = Id(F), t^2 =
Id(F), b * a = a * b, t * a = b * t, b^-1 * a = a * b^-1, t * b = a * t, b *
a^-1 = a^-1 * b, t * a^-1 = b^-1 * t, t^-1 = t, b^-1 * a^-1 = a^-1 * b^-1, t *
b^-1 = a^-1 * t ]
> Nrels(G);
14
> Ordering(G);
ShortLex
```

83.3 Properties of a Rewrite Group

IsConfluent(G): GrpRWS -> BoolElt

Returns **true** if G is confluent, **false** otherwise.

IsFinite(G): GrpRWS -> BoolElt, RngIntElt

Given a confluent group G return **true** if G has finite order and **false** otherwise. If G does have finite order also return the order of G.

Order(G): GrpRWS -> RngIntElt

 # G: GrpRWS -> RngIntElt

The order of the group G as an integer. If the order of G is known to be infinite, the symbol ∞ is returned.

Example: Confluence of the Weyl Group E8 (ex-40c60a)

We construct the Weyl group E_8.

```
> FG<a,b,c,d,e,f,g,h> := FreeGroup(8);
> Q := quo< FG | a^2=1, b^2=1, c^2=1, d^2=1, e^2=1, f^2=1, g^2=1,
>     h^2=1, b*a*b=a*b*a, c*a=a*c, d*a=a*d, e*a=a*e, f*a=a*f,
>     g*a=a*g, h*a=a*h, c*b*c=b*c*b, d*b=b*d, e*b=b*e, f*b=b*f,
>     g*b=b*g, h*b=b*h, d*c*d=c*d*c, e*c*e=c*e*c, f*c=c*f,
>     g*c=c*g, h*c=c*h, e*d=d*e, f*d=d*f, g*d=d*g, h*d=d*h,
```

(continues on next page)

(continued from previous page)

```
>     f*e*f=e*f*e, g*e=e*g, h*e=e*h, g*f*g=f*g*f, h*f=f*h,
>     h*g*h=g*h*g>;
> G := RWSGroup(Q);
> IsConfluent(G);
true
> IsFinite(G);
true 696729600
```

So the group is finite of order $696,729,600$.

Example: Order of an Infinite Rewrite Group (ex-174016)

We construct a 2-generator 2-relator group and use the order function to show that the group is infinite. The symbol `Infinity`, returned by `Order`, indicates that the group has infinite order.

```
> G := Group< x, y | x^2 = y^2, x*y*x = y*x*y >;
> R := RWSGroup(G);
> print Order(G);
Infinity
```

83.4 Arithmetic with Words

83.4.1 Construction of a Word

`Identity(G): GrpRWS -> GrpRWSElt`

`Id(G): GrpRWS -> GrpRWSElt`

`G ! 1: GrpRWS, RngIntElt -> GrpRWSElt`

Construct the identity word in G.

`G ! [ i1, ..., is ]: GrpRWS, [ RngIntElt ] -> GrpRWSElt`

Given a rewrite group G defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[-r, r]$, excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \cdots * G.|i_s|^{\epsilon_s}$$

where ϵ_j is $+1$ if i_j is positive, and -1 if i_j is negative. The resulting word is reduced using the reduction machine associated with G.

`Parent(w): GrpRWSElt -> GrpRWS`

The parent group G for the word w.

Example: Constructing Words in a Rewrite Group (ex-6e6f68)

We construct the Fibonacci group $F(2,7)$, and its identity.

```
> FG<a,b,c,d,e,f,g> := FreeGroup(7);
> F := quo< FG | a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b>;
> G := RWSGroup(F : TidyInt := 1000);
> Id(G);
Id(G)
> G!1;
Id(G)
> G![1,2];
G.3
```

83.4.2 Element Operations

Having constructed a rewrite group G one can perform arithmetic with words in G. Assuming we have $u, v \in G$ then the product $u * v$ will be computed as follows:

(i) the product $w = u * v$ is formed as a product in the appropriate free group.

(ii) w is reduced using the reduction machine associated with G.

If G is confluent, then w will be the unique minimal word that represents $u * v$ under the ordering of G. If G is not confluent, then there are some pairs of words which are equal in G, but which reduce to distinct words, and hence w will not be a unique normal form. Note that:

(i) reduction of w can cause an increase in the length of w. At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.

(ii) the implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v: GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return their product.

`u / v: GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return the product of the word u by the inverse of the word v, i.e. the word $u * v^{-1}$.

`u ^ n: GrpRWSElt, RngIntElt -> GrpRWSElt`

The n-th power of the word w.

`u ^ v: GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return the conjugate of the word u by the word v, i.e. the word $v^{-1} * u * v$.

`Inverse(w): GrpRWSElt -> GrpRWSElt`

The inverse of the word w.

`(u, v): GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return the commutator of the words u and v, i.e., the word $u^{-1}v^{-1}uv$.

`(u`$_1$`, ..., u`$_r$`): GrpRWSElt, ..., GrpRWSElt -> GrpRWSElt`

Given r words $u_1, \ldots, u_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`u eq v: GrpRWSElt, GrpRWSElt -> BoolElt`

Given words w and v belonging to the same group, return `true` if w and v reduce to the same normal form, `false` otherwise. If G is confluent this tests for equality. If G is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v: GrpRWSElt, GrpRWSElt -> BoolElt`

Given words w and v belonging to the same group, return `false` if w and v reduce to the same normal form, `true` otherwise. If G is confluent this tests for non-equality. If G is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w): GrpRWSElt -> BoolElt`

`IsIdentity(w): GrpRWSElt -> BoolElt`

Returns `true` if the word w is the identity word.

`# u: GrpRWSElt -> RngIntElt`

The length of the word w.

`ElementToSequence(u): GrpRWSElt -> [ RngIntElt ]`

`Eltseq(u): GrpRWSElt -> [ RngIntElt ]`

The sequence Q obtained by decomposing the element u of a rewrite group into its constituent generators and generator inverses. Suppose u is a word in the rewrite group G. Then, if $u = G.i_1^{e_1} \cdots G.i_m^{e_m}$, with each $e_i = \pm 1$, then $Q[j] = i_j$ if $e_j = +1$ and $Q[j] = -i_j$ if $e_j = -1$, for $j = 1, \ldots, m$.

Example: Word Arithmetic in a Fibonacci Group (ex-0eba9a)

We illustrate the word operations by applying them to elements of the Fibonacci group $F(2, 5)$.

```
> FG<a,b,c,d,e> := FreeGroup(5);
> F := quo< FG | a*b=c, b*c=d, c*d=e, d*e=a, e*a=b>;
> G<a,b,c,d,e> := RWSGroup(F);
> a*b^-1;
e^-1
> a/b;
e^-1
> (c*d)^4;
a
> a^b, b^-1*a*b;
a a
> a^-2,
> Inverse(a)^2;
d d
> c^-1*d^-1*c*d eq (c,d);
true
> IsIdentity(a*b*c^-1);
true
> #(c*d);
1
```

83.5 Operations on the Set of Group Elements

`Random(G, n): GrpRWS, RngIntElt -> GrpRWSElt`

A random word of length at most n in the generators of G.

`Random(G): GrpRWS -> GrpRWSElt`

A random word (of length at most the order of G) in the generators of G.

`Representative(G): GrpRWS -> GrpRWSElt`

`Rep(G): GrpRWS -> GrpRWSElt`

An element chosen from G.

`Set(G, a, b): GrpRWS, RngIntElt, RngIntElt -> SetEnum`

```
Search: MonStgElt                    Default: "DFS"
```

Create the set of reduced words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Set(G): GrpRWS -> SetEnum`

```
Search: MonStgElt                    Default: "DFS"
```

Create the set of reduced words that is the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Seq(G, a, b): GrpRWS, RngIntElt, RngIntElt -> SeqEnum`

```
Search: MonStgElt                    Default: "DFS"
```

Create the sequence S of reduced words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

`Seq(G): GrpRWS -> SeqEnum`

```
Search: MonStgElt                    Default: "DFS"
```

Create a sequence S of reduced words in the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Example: Element Sets of a Rewrite Group (ex-6fc61b)

We construct the group D_{22}, together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FG<a,b,c,d,e,f> := FreeGroup(6);
> Q := quo< FG | a*c^-1*a^-1*d=1, b*f*b^-1*e^-1=1, c*e*c^-1*d^-1=1,
>                d*f^-1*d^-1*a=1, e*b*e^-1*a^-1=1, f*c^-1*f^-1*b^-1=1>;
> G<a,b,c,d,e,f> := RWSGroup(Q);
> Representative(G);
Id(G)
> Random(G);
b
> Random(G, 5);
a * d * b
> Set(G);
{ a * d * b, a * b, a * b * e, a * c, a * d, d * b, b * e, a * b * a,
  a * b * d, b * a, a * c * e, Id(G), b * d, c * e, e, f, a, a * e, b,
  c, a * f, d }
> Seq(G : Search := "DFS");
[ Id(G), a, a * b, a * b * a, a * b * d, a * b * e, a * c, a * c * e,
  a * d, a * d * b, a * e, a * f, b, b * a, b * d, b * e, c, c * e, d,
  d * b, e, f ]
```

83.6 Homomorphisms

For a general description of homomorphisms, we refer to chapter Mappings. This section describes some special aspects of homomorphisms whose domain or codomain is a rewrite group.

83.6.1 General Remarks

Groups in the category GrpRWS currently are accepted as codomains only in some special situations. The most important cases in which a rewrite group can be used as a codomain are group homomorphisms whose domain is in one of the categories *GrpFP*, *GrpGPC*, GrpRWS, or *GrpAtc*.

83.6.2 Construction of Homomorphisms

`hom< R -> G | S >: Struct, Struct -> Map`

Returns the homomorphism from the rewrite group R to the group G defined by the expression S which can be the one of the following:

(i) A list, sequence, or indexed set containing the images of the n generators $R.1, \ldots, R.n$ of R. Here, the i-th element of S is interpreted as the image of $R.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set, or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i \to y_i$, where x_i is a generator of R and $y_i \in G$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of R. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor.

Note that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of R.

83.7 Conversion to a Finitely Presented Group

There is a standard way to convert a rewrite group into a finitely presented group using the functions *Relations* and *Simplify*. This is shown in the following example.

Example: Converting a Rewrite Group to an FP-Group (ex-c2fddd)

We construct a two generator free abelian group as a rewrite group and then convert it into a finitely presented group.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | b^-1*a*b=a >;
> G := RWSGroup(F);
> print G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = ShortLex.
The reduction machine has 5 states.
The rewrite relations are:
    a * a^-1 = Id(FG)
    a^-1 * a = Id(FG)
    b * b^-1 = Id(FG)
    b^-1 * b = Id(FG)
    b^-1 * a = a * b^-1
    b * a = a * b
    b^-1 * a^-1 = a^-1 * b^-1
    b * a^-1 = a^-1 * b
> P<x,y> := Simplify(quo< FG | Relations(G)>);
> print P;
Finitely presented group P on 2 generators
Generators as words
    x = $.1
```

(continues on next page)

(continued from previous page)

```
    y = $.2
Relations
    (y, x^-1) = Id(P)
```

83.8 Bibliography

- [Hol97] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.
- [Sim94] Charles C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

Chapter 84

Automatic and Hyperbolic Groups

84.1 Introduction

Automatic groups are a family of finitely generated groups whose Cayley graph satisfies a certain condition (the fellow traveller property). This condition is realised for a group G by associating with it a number of finite state automata. This makes various group operations, notably equality between words of G and word enumeration, decidable through the use of these automata.

Word hyperbolic groups are an important subclass of automatic groups. In fact the concept of an automatic group emerged from work of J. W. Cannon on the Cayley graphs of word hyperbolic groups.

In the first part of the chapter, a Magma level interface to Derek Holt's KBMAG programs, and specifically to KBMAG's automatic groups program *autgroup* is described. The main procedure applies the Knuth-Bendix completion procedure in an attempt to construct a confluent presentation for the group.

In the second part of the chapter an intrinsic that attempts to determine if a finitely presented group is word hyperbolic is described. The procedure was developed by Steve Linton, Max Neunhoffer, Richard Parker and Colva Roney-Dougal and implemented in Magma by Derek Holt. A separate procedure that tests whether an automatic group is word hyperbolic is also described.

84.2 Automatic Groups

The central intrinsic attempts to construct an automatic structure in the form of finite state automata for a given finitely presented group. Subsequent sections describe operations with elements including arithmetic, element enumeration, and the construction of a growth function.

The words in an automatic group G are ordered using the short-lex ordering on words. Under this ordering shorter words come before longer, and for words of equal length lexicographical ordering is used, based on the given ordering of the generators. Thus, a *short-lex automatic group* is constructed.

The family of all automatic groups forms a category. The objects are the automatic groups and the morphisms are group homomorphisms. The Magma designation for this category of groups is `GrpAtc`. Elements of a automatic group are designated as `GrpAtcElt`.

Much of the material in this section is based on the KBMAG documentation [Hol97]. Some familiarity with the Knuth–Bendix completion procedure and the automata associated with a short-lex automatic group is assumed.

84.2.1 Construction of an Automatic Group

An automatic group G is constructed in a three-step process:

(i) We construct a free group FG.

(ii) We construct a quotient F of FG.

(iii) We create a monoid presentation for F and then run procedures which attempt to construct the automata associated with G and to prove them correct.

These procedures may or may not succeed. Of course, if G is not an automatic group then they have no chance of succeeding.

AutomaticGroup(F: parameters): GrpFP -> GrpAtc

IsAutomaticGroup(F: parameters): GrpFP -> BoolElt, GrpAtc

Internally a monoid presentation P of the group F is constructed. By default the generators of P are taken to be $g_1, {g_1}^{-1}, \ldots, g_n, {g_n}^{-1}$ where $g_1, \ldots, g_n$ are the generators of F. The relations of P are taken to be the relations of F. The trivial relations between the generators and their inverses are also added. The word ordering is the short-lex ordering. The Knuth–Bendix completion procedure for monoids is now run on P to calculate the word difference automata corresponding to the generated equations, which are then used to calculate the finite state automata associated with a short-lex automatic group. In successful cases these automata are proved correct in the final step.

If the procedure succeeds the result will be an automatic group, G, containing four automata. These are the first and second word-difference machines, the word acceptor, and the word multiplier. The form `AutomaticGroup` returns an automatic group while the form `IsAutomaticGroup` returns the boolean value *true* and the automatic group. If the procedure fails, the first form does not return a value while the second returns the boolean value *false*.

For simple examples, the algorithms work quickly, and do not require much space. For more difficult examples, the algorithms are often capable of completing successfully, but they can sometimes be expensive in terms of time and space requirements. Another point to be borne in mind is that the algorithms sometimes produce temporary disk files which the user does not normally see (because they are automatically removed after use), but can occasionally be very large. These files are stored in the */tmp* directory. If you interrupt a running automatic group calculation you must remove these temporary files yourself.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent automatic group. The optimal values for these limits varies from example to example. Some of these limits may be specified by setting parameters (see the next section). In particular, if a first attempt to compute the automatic structure of a group fails, it should be run again with the parameter `Large` (or `Huge`) set to `true`.

Example: Automatic Structure of a Surface Group (ex-6ecc26)

We construct the automatic structure for the fundamental group of the torus. Since a generator ordering is not specified, the default generator ordering, $[a, a^{-1}, b, b^{-1}, c, c^{-1}, d, d^{-1}]$, is used.

```
> FG<a,b,c,d> := FreeGroup(4);
> F := quo< FG | a^-1*b^-1*a*b=d^-1*c^-1*d*c>;
> f, G := IsAutomaticGroup(F);
> f;

true

> G;

An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1 ]
The second word difference machine has 33 states.
The word acceptor has 36 states.
```

We now do the same computation with a verbose flag (discussed later) turned on to display some information.

```
> FG<a,b,c,d> := FreeGroup(4);
> F := quo< FG | a^-1*b^-1*a*b=d^-1*c^-1*d*c>;
> SetVerbose("KBMAG", 1);
> f, G := IsAutomaticGroup(F);

Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Halting with 118 equations.
#First word-difference machine with 33 states computed.
#Second word-difference machine with 33 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 36 states computed.
#General multiplier with 104 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 220 states computed.
#Checking inverse and short relations.
#Checking relation:  _8*_6*_7*_5 = _2*_4*_1*_3
#Axiom checking succeeded.

> G;

An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1 ]
The second word difference machine has 33 states.
The word acceptor has 36 states.
```

84.2.2 Modifying Limits

In this section we describe the various parameters used to control the execution of the procedures employed to determine the automatic structure.

AutomaticGroup(F: parameters): GrpFP -> GrpAtc

IsAutomaticGroup(F: parameters): GrpFP -> BoolElt, GrpAtc

```
Large         : BoolElt                    Default: false
Huge          : BoolElt                    Default: false
MaxRelations  : RngIntElt                  Default: 200
TidyInt       : RngIntElt                  Default: 20
GeneratorOrder: SeqEnum                    Default:
MaxWordDiffs  : RngIntElt                  Default:
HaltingFactor : RngIntElt                  Default: 100
MinTime       : RngIntElt                  Default: 5
```

Attempt to construct an automatic structure for the finitely presented group F (see the main entry). We now present details of the various parameters used to control the execution of the procedures.

If Large is set to true large hash tables are used internally. Also the Knuth–Bendix algorithm is run with larger parameters, specifically TidyInt is set to 500, MaxRelations is set to 262144, MaxStates is set to unlimited, HaltingFactor is set to 100, MinTime is set to 20, and ConfNum is set to 0. It is advisable to use this option only after having first tried without it, since it will result in much longer execution times for easy examples.

Setting `Huge` to `true` doubles the size of the hash tables and `MaxRelations` over the `Large` parameter. As with the `Large` parameter, it is advisable to use this option only after having first tried without it.

Limit the maximum number of reduction equations to `MaxRelations`.

After finding n new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.) The word-differences arising from the equations are calculated after each such tidying and the number reported if verbose printing is on. The best strategy in general is to try a small value of `TidyInt` first and, if that is not successful, try increasing it. Large values such as 1000 work best in really difficult examples.

Give an ordering for the generators of P. This ordering affects the ordering of words in the alphabet. If not specified, the ordering defaults to $[g_1, {g_1}^{-1}, \ldots, g_n, {g_n}^{-1}]$ where $g_1, \ldots, g_n$ are the generators of F.

Limit the maximum number of word differences to `MaxWordDiffs`. The default behaviour is to increase the number of allowed word differences dynamically as required, and so usually one does not need to set this option.

These options are experimental halting options. `HaltingFactor` is a positive integer representing a percentage. After each tidying it is checked whether both the number of equations and the number of states have increased by more than `HaltingFactor` percent since the number of word-differences was last less than what it is now. If so the program halts. A sensible value seems to be 100, but occasionally a larger value is necessary. If the `MinTime` option is also set then halting only occurs if at least `MinTime` seconds of cpu-time have elapsed altogether. This is sometimes necessary to prevent very early premature halting. It is not very satisfactory, because of course the cpu-time depends heavily on the particular computer being used, but no reasonable alternative has been found yet.

`SetVerbose("KBMAG", v): MonStgElt, RngIntElt`

Set the verbose printing level for the Knuth–Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for v are 0 to 3 inclusive. Setting v to 0 corresponds to the '-silent' option of KBMAG in which no extra output is printed. Setting v to 2 corresponds to the '-v' (verbose) option of KBMAG in which a small amount of extra output is printed. Setting v to 3 corresponds to the '-vv' (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

Example: Modifying Limits for a Knot Group (ex-048dc9)

We attempt to construct an automatic structure for one of Listing's knot groups.

```
> F := Group< d, f | f*d*f^-1*d*f*d^-1*f^-1*d*f^-1*d^-1 = 1 >;
> SetVerbose("KBMAG", 1);
> b, G := IsAutomaticGroup(F);

Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Maximum number of equations exceeded.
#Halting with 195 equations.
#First word-difference machine with 45 states computed.
#Second word-difference machine with 53 states computed.
```

(continues on next page)

(continued from previous page)

```
> b;
false;
```

So this attempt has failed. We run the `IsAutomaticGroup` function again setting `Large` to `true`. This time we succeed.

```
> f, G := IsAutomaticGroup(F : Large := true);

Running Knuth-Bendix with the following parameter values
MaxRelations  = 262144
MaxStates     = 0
TidyInt       = 500
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Halting with 3055 equations.
#First word-difference machine with 49 states computed.
#Second word-difference machine with 61 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 101 states computed.
#General multiplier with 497 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 509 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 521 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 525 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 835 states computed.
#Checking inverse and short relations.
#Checking relation:  _3*_1*_4*_1*_3 = _1*_3*_2*_3*_1
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ d, d^-1, f, f^-1 ]
The second word difference machine has 89 states.
The word acceptor has 101 states.
```

Example: Automatic Group of the Trefoil Knot (ex-7b6c80)

We construct the automatic group corresponding to the fundamental group of the trefoil knot. A generator order is specified.

```
> F<a, b> := Group< a, b | b*a^-1*b=a^-1*b*a^-1 >;
> SetVerbose("KBMAG", 1);
> f, G := IsAutomaticGroup(F: GeneratorOrder := [a,a^-1, b, b^-1]);

Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Halting with 83 equations.
#First word-difference machine with 15 states computed.
#Second word-difference machine with 17 states computed.
```

(continues on next page)

(continued from previous page)

```
#System is confluent, or halting factor condition holds.
#Word-acceptor with 15 states computed.
#General multiplier with 67 states computed.
#Multiplier incorrect with generator number 4.
#General multiplier with 71 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 361 states computed.
#Checking inverse and short relations.
#Checking relation:  _3*_2*_3 = _2*_3*_2
#Axiom checking succeeded.

> G;

An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
The second word difference machine has 21 states.
The word acceptor has 15 states.
```

84.2.3 Accessing Group Information

The functions in this section provide access to basic information stored for an automatic group G.

`G . i: GrpRWS, RngIntElt -> GrpRWSElt`

The i-th defining generator for G. The integer i must lie in the range $[-r, r]$, where r is the number of group G.

`Generators``(G): GrpRWS -> [GrpRWSElt]`

A sequence containing the defining generators for G.

`NumberOfGenerators``(G): GrpRWS -> RngIntElt`

`Ngens``(G): GrpRWS -> RngIntElt`

The number of defining generators for G.

Example: Access Functions for the Von Dyck Group `(ex-94d627)`

We illustrate the access operations using the Von Dyck (2,3,5) group (isomorphic to A_5).

```
> F<a, b> := FreeGroup(2);
> Q := quo< F | a*a = 1, b*b = b^-1, a*b^-1*a*b^-1*a = b*a*b*a*b>;
> f, G<a, b> := IsAutomaticGroup(Q);
> G;

An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
The second word difference machine has 33 states.
The word acceptor has 28 states.

> print G.1*G.2;
a * b
> print Generators(G);
[ a, b ]
> print Ngens(G);
2
> rels := Relations(G);
> print rels[1];
Q.2 * Q.2^-1 = Id(Q)
> print rels[2];
```

(continues on next page)

(continued from previous page)

```
Q.2^-1 * Q.2 = Id(Q)
> print rels[3];
Q.1^2 = Id(Q)
> print rels[4];
Q.2^2 = Q.2^-1
> print Nrels(G);
18
> print Ordering(G);
ShortLex
```

FPGroup(G): GrpAtc -> GrpFP, Map

Returns the finitely presented group F used in the construction of G, and the isomorphism from F to G.

WordAcceptor(G): GrpAtc -> Rec

A record describing the word acceptor automaton stored in G.

WordAcceptorSize(G): GrpAtc -> RngIntElt, RngIntElt

The number of states of the word acceptor automaton stored in G, and the size of the alphabet of this automaton.

WordDifferenceAutomaton(G): GrpAtc -> Rec

A record describing the word difference automaton stored in G.

WordDifferenceSize(G): GrpAtc -> RngIntElt, RngIntElt

The number of states of the 2nd word difference automaton stored in G, and the size of the alphabet of this automaton.

WordDifferences(G): GrpAtc -> SeqEnum

The labels of the states of the word difference automaton stored in G. The result is a sequence of elements of the finitely presented group used in the construction of G.

GeneratorOrder(G): GrpAtc -> SeqEnum

The value of the `GeneratorOrder` parameter used in the construction of G. The result is a sequence of generators and their inverses from the finitely presented group used in the construction of G.

84.2.4 Properties of an Automatic Group

IsFinite(G): GrpRWS -> BoolElt, RngIntElt

Given an automatic group G return `true` if G has finite order and `false` otherwise. If G does have finite order also return the order of G.

Order(G): GrpRWS -> RngIntElt

 # G: GrpRWS -> RngIntElt

The order of the group G as an integer. If the order of G is known to be infinite, the symbol ∞ is returned.

Example: Order of an Infinite Automatic Group (ex-0ae3d4)

We construct the group $\mathbb{Z} \wr C_2$ and compute its order. The result of `Infinity` indicates that the group has infinite order.

```
> F<a,b,t> := FreeGroup(3);
> Q := quo< F | t^2 = 1, b*a = a*b, t*a*t = b>;
> SetVerbose("KBMAG", 1);
> f, G := IsAutomaticGroup(Q);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
```

(continues on next page)

(continued from previous page)

```
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#System is confluent.
#Halting with 14 equations.
#First word-difference machine with 14 states computed.
#Second word-difference machine with 14 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 6 states computed.
#General multiplier with 27 states computed.
#Validity test on general multiplier succeeded.
#Checking inverse and short relations.
#Axiom checking succeeded.
> Order(G);
Infinity
```

Example: Finiteness of a Triple Cover of A6 (ex-12f166)

We construct a three fold cover of A_6 and check whether it has finite order.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | a^3 = 1, b^3 = 1, (a*b)^4 = 1, (a*b^-1)^5 = 1>;
> SetVerbose("KBMAG", 1);
> f, G := IsAutomaticGroup(F : GeneratorOrder := [a, b, a^-1, b^-1]);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#System is confluent.
#Halting with 183 equations.
#First word-difference machine with 289 states computed.
#Second word-difference machine with 360 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 314 states computed.
#General multiplier with 1638 states computed.
#Multiplier incorrect with generator number 4.
#General multiplier with 1958 states computed.
#Multiplier incorrect with generator number 2.
#General multiplier with 2020 states computed.
#Multiplier incorrect with generator number 1.
#General multiplier with 2038 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 4252 states computed.
#Checking inverse and short relations.
#Checking relation:  _1*_2*_1*_2 = _4*_3*_4*_3
#Checking relation:  _1*_4*_1*_4*_1 = _2*_3*_2*_3*_2
#Axiom checking succeeded.
> IsFinite(G);
true 1080
> isf, ord := IsFinite(G);
> isf, ord;
true 1080
```

84.2.5 Constructing Words

`G ! [ i1, ..., is ]: GrpAtc, [ RngIntElt ] -> GrpAtcElt`

Given an automatic group G defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[-r, r]$, excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \cdots * G.|i_s|^{\epsilon_s}$$

where ϵ_j is $+1$ if i_j is positive, and -1 if i_j is negative. The word will be returned in reduced form.

`Identity(G): GrpAtc -> GrpAtcElt`

`Id(G): GrpAtc -> GrpAtcElt`

`G ! 1: GrpAtc, RngIntElt -> GrpAtcElt`

Construct the identity word in the automatic group G.

`Parent(w): GrpAtcElt -> GrpAtc`

The parent group G for the word w.

Example: Constructing Words in an Automatic Group (ex-bccdb0)

We construct some words in a two-generator two-relator group.

```
> F<a, b> := Group< a, b | a^2 = b^2, a*b*a = b*a*b >;
> f, G<a, b> := IsAutomaticGroup(F);
> G;
An automatic group.
Generator Ordering = [ $.1, $.1^-1, $.2, $.2^-1 ]
The second word difference machine has 11 states.
The word acceptor has 8 states.

> Id(G);
Id(G)
> print G!1;
Id(G)
> a*b*a*b^3;
a^4 * b * a
> G![1,2,1,2,2,2];
a^4 * b * a
```

84.2.6 Operations on Elements

Having constructed an automatic group G one can perform arithmetic with words in G. Assuming we have $u, v \in G$ then the product $u * v$ will be computed as follows:

(i) The product $w = u * v$ is formed as a product in the appropriate free group.

(ii) w is reduced using the second word difference machine associated with G.

Note that:

(i) Reduction of w can cause an increase in the length of w. At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.

(ii) The implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v: GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return their product.

`u / v: GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return the product of the word u by the inverse of the word v, i.e. the word $u * v^{-1}$.

`u ^ n: GrpRWSElt, RngIntElt -> GrpRWSElt`

The n-th power of the word w.

`u ^ v: GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return the conjugate of the word u by the word v, i.e. the word $v^{-1} * u * v$.

`Inverse(w): GrpRWSElt -> GrpRWSElt`

The inverse of the word w.

`(u, v): GrpRWSElt, GrpRWSElt -> GrpRWSElt`

Given words w and v belonging to a common group, return the commutator of the words u and v, i.e., the word $u^{-1}v^{-1}uv$.

`(u`$_1$`, ..., u`$_r$`): GrpRWSElt, ..., GrpRWSElt -> GrpRWSElt`

Given r words $u_1, \ldots, u_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`u eq v: GrpRWSElt, GrpRWSElt -> BoolElt`

Given words w and v belonging to the same group, return `true` if w and v reduce to the same normal form, `false` otherwise. If G is confluent this tests for equality. If G is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v: GrpRWSElt, GrpRWSElt -> BoolElt`

Given words w and v belonging to the same group, return `false` if w and v reduce to the same normal form, `true` otherwise. If G is confluent this tests for non-equality. If G is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w): GrpRWSElt -> BoolElt`

`IsIdentity(w): GrpRWSElt -> BoolElt`

Returns `true` if the word w is the identity word.

`# u: GrpRWSElt -> RngIntElt`

The length of the word w.

`ElementToSequence(u): GrpRWSElt -> [ RngIntElt ]`

`Eltseq(u): GrpRWSElt -> [ RngIntElt ]`

The sequence Q obtained by decomposing the element u of a rewrite group into its constituent generators and generator inverses. Suppose u is a word in the rewrite group G. Then, if $u = G.i_1^{e_1} \cdots G.i_m^{e_m}$, with each $e_i = \pm 1$, then $Q[j] = i_j$ if $e_j = +1$ and $Q[j] = -i_j$ if $e_j = -1$, for $j = 1, \ldots, m$.

Example: Word Operations in an Automatic Group (ex-923349)

We illustrate the word operations by applying them to elements of the fundamental group of a 3-manifold.

We illustrate the word operations by applying them to elements of a free group of rank two (with lots of redundant generators).

```
> FG<a,b,c,d,e> := FreeGroup(5);
> F := quo< FG | a*d*d = 1, b*d*d*d = 1, c*d*d*d*d*d*e*e*e = 1>;
> f, G<a,b,c,d,e> := IsAutomaticGroup(F);
> G;

An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1, e, e^-1 ]
```

(continues on next page)

(continued from previous page)

```
The second word difference machine has 41 states.
The word acceptor has 42 states.

> print a*d;
d^-1
> print a/(d^-1);
d^-1
> print c*d^5*e^2;
e^-1
> print a^b, b^-1*a*b;
a a
> print (a*d)^-2, Inverse(a*d)^2;
a^-1 a^-1
> print c^-1*d^-1*c*d eq (c,d);
true
> print IsIdentity(b*d^3);
true
> print #(c*d*d*d*d*d*e*e);
1
```

84.2.7 Set Operations

In this section we describe functions which allow the user to enumerate various sets of elements of an automatic group G.

Random(G, n): GrpAtc, RngIntElt -> GrpAtcElt

A random word of length at most n in the generators of G.

Random(G): GrpAtc -> GrpAtcElt

A random word (of length at most the order of G) in the generators of G.

Representative(G): GrpAtc -> GrpAtcElt

Rep(G): GrpAtc -> GrpAtcElt

An element chosen from G.

Set(G, a, b): GrpAtc, RngIntElt, RngIntElt -> SetEnum

Search: MonStgElt	Default: "DFS"

Create the set of words, w, in G with $a \leq \text{length}(w) \leq b$. If Search is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If Search is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

Set(G): GrpAtc -> SetEnum

Search: MonStgElt	Default: "DFS"

Create the set of words that is the carrier set of G. If Search is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If Search is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Seq(G, a, b): GrpAtc, RngIntElt, RngIntElt -> SeqEnum`

`Search: MonStgElt`	`Default: "DFS"`

Create the sequence S of words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

`Seq(G): GrpAtc -> SeqEnum`

`Search: MonStgElt`	`Default: "DFS"`

Create a sequence S of words from the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Example: Set Operations on a Dihedral Group (ex-dc57c0)

We construct the group D_{22}, together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FG<a,b,c,d,e,f> := FreeGroup(6);
> F := quo< FG | a*c^-1*a^-1*d = 1, b*f*b^-1*e^-1 = 1,
>                c*e*c^-1*d^-1 = 1, d*f^-1*d^-1*a = 1,
>                e*b*e^-1*a^-1 = 1, f*c^-1*f^-1*b^-1 = 1 >;
> SetVerbose("KBMAG", 1);
> f, G<a,b,c,d,e,f> := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#System is confluent.
#Halting with 41 equations.
#First word-difference machine with 16 states computed.
#Second word-difference machine with 17 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 6 states computed.
#General multiplier with 58 states computed.
#Validity test on general multiplier succeeded.
#Checking inverse and short relations.
#Axiom checking succeeded.
> Representative(G);
Id(G)
> Random(G);
a*c;
> Random(G, 5);
a * b
> Set(G);
{ a * d * b, a * b, a * b * e, a * c, a * d, d * b, b * e,
  a * b * a, a * b * d, b * a, a * c * e, Id(G), b * d, c * e,
  e, f, a, a * e, b, c, a * f, d }
> Seq(G : Search := "BFS");
[ Id(G), a, b, c, d, e, f, a * b, a * c, a * d, a * e, a * f,
```

(continues on next page)

(continued from previous page)

```
  b * a, b * d, b * e, c * e, d * b, a * b * a, a * b * d,
  a * b * e, a * c * e, a * d * b ]
```

84.2.8 Homomorphisms

For a general description of homomorphisms, we refer to chapter Mappings. This section describes some special aspects of homomorphisms whose domain or codomain is an automatic group.

Groups in the category `GrpAtc` currently are accepted as codomains only in some special situations. The most important cases in which an automatic group can be used as a codomain are group homomorphisms whose domain is in one of the categories `GrpFP`, `GrpGPC`, `GrpRWS`, or `GrpAtc`.

`hom< A -> G | S >: Struct, Struct -> Map`

Returns the homomorphism from the automatic group A to the group G defined by the expression S which can be the one of the following:

(i) A list, sequence, or indexed set containing the images of the n generators $A.1, \ldots, A.n$ of A. Here, the i-th element of S is interpreted as the image of $A.i$, that is, the order of the elements in S is important.

(ii) A list, sequence, enumerated set, or indexed set, containing n tuples $\langle x_i, y_i \rangle$ or arrow pairs $x_i \to y_i$, where x_i is a generator of A and $y_i \in G$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of A. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor.

Note that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of A.

84.2.9 The Growth Function

`GrowthFunction(G): GrpAtc -> FldFunRatElt`

```
Primes: SeqEnum                    Default: [ ]
```

Compute the growth function of the word acceptor automaton associated with G. The growth function of a DFA, A, is the quotient of two integral polynomials in a single variable x. The coefficient of x^n in the Taylor expansion (about 0) of this quotient is equal to the number of words of length n accepted by A. That is, the result is a closed form for this generating function.

The algorithm is due to Derek Holt. The `Primes` parameter is no longer used, but is kept for backward compatibility. It may be removed in future releases.

Example: Growth Function of a Dihedral Group `(ex-dfea9f)`

We construct a dihedral group of order 10 and compute the growth function of its word acceptor. As the group is finite, the result will be a polynomial. Note here that the `R!f` is only necessary to get pretty printing, specifically to ensure that f is printed in the variable x.

```
> R<x> := RationalFunctionField(Integers());
> FG<a,b> := FreeGroup(2);
> Q := quo< FG | a^5, b^2, a^b = a^-1>;
> G := AutomaticGroup(Q);
> f := GrowthFunction(G);
> R!f;
2*x^3 + 4*x^2 + 3*x + 1
```

Now we take as example an infinite dihedral group. The group is infinite, so the result cannot be polynomial. We then extract the coefficients of the growth function for word lengths 0 to 14.

```
> FG2<d,e> := FreeGroup(2);
> Q2 := quo<FG2| e^2, d^e = d^-1>;
> G2 := AutomaticGroup(Q2);
> f2 := GrowthFunction(G2);
> R!f2;
(-x^2 - 2*x - 1)/(x - 1)
> PSR := PowerSeriesRing(Integers():Precision := 15);
> Coefficients(PSR!f2);
[ 1, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

84.3 Hyperbolic Groups

For an introduction to the theory of hyperbolic groupss (also known as *word-hyperbolic* or sometimes *Gromov hyperbolic* groups), see for example [HRR17].

The problem of determining whether or not a finitely presented group is *hyperbolic* is undecidable in general, but if the group is hyperbolic, then it is possible in principle to verify this property,

In the following two sections, we describe two Magma intrinsics for attempting to verify that a group is hyperbolic. The first of these requires the prior construction of an automatic structure for the group, whereas the second can be applied directly to a finitely presented group.

84.3.1 Testing an Automatic Group for Hyperbolicity

In this section we describe the version of the intrinsic `IsHyperbolic` that can applied to an automatic group.

The other version of `IsHyperbolic`, described below in Section *Testing a Finitely Presented Group for Hyperbolicity*, can be applied to any finitely presented group. It succeeds less often than the hyperbolicity test for an automatic group when applied to hyperbolic groups having a small number of generators, but is much faster when it does succeed, and it does not require the prior computation of an automatic structure for G.

`IsHyperbolic(G): GrpAtc -> BoolElt, Rec`

```
MaxTries   : RngIntElt                    Default: 10
MaxBadWords: RngIntElt                    Default: 200
```

This intrinsic attempts to prove that the automatic group G is hyperbolic. If `true` is returned, then the group has been proved to be hyperbolic, and a finite state automaton that accepts all geodesic words over the group generators is also returned.

If `false` is returned, then this means that G has not been proved to be hyperbolic. It does *not* mean that G is not hyperbolic, but the program has no possibility of proving it.

It works by making successive attempts to construct the automaton that accepts geodesic words, and it gives up and returns `false` if it fails to do this after `MaxTries` attempts. So it may still be possible to verify hyperbolicity by increasing the value of this parameter. When an attempt fails, a set of at most `MaxBadWords` words is found consisting of geodesic words that are not accepted by the current version of the putative geodesic word acceptor.

If the verbose flag "KBMAG" is set to 1 or more, then the number of states of the putative geodesic word acceptor is printed after each attempt, which might help the use to decide whether it might be worthwhile to try with an increased value of `maxtries`.

Example: Verbose Hyperbolicity Testing `(ex-29ea10)`

```
> G := Group< x,y | x^3, y^3, (x*y)^6, (x,y)^6 >;
> A := AutomaticGroup(G : Large:=true);
> SetVerbose("KBMAG",1);
> ish, GWA := IsHyperbolic(A);
Word acceptor has 459 states
There are 219 word differences
Geodesic word acceptor has 1028 states
Attempt number 1 failed. There are now 245 word differences
Geodesic word acceptor has 762 states
Attempt number 2 failed. There are now 251 word differences
Geodesic word acceptor has 508 states
Hyperbolicity proved after 3 attempts
> ish;
true
```

IsConjugate(G, x, y): GrpAtc, GrpAtcElt, GrpAtcElt -> BoolElt, GrpAtcElt

Test elements x and y of the hyperbolic automatic group G for conjugacy in G, and return `true` or `false`, together with an element g with $g^{-1}xg = y$ in the positive case.

This will only work if `IsHyperbolic`(G) returns `true`. The algorithm, which is due to Eric Swenson and Joe Marshall, is described in [Mar05]. and it can only be applied to elements of infinite order. It will exit with an error message if x and y have the same finite order.

The first call of this function on an automatic group G will require the computation of some automata, which may take some time. Subsequent calls on the same group G will be fast.

84.3.2 Testing a Finitely Presented Group for Hyperbolicity

This version of the intrinsic `IsHyperbolic` attempts to prove that a finitely-presented group is hyperbolic. It is based on a forthcoming paper by Holt, Linton, Neunhöffer, Parker, Pfeiffer, and Roney-Dougal, provisionally entitled *A New Approach to Proving Hyperbolicity*.

If `IsHyperbolic` returns `true` then the given group is guaranteed to have a linear Dehn function, and hence be hyperbolic. It may fail and return `false`, in which case the group may still be hyperbolic: if it fails, then it returns extra data on the cause of failure, which may help the user to modify the presentation and try again.

If `IsHyperbolic` succeeds, then the value of the parameter ϵ (written as `epsilon` in the intrinsic) yields information about the group. In particular, let r be the length of the longest relator in `green`, then the Dehn function of the group is bounded above by

$$f(n) = n(4 + r + \frac{3+r}{2\epsilon}) - \frac{3+r}{\epsilon}.$$

By default, if `IsHyperbolic` succeeds, then it also attempts to compute a Dehn algorithm for the group, which can be used to test whether words in the group generators define the identity element of the group. This attempt is usually successful but may fail.

The alternative intrinsic, also called `IsHyperbolic`, for attempting to prove that an automatic group is hyperbolic was described above in Section *Testing an Automatic Group for Hyperbolicity*). To use that version, it is necessary to first convert the finitely presented group G into an automatic group, by using the function `AutomaticGroup(G)`. For groups with a moderately small number of generators, applying `IsHyperbolic` to an automatic group will succeed on more examples than the version of `IsHyperbolic` that we describe in this section. But the version in this section will typically be much faster if it does work, and it does not require prior computation of an automatic structure for the finitely-presented group G. Also the version described here can be applied to groups with large numbers (several thousands) of generators, and it can be used for investigating the asymptotic behaviour of such groups.

IsHyperbolic(F, red, green): GrpFP, SeqEnum, SeqEnum -> BoolElt, BoolElt, SeqEnum, SeqEnum

```
epsilon     : FldRatElt                    Default: 1/10
VerifySolver: BoolElt                      Default: true
Print       : RngIntElt                    Default: 0
noboundary  : BoolElt                      Default: false
verifypgp   : BoolElt                      Default: true
complete    : BoolElt                      Default: false
```

The group F should be a free group, and `red` and `green` should be (possibly empty) lists A and B of elements of F that define a finitely presented group $G = F/\mathrm{ncl}_F\langle A \cup B\rangle$. The words in `red` must all have length 3.

The easiest way to use `IsHyperbolic` is to put all relators into `green`, leaving `red` empty. The following example shows that $\langle x, y, z | (xyz)^2\rangle$ is hyperbolic:

Example: Hyperbolicity of a One-Relator Group (ex-d7a495)

```
> F<x, y, z>:= FreeGroup(3);
> red:= [];
> green:= [(x*y*z)^2];
> ish, dehn, D1, D2 := IsHyperbolic(F, red, green);
> ish, dehn;
true true
```

The variables `ish` and `dehn` are both `true`, which means that the group has been proved hyperbolic and a Dehn algorithm has been calculated. The data returned as `D1` can be used to decide whether words in F define the identity element of G, using the function `IsIdentity` described in Section *The Dehn Algorithm* below.

The intrinsic `IsHyperbolic` can be applied to a group G that is already constructed in Magma as a group of type `FPGroup`, as follows:-

Example: Hyperbolicity from Group Relators (ex-b541e8)

```
> F<a, b, c>:= FreeGroup(3);
> G:= quo<F | a^5, (b*c)^7>;
> ish, dehn, L1, L2 := IsHyperbolic(F, [], Relators(G));
> ish, dehn;
true true
```

84.3.3 Common Error Messages, and How to Avoid Them

The function will return an error if `green` contains any relators of the form ab with $a \neq b$. The intrinsic *Simplify* should be applied before invoking `IsHyperbolic` in this case.

Example: Error for Short Relators (ex-cd92b6)

```
> F<a, b, c>:= FreeGroup(3);
> green:= [a*b, (a*b*c)^6];
> flag:= IsHyperbolic(F, [], green);
Runtime error: Non-square relators must have length at least 3
> G:= Simplify(quo<F|green>);
> G;
Finitely presented group on 2 generators
Generators as words
    G.1 = $.1
    G.2 = $.3
Relations
    G.2^6 = Id($)
> flag:= IsHyperbolic(FreeGroup(G), [], Relators(G));
```

(continues on next page)

(continued from previous page)

```
> flag;
true
```

The function will also return an error if `green` contains any pairs of relators (or their inverses) that are identical in all but one letter. Again the function `Simplify` may be able to resolve the problem.

Example: Error for Large Relator Pieces (ex-976118)

```
> F<a, b, c>:= FreeGroup(3);
> green:= [(a*b*c)^2, a*b*c*a*b*b];
> IsHyperbolic(F, [], green);
Runtime error: A piece of a relator consists of all but one letter
> G:= Simplify(quo<F|green>);
> G;
Finitely presented group G on 2 generators
Generators as words
  G.1 = $.1
  G.2 = $.2
Relations
  (G.1 * G.2^2)^2 = Id(G)
> IsHyperbolic(FreeGroup(G), [], Relators(G));
true
```

84.3.4 The Relators red

The `IsHyperbolic` function is based on a generalisation of the ideas of small cancellation theory. If a presentation contains very short relators then it is harder for small cancellation conditions to be satisfied. However, it is possible for `IsHyperbolic` to work on such presentations, by using the `red` relators. These relators are required to have length 3 and (together with any relators of the form x^2 in `green`) should define a pregroup structure on the group generators, their inverses, and the identity, as defined by Stallings.

A *pregroup* P is a set X, with a distinguished element 1, an involution σ, and a partial multiplication $(x, y) \to xy$ which is defined for $(x, y) \in D(P)$, satisfying five axioms. The first three of these axioms essentially state that σ operates in the same way as group inversion, and that 1 acts as an identity. The final two axioms are

(P4) if $(x, y), (y, z) \in D(P)$ then $(xy, z) \in D(P)$ if and only if $(x, yz) \in D(P)$, in which case $(xy)z = x(yz)$; and

(P5) if $(x, y), (y, z), (z, t) \in D(P)$ then at least one of (xy, z) and (yz, t) is in $D(P)$.

These axioms are checked unless the parameter `verifypgp` is set to `false`.

Any group which is a quotient of a free product of free and finite groups can be defined by a presentation with relators in `red` and `green`, satisfying a pregroup structure. It is also possible to present any group which is a quotient of a virtually free group by including additional relators in this way, but the choice of relators to put in `red` may be more complicated.

The following example shows that the quotient $\langle a, b | a^2, b^3, (ab)^7 \rangle$ of the modular group $C_2 * C_3$ is hyperbolic.

Example: Splitting Relators into Red and Green (ex-5a8018)

```
> F<a, b>:= FreeGroup(2);
> ish:= IsHyperbolic(F, [], [a^2, b^3, (a*b)^7]);
> ish;
false
> red:= [b^3];
> green:= [a^2, (a*b)^7];
```

(continues on next page)

(continued from previous page)

```
> ish, dehn := IsHyperbolic(F, red, green);
> ish, dehn;
true true
```

As a more complicated example, the following example shows how to work with a quotient of $S_3 * C_3 * Z$, using the function `RedRelatorsForFreeProduct`.

`RedRelatorsForFreeProduct(groupList, freeFacs): List, RngIntElt -> GrpFP, SeqEnum, SeqEnum, Map, SeqEnum`

The list `groupList` should be a nonempty list of finite permutation groups $G_1, \ldots, G_t$, and the integer `freeFacs` should be an integer $k \geq 0$. It produces the free product $G_1 * G_2 * \cdots * G_t * F_k$ in a form that can be worked with by `IsHyperbolic`. The function `RedRelatorsForFreeProduct` will return a free group, a list of red relators, a list of involutory generators (to which additional green relators can be added by the user) and a function for turning lists of elements from the groups G_i and F_k into relators over the free product. The final return value is a list of generators for each group, in the order to which they correspond to the generators of the new free group F, and may be helpful for checking accuracy.

Example: Red Relators for a Free Product (ex-fac47e)

```
> groupList:= [*Sym(3), CyclicGroup(3)*];
> F, red, green, GreenRelator, gens:=
>     RedRelatorsForFreeProduct(groupList, 1);
> F;
Finitely presented group F on 6 generators (free)
> ish, dehn := IsHyperbolic(F, red, green);
> ish, dehn;
true true
```

This is unsurprising, since only a free product has been constructed at this point! The next step is to define an additional relator, using the returned function `GreenRelator`:

```
> extra_rel:= GreenRelator([* < 1, Sym(3)!(1, 3, 2)>,
>   <3, 1>, <2, CyclicGroup(3)!(1, 2, 3)>, <3, -1>,
>   <2, CyclicGroup(3)!(1, 3, 2)>, <1, Sym(3)!(1, 2)>, <3, 1>*]);
> extra_rel;
F.1^-1 * F.6 * F.5 * F.6^-1 * F.5^-1 * F.3 * F.6
> Append(~green, extra_rel);
> ish, dehn, D1, D2 := IsHyperbolic(F, red, green);
> ish, dehn;
true false
```

So this is an example where hyperbolicity was proved, but `IsHyperbolic` failed to compute a Dehn algorithm. In this case, the data items `D1` and `D2` provide extra information on the cause of this failure as described in the forthcoming paper mentioned earlier. If it is not known how to represent the input group as a quotient of a free product of groups, and `IsHyperbolic` fails to succeed with the given presentation, some general tips are:

1. It is advisable to include all relators of the form x^3 in `red`.
2. If there are relators x^k for small values of k ($k \leq 6$), then it may be a good idea to introduce new generators so that the relator x^k can be broken up into relators of length 3, all of which can be put into `red`.

This is illustrated in the following example.

Example: Breaking Up a Relator (ex-2ba258)

```
> F<x,y> := FreeGroup(2);
> red := [y^3]; green := [x^4, (x*y)^4 ];
> ish := IsHyperbolic(F, red, green);
> ish;
false
```

That didn't work, but let's try breaking up the relator x^4.

```
> F<x,y,t> :=FreeGroup(3);
> red := [y^3, x^2*t]; green := [t^2, (x*y)^4];
> ish, dehn := IsHyperbolic(F, red, green);
> ish, dehn;
true true
```

However, it will not always work simply to put all relators of length 3 into `red`, as the conditions for being a pregroup may fail.

```
> F<a,b,c,d,e> := FreeGroup(5);
> red := [a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*a^-1, e*a*b^-1 ];
> IsHyperbolic(F,red,[]);
Runtime error: Pregroup axiom (P5) fails on generators <1, 2, 3, 4>
```

Here is another example, where we use the same technique to prove that the group $\langle x, y \mid x^4, y^5, (xy)^7, [x,y]^3 \rangle$ is hyperbolic. Note that the alternative intrinsic `IsHyperbolic` described in Section *Testing an Automatic Group for Hyperbolicity* for proving hyperbolicity of automatic groups succeeds on the groups $\langle x, y \mid x^4, y^5, (xy)^k, [x,y]^3 \rangle$ for all $k \geq 4$.

Example: Hyperbolicity with Several Red Relators (ex-7ee847)

```
> F<x,t,y,u> := FreeGroup(4);
> ish, dehn := IsHyperbolic(F, [x^2*t, y^2*u, u^-2*y], [t^2, (x*y)^7, (x,y)^3]);
> ish, dehn;
true true
```

84.3.5 The Parameter epsilon

The intrinsic `IsHyperbolic` works by attempting to move positive curvature in van Kampen diagrams for the group presentation to the boundary of these diagrams. The aim is to ensure that, after this curvature shifting, all non-boundary regions of a diagram that are labelled by green relators have negative curvature of at most $-$`epsilon`, where `epsilon` is a small positive number that can be adjusted by the user. Its default value is 1/10.

If `IsHyperbolic` fails, then the third and fourth return values describe how it failed, in the form of a list of lists and a list of places. If the fourth value in the third entry is less than the value of `epsilon` that was used, then running again with a smaller value of `epsilon` *might* enable it to succeed. Conversely, if it succeeds with the default value, then the user may wish to try again with a larger value of ϵ, to deduce lower bounds for the Dehn function.

Example: Varying the Epsilon Parameter (ex-9972b6)

```
> F<a, b>:= FreeGroup(2);
> ish, dehn, D1, D2:= IsHyperbolic(F, [b^3], [a^2, (a*b)^7]: epsilon:= 1/5);
> ish, dehn;
false false
> D1;
[
    [ 1, 0, 0, 0, 0, 0 ],
    [ 1, 2, 1, 1/210, 1, 2 ],
```

(continues on next page)

(continued from previous page)

```
    [ 1, 4, 2, 1/105, 2, 2 ],
    [ 1, 6, 3, 1/70, 3, 2 ],
    [ 1, 8, 4, 2/105, 4, 2 ],
    [ 1, 10, 5, 1/42, 5, 2 ],
    [ 1, 12, 6, 1/35, 6, 2 ],
    [ 1, 14, 7, 1/30, 7, 2 ]
]
> ish, dehn, D1, D2:= IsHyperbolic(F, [b^3], [a^2, (a*b)^7]: epsilon:= 1/6);
> ish, dehn;
true true
```

And it worked this time!

84.3.6 The Dehn Algorithm

`IsIdentity(w, D1): GrpFPElt, Rec -> BoolElt, GrpFPElt, SeqEnum, SeqEnum`

If `IsHyperbolic` has proved that a group G is hyperbolic and computed a Dehn algorithm for G, then the data returned as the third return value can be used to apply the Dehn algorithm to words in the free group, and to test whether they map onto the identity element of G. The result of application of the Dehn algorithm to the word is also returned. In general, it is possible for different representatives of the same group element to reduce to different words, so this reduction cannot be used as a normal form for group elements.

Example: Word Problem via the Dehn Algorithm (ex-1c88ea)

```
> F<a, b>:= FreeGroup(2);
> ish, dehn, D1, D2:= IsHyperbolic(F, [b^3], [a^2, (a*b)^7]);
> ish, dehn;
true true
> isid, wd := IsIdentity((a*b^2)^6,D1);
> isid, wd;
false b * a
> isid, wd := IsIdentity(b^-1*(a*b^2)^6*a,D1);
> isid, wd;
true Id(F)
```

84.4 Bibliography

- [Hol97] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.
- [HRR17] Derek F. Holt, Sarah Rees, and Claas E. Röver. *Groups, languages and automata*. Volume 88 of London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 2017.
- [Mar05] Joseph Marshall. Computational problems in hyperbolic groups. *Internat. J. Algebra Comput.*, 15(1):1–13, 2005. URL: https://doi.org/10.1142/S0218196705002086, doi:10.1142/S0218196705002086.

Chapter 85

Groups of Straight-Line Programs

85.1 Introduction

This Chapter describes the category of straight-line program groups (SLP-groups). A straight-line program is formally a sequence $[s_1, s_2, \ldots, s_n]$ such that each s_i is one of the following:

(i) A generator of the SLP-group;

(ii) A product $s_j s_k, \quad j < i, k < i$;

(iii) A power $s_j^n, \quad j < i$;

(iv) A conjugate $s_j^{s_k}, \quad j < i, k < i$.

Effectively, a straight-line program can be regarded as a word in the generators which is stored as an expression tree instead of a list of generator-exponent pairs.

The importance of such a category of groups is that storing a word as an expression tree allows much faster evaluation of homomorphisms given as the unique extension of a mapping of the generators into a group of any category, as common subexpressions may be computed once only, and powers or conjugates may be more efficiently computed in the target group than by a linear product of generators and their inverses.

The name in Magma for the category of SLP-groups is `GrpSLP`.

85.2 Construction of an SLP-Group and its Elements

85.2.1 Structure Constructors

`SLPGroup(n): RngIntElt -> GrpSLP`

Construct the free group F of straight-line programs on n generators, where n is a non-negative integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$.

Example: Creating an SLP-Group (ex-059102)

The statement

```
> F := SLPGroup(2);
```

creates the free group on two generators. Here the generators may be referenced using the standard names, `F.1` and `F.2`. Group operations on the elements will be stored as part of the result.

85.2.2 Construction of an Element

`Identity(G): GrpSLP -> GrpSLPElt`

`Id(G): GrpSLP -> GrpSLPElt`

`G ! 1: GrpSLP, RngIntElt -> GrpSLPElt`

Construct the identity element (the straight-line program [] of length 0) for the SLP-group G.

85.3 Arithmetic with Elements

`u * v: GrpSLPElt, GrpSLPElt -> GrpSLPElt`

Given straight-line programs $u = [u_1, \ldots, u_m]$ and $v = [v_1, \ldots, v_n]$ belonging to the same SLP-group G, return a straight-line program corresponding to the product of u and v. It is clear that the straight-line program $[u_1, \ldots, u_m, v_1, \ldots, v_n, u_m v_n]$ satisfies the formal definition. In practice, the u_i and v_i need not be distinct, so the resulting program may be shorter.

`u ^ m: GrpSLPElt, RngIntElt -> GrpSLPElt`

Given an integer m and a straight-line program u, return the straight-line program corresponding to the m-th power of u.

`u ^ v: GrpSLPElt, GrpSLPElt -> GrpSLPElt`

Given straight-line programs u and v, return the straight-line program corresponding to the conjugate of u by v.

`# u: GrpSLPElt -> RngIntElt`

Given a straight-line program u, return the number of multiplication, power, or conjugate operations required to evaluate a homomorphism on u.

85.3.1 Accessing the Defining Generators and Relations

The functions described here provide access to basic information stored for an SLP-group G.

`G . i: GrpSLP, RngIntElt -> GrpSLPElt`

The i-th generator for G.

`Generators(G): GrpSLP -> { GrpSLPElt}`

A set containing the generators for G.

`NumberOfGenerators(G): GrpSLP -> RngIntElt`

`Ngens(G): GrpSLP -> RngIntElt`

The number of generators for B.

`Parent(u): GrpSLPElt -> GrpSLP`

The parent group G of the straight-line program u.

85.4 Addition of Extra Generators

It is often the case that a particular expression in the original generators is important in some homomorphic image of the group in the sense that it will very often form a common subexpression of subsequent expression. For this reason, Magma provides facilities to build related SLP-groups in which particular words acquire generator status themselves.

`AddRedundantGenerators(G, Q): GrpSLP, [ GrpSLPElt ] -> GrpSLP`

An SLP-group H on $n + q$ generators (where G has n generators and Q has q elements). Furthermore, the identification of $G.i$ with $H.i$ ($i \leq n$) and of $Q[i]$ with $H.(n + i)$ is maintained, allowing coercion between G and H and also simple definition of homomorphisms.

85.5 Creating Homomorphisms

Because SLP-groups exist primarily to allow the user to write efficient code for evaluating words under a homomorphism, there are some extra features in the homomorphism constructor which rely on the user providing correct input.

When evaluating single words, it may not be desirable to explicitly construct the homomorphism. The `Evaluate` function uses the same evaluation mechanism as the homomorphisms and may a useful alternative.

`hom< G -> H | L: parameters>: GrpSLP, Grp -> Map`

`CheckCodomain: BoolElt` `Default: true`

Return the group homomorphism $\phi : G \to H$ defined by the list L. The list may contain:

(i) Elements of the codomain. This form can only be used when all the preceding entries have given the image of the corresponding generator of G;

(ii) Generator-image pairs of the form `G.i -> x` or `<G.i, x>`;

(iii) A homomorphism ψ from an SLP-group B to H where G has been defined as a result of adding redundant generators to B. If this item appears, it must appear first. After the remaining generators have been processed, any images which are not yet assigned are computed from ψ. If the parameter `CheckCodomain` has the value `false`, then it is assumed that the generator images lie in the codomain.

`Evaluate(u, Q): GrpSLPElt, [ GrpElt ] -> GrpElt`

`Evaluate(u, G): GrpSLPElt, Grp -> GrpElt`

`Evaluate(v, Q): [ GrpSLPElt ], [ GrpElt ] -> GrpElt`

`Evaluate(v, G): [ GrpSLPElt ], Grp -> GrpElt`

Evaluate the word u using the elements of Q as images of the generators of the parent of u. The sequence Q must contains at least as many group elements as the parent of u has generators.

The second form evaluates all the words in v simultaneously, which is usually quicker than doing individual evaluations.

When the second argument is a group G, Q is taken as the sequence of generators of G.

Example: Homomorphisms with Redundant Generators (ex-efbc6c)

An illustration of the use of `AddRedundantGenerators` and the homomorphism constructing machinery.

```
> G := SLPGroup(2);
> M := GeneralLinearGroup(19, 7);
> P := RandomProcess(G);
> x := Random(P);
> #x;
74
```

We evaluate x im M using the `Evaluate` function.

```
> m := Evaluate(x, [M.1, M.2]);
> Order(m);
118392315154200
```

If we wish to evaluate several different words, we may be better off using a homomorphism.

```
> Q := [x^G.1, x^G.2, x^(G.1*G.2)];
> phi := hom<G -> M | M.1, M.2>;
> time R1 := phi(Q);
Time: 0.129
```

We note that x has become important since it is now a common sub-expression of several straight-line programs. We can build a homomorphism which will store the image of x by adding x as a redundant generator and defining the same homomorphism from the resulting group.

```
> H := AddRedundantGenerators(G, [x]);
> QQ := [H | x: x in Q];
```

We will define psi as the unique map on H which matches phi.

```
> psi := hom<H -> M | phi>;
> time R2 := psi(QQ);
Time: 0.000
> R1 eq R2;
true
```

In fact, if we had looked at the expression lengths of the straight-line programs involved, we would have found the following, which explains the significant speed up:

```
> [#x: x in Q];
[ 75, 75, 75 ]
> [#x: x in QQ];
[ 1, 1, 2 ]
```

85.6 Operations on Elements

85.6.1 Equality and Comparison

`u eq v: GrpSLPElt, GrpSLPElt -> BoolElt`

Returns `true` if and only if the straight-line programs u and v are identical. Identical here means they are identical as expression trees, not that they will always evaluate to the same word in the generators.

`u ne v: GrpSLPElt, GrpSLPElt -> BoolElt`

Returns `true` if and only if the straight-line programs u and v are not identical. Identical here means they are identical as expression trees, not that they will always evaluate to the same word in the generators.

85.7 Set-Theoretic Operations

85.7.1 Membership and Equality

`g in G: GrpSLPElt, GrpSLP -> BoolElt`

Given a straight-line program g and an SLP-group G, return `true` if g is an element of G, `false` otherwise.

`g notin G: GrpSLPElt, GrpSLP -> BoolElt`

Given an straight-line program g and an SLP-group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G: { GrpSLPElt}, GrpSLP -> BoolElt`

Given a group G and a set S of elements belonging to a group H, where G and H are related, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G: { GrpSLPElt}, GrpSLP -> BoolElt`

Given a group G and a set S of elements belonging to a group H, where G and H are related, return `true` if S is not a subset of G, `false` otherwise.

85.7.2 Set Operations

`RandomProcess(G): GrpSLP -> Process`

```
Slots   : RngIntElt                    Default: 10
Scramble: RngIntElt                    Default: 100
```

Create a process to generate randomly chosen elements from the group G. The process is based on the product-replacement algorithm of [CLGM+95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G) + 1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

It should be noted that this process is not suitable for infinite groups, since all elements produced are products of the generators only and not their inverses. However, as long as the homomorphic image of G that is being worked with is finite, there is no problem.

`Random(P): Process -> GrpSLPElt`

Given a random element process P created by the function `RandomProcess(G)` for the SLP-group G, construct a random element of G by forming a random product over the expanded generating set stored as part of the process. The expanded generating set stored with the process is modified by replacing an existing generator by the element returned.

`Rep(G): GrpSLP -> GrpSLPElt`

A representative element of G.

Example: Speed of SLP-Group Homomorphisms (ex-827c15)

As an illustration of the efficiency of computing homomorphisms from SLP-groups, we set up the same random expression as both a straight-line program and a linear word.

```
> G := SLPGroup(3);
> F := FreeGroup(3);
> M := GeneralOrthogonalGroup(7, 3);
> gf := hom<G -> F | F.1, F.2, F.3>;
> gm := hom<G -> M | M.1, M.2, M.3>;
> fm := hom<F -> M | M.1, M.2, M.3>;
> P := RandomProcess(G);
> x := Random(P);
> #x;
85
```

The evaluation of the straight-line program will take 85 operations in M.

```
> w := gf(x);
> #w;
52307
```

The evaluation of the word will take 52306 multiplications in M.

```
> time h1 := gm(x);
Time: 0.020
> time h2 := fm(w);
Time: 1.640
> h1 eq h2;
true
```

85.7.3 Coercions Between Related Groups

`G ! g: GrpSLP, GrpSLPElt -> GrpSLPElt`

Given an element g belonging to an SLP-group H related to the group G, rewrite g as an element of G.

85.8 Bibliography

- [CLGM+95] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

Chapter 86

Finitely Presented Semigroups

86.1 Introduction

This Chapter presents the functions designed for computing with finitely-presented semigroups (fp-semigroups for short).

86.2 The Construction of Free Semigroups and their Elements

86.2.1 Structure Constructors

`FreeSemigroup(n): RngIntElt -> SgpFP`

Construct the free semigroup F on n generators, where n is a positive integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the **left hand side** of the assignment statement.

`FreeMonoid(n): RngIntElt -> MonFP`

Construct the free monoid F on n generators, where n is a positive integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the **left hand side** of the assignment statement.

Example: Free Semigroup (ex-70a198)

The statement

```
> F := FreeSemigroup(2);
```

creates the free semigroup on two generators. Here the generators may be referenced using the standard names, $F.1$ and $F.2$.

The statement

```
> F<x, y> := FreeSemigroup(2);
```

defines F to be the free semigroup on two generators and assigns the names x and y to the generators.

86.2.2 Element Constructors

Suppose S is an fp-semigroup, not necessarily free, for which generators have already been defined. A *word* is defined inductively as follows:

(i) A generator is a word;

(ii) The product uv of the words u and v is a word;

(iii) The power of a word, u^n, where u is a word and n is an integer, is a word.

An element (word) of S may be constructed as an expression in the generators as outlined below.

`S ! [i1, ... is]: SgpFP, [RngIntElt] -> SgpFPElt`

Given a semigroup S defined on r generators and a sequence $Q = [i_1, \cdots, i_s]$ of integers lying in the range $[1, r]$, construct the word $G.i_1G.i_2 \cdots G.i_s$.

`Id(M): MonFP -> MonFPElt`

`M ! 1: MonFP -> MonFPElt`

Construct the identity element (empty word) for the fp-monoid M.

86.3 Elementary Operators for Words

86.3.1 Multiplication and Exponentiation

The word operations defined here may be applied either to the words of a free semigroup or the words of a semigroup with non-trivial relations.

`u * v: SgpFPElt, SgpFPElt -> SgpFPElt`

Given words u and v belonging to the same fp-semigroup S, return the product of u and v.

`u ^ n: SgpFPElt, RngIntElt -> SgpFPElt`

The n-th power of the word u, where n is a positive integer.

`G ! Q: SgpFP, [ SgpFPElt ] -> SgpFPElt`

Given a sequence Q of words belonging to the fp-semigroup G, return the product $Q[1]Q[2]\cdots Q[n]$ of the terms of Q as a word in G.

86.3.2 The Length of a Word

`# u: SgpFPElt -> RngIntElt`

The length of the word u.

86.3.3 Equality and Comparison

The words of an fp-semigroup S are ordered first by length and then lexicographically. The lexicographic ordering is determined by the following ordering on the generators:

$$S.1 < S.2 < S.3 < S.4 < \cdots$$

Here, u and v are words belonging to some common fp-semigroup.

`u eq v: SgpFPElt, SgpFPElt -> BoolElt`

Returns `true` if the words u and v are identical (as elements of the appropriate free semigroup), `false` otherwise.

`u ne v: SgpFPElt, SgpFPElt -> BoolElt`

Returns `true` if the words u and v are not identical (as elements of the appropriate free semigroup), `false` otherwise.

`u lt v: SgpFPElt, SgpFPElt -> BoolElt`
Returns `true` if the word u precedes the word v, with respect to the ordering defined above for elements of an fp-semigroup, `false` otherwise.

`u le v: SgpFPElt, SgpFPElt -> BoolElt`
Returns `true` if the word u either precedes, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-semigroup, `false` otherwise.

`u ge v: SgpFPElt, SgpFPElt -> BoolElt`
Returns `true` if the word u either follows, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-semigroup, `false` otherwise.

`u gt v: SgpFPElt, SgpFPElt -> BoolElt`
Returns `true` if the word u follows the word v, with respect to the ordering defined above for elements of an fp-semigroup.

`IsOne(u): MonFPElt -> BoolElt`
Returns `true` if the word u, belonging to the monoid M, is the identity word, `false` otherwise.

86.4 Specification of a Presentation

86.4.1 Relations

`w1 = w2: SgpFPElt, SgpFPElt -> Rel`
Given words w_1 and w_2 over the generators of an fp-semigroup S, create the relation $w_1 = w_2$. Note that this relation is not automatically added to the existing set of defining relations R for S. It may be added to R, for example, through use of the `quo`-constructor (see below).

`LHS(r): Rel -> SgpFPElt`
Given a relation r over the generators of S, return the left hand side of the relation r. The object returned is a word over the generators of S.

`RHS(r): Rel -> SgpFPElt`
Given a relation r over the generators of S, return the right hand side of the relation r. The object returned is a word over the generators of S.

86.4.2 Presentations

A semigroup with non-trivial relations is constructed as a quotient of an existing semigroup, possibly a free semigroup.

`Semigroup< generators | relations >: SgpFPElt, ..., SgpFPElt, Rel, ...Rel -> SgpFP`
Given a clause consisting of a list of variables $x_1, \cdots, x_r$, and a set of relations over these generators, first construct the free semigroup F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the ideal of F defined by . The syntax for the clause is the same as for the `quo`-constructor. The function returns:

(a) The quotient semigroup S;

(b) The natural homomorphism $\phi : F \to S$.

Thus, the statement

```
S< y1, ..., yr > := Semigroup< x1, ..., xr | w1, ..., ws >;
```

is an abbreviation for

```
F< x1, ..., xr > := FreeSemigroup(r);

S< y1, ..., yr > := quo< F | w1, ..., ws >;
```

`Monoid< generators | relations >: MonFPElt, ..., MonFPElt, Rel, ..., Rel -> MonFP`

Given a clause consisting of a list of variables $x_1, \cdots, x_r$, and a set of relations over these generators, first construct the free monoid F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the ideal of F defined by . The syntax for the clause is the same as for the quo-constructor. The function returns:

(a) The quotient monoid M;

(b) The natural homomorphism $\phi : F \to M$.

Thus, the statement

```
M< y1, ..., yr > := Monoid< x1, ..., xr | w1, ..., ws >;
```

is an abbreviation for

```
F< x_1, ..., x_r > := FreeMonoid(r);

M< y_1, ..., y_r > := quo< F | w_1, ..., w_s >;
```

Example: Finitely Presented Monoid (ex-a36e1d)

We create the monoid defined by the presentation $< x, y \mid x^2, y^2, (xy)^2 >$.

```
> M<x,y> := Monoid< x, y | x^2, y^2, (x*y)^2 >;
> M;
Finitely presented monoid
Relations:
    x^2 = Id(M)
    y^2 = Id(M)
    (x * y)^2 = Id(M)
```

86.4.3 Accessing the Defining Generators and Relations

The functions in this group provide access to basic information stored for a finitely-presented semigroup G.

`S . i: SgpFP, RngIntElt -> SgpFPElt`

The i-th defining generator for S.

`Generators(S): SgpFP -> { SgpFPElt}`

A set containing the generators for S.

`NumberOfGenerators(S): SgpFP -> RngIntElt`

`Ngens(S): SgpFP -> RngIntElt`

The number of generators for S.

`Parent(u): SgpFPElt -> SgpFP`

The parent semigroup S of the word u.

`Relations(S): SgpFP -> [ Rel ]`

A sequence containing the defining relations for S.

86.5 Subsemigroups, Ideals, and Quotients

86.5.1 Subsemigroups and Ideals

`sub<S | L1, ..., Lr>: SgpFP, SgpFPElt, ..., SgpFPElt -> SgpFP`

Construct the subsemigroup R of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$.

A term L_i of the generator list may consist of any of the following objects:

(a) A word;

(b) A set or sequence of words;

(c) A sequence of integers representing a word;

(d) A set or sequence of sequences of integers representing words;

(e) A subsemigroup of an fp-semigroup;

(f) A set or sequence of subsemigroups.

The collection of words and semigroups specified by the list must all belong to the semigroup S, and R will be constructed as a subgroup of S. The generators of R consist of the words specified directly by terms L_i together with the stored generating words for any semigroups specified by terms of L_i. Repetitions of an element and occurrences of the identity element are removed (unless R is trivial).

`ideal<S | L1, ..., Lr>: SgpFP, SgpFPElt, ..., SgpFPElt -> SgpFPIdl`

Construct the two-sided ideal I of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$. The possible forms of a term L_i of the generator list are the same as for the `sub`-constructor.

`lideal<G | L1, ..., Lr>: SgpFP, SgpFPElt, ..., SgpFPElt -> SgpFPIdl`

Construct the left ideal I of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$. The possible forms of a term L_i of the generator list are the same as for the `sub`-constructor.

`rideal<G | L1, ..., Lr>: SgpFP, SgpFPElt, ..., SgpFPElt -> SgpFPIdl`

Construct the right ideal I of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$. The possible forms of a term L_i of the generator list are the same as for the `sub`-constructor.

86.5.2 Quotients

`quo< F | relations >: SgpFP, Rel, ..., Rel -> SgpFP`

Given an fp-semigroup F, and a list of relations over the generators of F, construct the quotient of F by the ideal of F defined by . The expression defining F may be either simply the name of a previously constructed semigroup, or an expression defining an fp-semigroup. Each term of the list must be a *relation*, a *relation list* or, if S is a monoid, a *word*. A *word* is interpreted as a relator if S is a monoid. A *relation* consists of a pair of words, separated by '='. (See above). A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=': $w_1 = w_2 = \cdots = w_r$. This is interpreted as the relations $w_1 = w_r, \ldots, w_{r-1} = w_r$.

Note that the relation list construct is only meaningful in the context of the `fp semigroup`-constructor. In the context of the `quo`-constructor, the identity element (empty word) of a monoid may be represented by the digit `1`.

Note that this function returns:

(a) The quotient semigroup S;

(b) The natural homomorphism $\phi : F \to S$.

86.6 Extensions

`DirectProduct(R, S): SgpFP, SgpFP -> SgpFP`

Given two fp-semigroups R and S, construct the direct product of R and S.

`FreeProduct(R, S): SgpFP, SgpFP -> SgpFP`

Given two fp-semigroups R and S, construct the free product of R and S.

86.7 Elementary Tietze Transformations

`AddRelation(S, r): SgpFP, Rel -> SgpFP`

`AddRelation(S, r, i): SgpFP, Rel, RngIntElt -> SgpFP`

Given an fp-semigroup S and a relation r in the generators of S, create the quotient semigroup obtained by adding the relation r to the defining relations of S. If an integer i is specified as third argument, insert the new relation after the i-th relation of S. If the third argument is omitted, r is added to the end of the relations that are carried across from S.

`DeleteRelation(S, r): SgpFP, Rel -> SgpFP`

Given an fp-semigroup S and a relation r that occurs among the given defining relations for S, create the semigroup T, having the same generating set as S but with the relation r removed.

`DeleteRelation(S, i): SgpFP, RngIntElt -> SgpFP`

Given an fp-semigroup S and an integer i, $1 \leq i \leq m$, where m is the number of defining relations for S, create the semigroup T having the same generating set as S but with the i-th relation omitted.

`ReplaceRelation(S, r1, r2): SgpFP, Rel, Rel -> SgpFP`

Given an fp-semigroup S and relations r_1 and r_2 in the generators of S, where r_1 is one of the given defining relations for S, create the semigroup T having the same generating set as S but with the relation r_1 replaced by the relation r_2.

`ReplaceRelation(S, i, r): SgpFP, RngIntElt, Rel -> SgpFP`

Given an fp-semigroup S, an integer i, $1 \leq i \leq m$, where m is the number of defining relations for S, and a relation r in the generators of S, create the semigroup T having the same generating set as S but with the i-th relation of S replaced by the relation r.

`AddGenerator(S): SgpFP -> SgpFP`

Given an fp-semigroup S with presentation $< X \mid R >$, create the semigroup T with presentation $< X \cup \{y\} \mid R >$, where y denotes a new generator.

`AddGenerator(S, w): SgpFP, SgpFPElt -> SgpFP`

Given an fp-semigroup S with presentation $< X \mid R >$ and a word w in the generators of S, create the semigroup T with presentation $< X \cup \{y\} \mid R \cup \{y = w\} >$, where y denotes a new generator.

`DeleteGenerator(S, y): SgpFP, SgpFPElt -> SgpFP`

Given an fp-semigroup S with presentation $< X \mid R >$ and a generator y of S such that either S has no relations involving y, or a single relation r containing a single occurrence of y, create the semigroup T with presentation $< X - \{y\} \mid R - \{r\} >$.

86.8 String Operations on Words

`Eliminate(u, x, v): SgpFPElt, SgpFPElt, SgpFPElt -> SgpFPElt`

Given words u and v, and a generator x, belonging to a semigroup S, return the word obtained from u by replacing each occurrence of x by v.

`Match(u, v, f): SgpFPElt, SgpFPElt, RngIntElt -> BoolElt, RngIntElt`

Suppose u and v are words belonging to the same semigroup S, and that f is an integer such that $1 \leq f \leq \#u$. If v is a subword of u, the function returns true, as well as the least integer l such that:

(a) $l \geq f$; and,

(b) v appears as a subword of u, starting at the l-th letter of u.

If no such l is found, `Match` returns only false.

`Random(S, m, n): SgpFP, RngIntElt, RngIntElt -> SgpFPElt`

A random word of length l in the generators of the semigroup S, where $m \leq l \leq n$.

`RotateWord(u, n): SgpFPElt, RngIntElt -> SgpFPElt`

The word obtained by cyclically permuting the word u by n places. If n is positive, the rotation is from left to right, while if n is negative the rotation is from right to left. In the case where n is zero, the function returns u.

`Substitute(u, f, n, v): SgpFPElt, RngIntElt, SgpFPElt, RngIntElt -> SgpFPElt`

Given words u and v belonging to a semigroup S, and non-negative integers f and n, this function replaces the substring of u of length n, starting at position f, by the word v. Thus, if $u = x_{i_1} \cdots x_{i_f} \cdots x_{i_{f+n-1}} \cdots x_{i_m}$ then the substring $x_{i_f} \cdots x_{i_{f+n-1}}$ is replaced by v. If u and v belong to a monoid M and the function is invoked with $v =$ `Id(M)`, then the substring $x_{i_f} \cdots x_{i_{f+n-1}}$ of u is deleted.

`Subword(u, f, n): SgpFPElt, RngIntElt, RngIntElt -> SgpFPElt`

The subword of the word u comprising the n consecutive letters commencing at the f-th letter of u.

`ElementToSequence(u): SgpFPElt -> [ SgpFPElt ]`

`Eltseq(u): SgpFPElt -> [ SgpFPElt ]`

The sequence obtained by decomposing u into the indices of its constituent generators. Thus, if $u = x_{i_1} \ldots x_{i_m}$, then the sequence constructed by `ElementToSequence` is $[i_1, i_2, \ldots, i_m]$.

Chapter 87

Monoids Given by Rewrite Systems

87.1 Introduction

The category of monoids defined by finite sets of rewrite rules provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to KBMAG's Knuth–Bendix completion procedure on monoids defined by a finite presentation. As such much of the documentation in this chapter is taken from the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure is assumed. Some familiarity with KBMAG would be beneficial.

87.1.1 Terminology

A rewrite monoid M is a finitely presented monoid in which equality between elements of M, called *words* or *strings*, is decidable via a sequence of rewriting equations, called *reduction relations, rules, or equations*. In the interests of efficiency the reduction rules are codified into a finite state automaton called a *reduction machine*. The words in a rewrite monoid M are ordered, as are the reduction relations of M. Several possible orderings of words are supported, namely short-lex, recursive, weighted short-lex, and wreath-product orderings. A rewrite monoid can be *confluent* or *non-confluent*. If a rewrite monoid M is confluent its reduction relations, or more specifically its reduction machine, can be used to reduce words in M to their irreducible normal forms under the given ordering, and so the word problem for M can be efficiently solved.

87.1.2 The Category of Rewrite Monoids

The family of all rewrite monoids forms a category. The objects are the rewrite monoids and the morphisms are monoid homomorphisms. The Magma designation for this category of monoids is `MonRWS`. Elements of a rewrite monoid are designated as `MonRWSElt`.

87.1.3 The Construction of a Rewrite Monoid

A rewrite monoid M is constructed in a three-step process:

(i) A free monoid F of the appropriate rank is defined.

(ii) A quotient Q of F is created.

(iii) The Knuth–Bendix completion procedure is applied to the monoid Q to produce a monoid M defined by a rewrite system.

The Knuth–Bendix procedure may or may not succeed. If it fails the user may need to perform the above steps several times, manually adjusting parameters that control the execution of the Knuth–Bendix procedure. If it succeeds then the rewrite systems constructed will be confluent.

87.2 Construction of a Rewrite Monoid

`RWSMonoid(Q: parameters): MonFP -> MonRWS`

```
MaxRelations   : RngIntElt                    Default: 32767
GeneratorOrder: SeqEnum                       Default:
Ordering       : MonStgElt                    Default: "ShortLex"
Levels         : SeqEnum                      Default:
Weights        : SeqEnum                      Default:
TidyInt        : RngIntElt                    Default: 100
RabinKarp      : Tup                          Default:
MaxStates      : RngIntElt                    Default:
MaxReduceLen   : RngIntElt                    Default: 32767
ConfNum        : RngIntElt                    Default: 500
MaxStoredLen   : Tup                          Default:
MaxOverlapLen  : RngIntElt                    Default:
Sort           : BoolElt                      Default: false
MaxOpLen       : RngIntElt                    Default: 0
```

The Knuth–Bendix completion procedure for monoids is run, with the relations of Q taken as the initial reduction rules for the procedure. Regardless of whether or not the completion procedure succeeds, the result will be a rewrite monoid, M, containing a reduction machine and a sequence of reduction relations. If the procedure succeeds M will be marked as confluent, and the word problem for M is therefore decidable. If, as is very likely, the procedure fails then M will be marked as non-confluent. In this case M will contain both the reduction relations and the reduction machine computed up to the point of failure.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent rewrite monoid. The optimal values for these limits varies from example to example.

Limit the maximum number of reduction equations to `MaxRelations`.

Give an ordering for the generators. This ordering affects the ordering of words in the alphabet. If not specified the ordering defaults to the order induced by Q's generators, that is $[g_1, \ldots, g_n]$ where $g_1, \ldots, g_n$ are the generators of Q.

`Ordering := "ShortLex"`: Use the short-lex ordering on strings. Shorter words come before longer, and for words of equal length lexicographical ordering is used, using the given ordering of the generators.

`Ordering := "Recursive" | "RTRecursive"`: Use a recursive ordering on strings. There are various ways to define this. Perhaps the quickest is as follows. Let u and v be strings in the generators. If one of u and v, say v, is empty, then $u \geq v$. Otherwise, let $u = u'a$ and $v = v'b$, where a and b are generators. Then $u > v$ if and only if one of the following holds:

(i) $a = b$ and $u' > v'$;

(ii) $a > b$ and $u > v'$;

(iii) $b > a$ and $u' > v$.

The `RTRecursive` ordering is similar to the `Recursive` ordering, but with $u = au'$ and $v = bv'$. Occasionally one or the other runs significantly quicker, but usually they perform similarly.

`Ordering := "WTShortLex"`: Use a weighted-lex ordering. `Weights` should be a sequence of non-negative integers, with the i-th element of `Weights` giving the weight of the i-th generator. The length of `Weights` must equal the number of generators. The length of words in the generators is then computed by adding up the weights of the generators in the words. Otherwise, ordering is as for short-lex.

`Ordering := "Wreath"`: Use a wreath-product ordering. `Levels` should be a sequence of non-negative integers, with the i-th element of `Levels` giving the level of the i-th generator. The length of `Levels` must equal the number of generators. In this ordering, two strings involving generators of the same level are ordered using short-lex, but all strings in generators of a higher level are larger than those involving generators of a lower level. That is not a complete definition; one can be found in [Sim94, pp. 46--50]. Note that the recursive ordering is the special case in which the level of generator number i is i.

After finding `TidyInt` new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.)

Use the Rabin-Karp algorithm for word-reduction on words having length at least l, provided that there are at least n equations, where `RabinKarp := <l, n>`. This uses less space than the default reduction automaton, but it is distinctly slower, so it should only be used when seriously short of memory. Indeed this option is only really useful for examples in which collapse occurs - i.e. at some intermediate stage of the calculation there is a very large set of equations, which later reduces to a much smaller confluent set. Collapse is not uncommon when analysing pathological presentations of finite groups, and this is one situation where the performance of the Knuth–Bendix algorithm can be superior to that of Todd-Coxeter coset enumeration. The best setting for `RabinKarp` varies from example to example - generally speaking, the smaller l is, the slower things will be, so set it as high as possible subject to not running out of memory. The number of equations n should be set higher than the expected final number of equations.

Limit the maximum number of states of the finite state automaton used for word reduction to `MaxStates`. By default there is no limit, and the space allocated is increased dynamically as required. The space needed for the reduction automaton can also be restricted by using the `RabinKarp` parameter. This limit is not usually needed.

Limit the maximum allowed length that a word can reach during reduction to `MaxReduceLen`. It is only likely to be exceeded when using the recursive ordering on words. This limit is usually not needed.

If `ConfNum` overlaps are processed and no new equations are discovered, then the overlap searching process is interrupted, and a fast check for confluence performed on the existing set of equations. Doing this too often wastes time, but doing it at the right moment can also save a lot of time. If `ConfNum` $= 0$, then the fast confluence check is performed only when the search for overlaps is complete.

Warning: Changing the default setting for any of the following parameters may either cause the procedure to terminate without having found a confluent presentation or change the monoid.

Only equations in which the left and right hand sides have lengths at most l and r, respectively, where `MaxStoredLen := <l, r>` are kept. Of course this may cause the overlap search to complete on a set of equations that is not confluent. In some examples, particularly those involving collapse (i.e. a large intermediate set of equations, which later simplifies to a small set), it can result in a confluent set being found much more quickly. It is most often useful when using a recursive ordering on words. Another danger with this option is that sometimes discarding equations can result in information being lost, with the effect of changing the monoid defined by the equations.

Only overlaps of total length at most `MaxOverlapLen` are processed. Of course this may cause the overlap search to complete on a set of equations that is not confluent.

If `Sort` is set to `true` then the equations will be sorted in order of increasing length of their left hand sides, rather than the default, which is to leave them in the order in which they were found. `MaxOpLen` should be a non-negative integer. If `MaxOpLen` is positive, then only equations with left hand sides having length at most `MaxOpLen` are output. If `MaxOpLen` is zero, then all equations are sorted by length. Of course, if `MaxOpLen` is positive, there is a danger that the monoid defined by the output equations may be different from the original.

SetVerbose("KBMAG", v): MonStgElt, RngIntElt

Set the verbose printing level for the Knuth-Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for v are 0 to 3 inclusive. Setting v to 0 corresponds to the '-silent' option of KBMAG in which no extra output is printed. Setting v to 2 corresponds to the '-v' (verbose) option of KBMAG in which a small amount of extra output is printed. Setting v to 3 corresponds to the '-vv' (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

Example: Rewrite Monoid for A4 (ex-1dd9f1)

Starting with a monoid presentation for the alternating group A_4, we construct a rewrite system. Since we don't specify an ordering the default ShortLex ordering is used. Since we don't specify a generator ordering, the default generator ordering, in this case that from Q, is used.

```
> FM<g10,g20,g30> := FreeMonoid(3);
> Q := quo< FM | g10^2=1, g20*g30=1, g30*g20=1,
>         g20*g20=g30, g30*g10*g30=g10*g20*g10>;
> M := RWSMonoid(Q);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ g10, g20, g30 ]
Ordering = ShortLex.
The reduction machine has 12 states.
The rewrite relations are:
    g10^2 = Id(FM)
    g20 * g30 = Id(FM)
    g30 * g20 = Id(FM)
    g20^2 = g30
    g30 * g10 * g30 = g10 * g20 * g10
    g30^2 = g20
    g20 * g10 * g20 = g10 * g30 * g10
    g30 * g10 * g20 * g10 = g20 * g10 * g30
    g10 * g20 * g10 * g30 = g30 * g10 * g20
    g20 * g10 * g30 * g10 = g30 * g10 * g20
    g10 * g30 * g10 * g20 = g20 * g10 * g30
```

Example: Neumann's Presentation of the Trivial Monoid (ex-917c6e)

We construct the second of Bernard Neumann's series of increasingly complicated presentations of the trivial monoid. The example runs best with a large value of TidyInt. Again the default ShortLex ordering is used.

```
> FM<x,X,y,Y,z,Z> := FreeMonoid(6);
> Q := quo< FM |
>         x*X=1, X*x=1, y*Y=1, Y*y=1, z*Z=1, Z*z=1,
>         y*y*X*Y*x*Y*z*y*Z*Z*X*y*x*Y*Y*z*z*Y*Z*y*z*z*Y*Z*y=1,
>         z*z*Y*Z*y*Z*x*z*X*X*Y*z*y*Z*Z*x*x*Z*X*z*x*x*Z*X*z=1,
>         x*x*Z*X*z*X*y*x*Y*Y*Z*x*z*X*X*y*y*X*Y*x*y*y*X*Y*x=1>;
> M := RWSMonoid(Q : TidyInt := 3000);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ x, X, y, Y, z, Z ]
Ordering = ShortLex.
The reduction machine has 1 state.
The rewrite relations are:
    Z = Id(FM)
    Y = Id(FM)
    z = Id(FM)
```

(continues on next page)

(continued from previous page)

```
    X = Id(FM)
    y = Id(FM)
    x = Id(FM)
```

Example: Confluent Submonoid of a Nilpotent Group (ex-fe6d08)

We construct a confluent presentation of a submonoid of a nilpotent group.

```
> FM<a,b,c> := FreeMonoid(6);
> Q := quo< FM | b*a=a*b*c, c*a=a*c, c*b=b*c >;
> M := RWSMonoid(Q:Ordering:="Recursive", GeneratorOrder:=[c,b,a]);
> M;
A confluent rewrite monoid.
Generator Ordering = [ c, b, a ]
Ordering = Recursive.
The reduction machine has 3 states.
    b * a = a * b * c
    c * a = a * c
    c * b = b * c
> Order(M);
Infinity
```

Example: Monoid Presentation of Fibonacci F(2,7) (ex-d43295)

We construct a monoid presentation corresponding to the Fibonacci group $F(2,7)$. This is a very difficult calculation unless the parameters of `RWSMonoid` are selected carefully. The best approach is to run the Knuth-Bendix once using a `Recursive` ordering with a limit on the lengths of equations stored (`MaxStoredLen := <15,15>` works well). This will halt returning a non-confluent rewrite monoid M, and give a warning message that the Knuth-Bendix procedure only partly succeeded. The original equations should then be appended to the relations of M, and the Knuth-Bendix re-run with no limits on lengths. It will then quickly complete with a confluent set. This is typical of a number of difficult examples, where good results can be obtained by running more than once.

```
> FM<a,b,c,d,e,f,g> := FreeMonoid(7);
> I := [a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b];
> Q := quo<FM | I>;
> M := RWSMonoid(Q: Ordering := "Recursive", MaxStoredLen := <15,15>);
Warning: Knuth Bendix only partly succeeded
> Q := quo< FM | Relations(M) cat I>;
> M := RWSMonoid(Q: Ordering := "Recursive");
> print M;
A confluent rewrite monoid.
Generator Ordering = [ a, b, c, d, e, f, g ]
Ordering = Recursive.
The reduction machine has 30 states.
The rewrite relations are:
    c = a^25
    d = a^20
    e = a^16
    f = a^7
    g = a^23
    b = a^24
    a^30 = a
> Order(M);
30
```

It turns out that the non-identity elements of this monoid form a submonoid isomorphic to the Fibonacci group $F(2,7)$ which is cyclic of order 29.

87.3 Basic Operations

87.3.1 Accessing Monoid Information

The functions in this section provide access to basic information stored for a rewrite monoid M.

`M . i: MonRWS, RngIntElt -> MonRWSElt`

The i-th defining generator for M.

`Generators(M): MonRWS -> [ MonRWSElt]`

A sequence containing the defining generators for M.

`NumberOfGenerators(M): MonRWS -> RngIntElt`

`Ngens(M): MonRWS -> RngIntElt`

The number of defining generators for M.

`Relations(M): MonRWS -> [MonFPRel]`

A sequence containing the defining relations for M. The relations will be given between elements of the free monoid of which M is a quotient. In these relations the (image of the) left hand side (in M) will always be greater than the (image of the) right hand side (in M) in the ordering on words used to construct M.

`NumberOfRelations(M): MonRWS -> RngIntElt`

`Nrels(M): MonRWS -> RngIntElt`

The number of relations in M.

`Ordering(M): MonRWS -> String`

The ordering of M.

`Parent(w): MonRWSElt -> MonRWS`

The parent monoid M for the word w.

Example: Accessing Rewrite Monoid Information (ex-de40ce)

We illustrate the access operations using the following presentation of S_4.

```
> FM<a,b> := FreeMonoid(2);
> Q := quo< FM | a^2=1, b^3=1, (a*b)^4=1 >;
> M<x,y> := RWSMonoid(Q);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ a, b ]
Ordering = ShortLex.
The reduction machine has 12 states.
    a^2 = Id(FM)
    b^3 = Id(FM)
    b * a * b * a * b = a * b^2 * a
    b^2 * a * b^2 = a * b * a * b * a
    b * a * b^2 * a * b * a = a * b * a * b^2 * a * b

> print Order(M);
24
> print M.1;
x
> print M.1*M.2;
x * y
> print Generators(M);
[ x, y ]
> print Ngens(M);
2
```

(continues on next page)

(continued from previous page)

```
> print Relations(M);
[ a^2 = Id(FM), b^3 = Id(FM), b * a * b * a * b = a * b^2 * a, b^2 * a * b^2 = a
* b * a * b * a, b * a * b^2 * a * b * a = a * b * a * b^2 * a * b ]
> print Nrels(M);
5
> print Ordering(M);
ShortLex
```

87.3.2 Properties of a Rewrite Monoid

`IsConfluent(M): MonRWS -> BoolElt`

Returns `true` if M is confluent, `false` otherwise.

`IsFinite(M): MonRWS -> BoolElt, RngIntElt`

Given a confluent monoid M return `true` if M has finite order and `false` otherwise. If M does have finite order also return the order of M.

`Order(M): MonRWS -> RngIntElt`

`# M: MonRWS -> RngIntElt`

Given a monoid M defined by a confluent presentation, this function returns the cardinality of M. If the order of M is known to be infinite ∞ is returned.

Example: Order of a Threefold Cover of A6 (ex-7a9289)

We construct a threefold cover of A_6.

```
> FM<a,b> := FreeMonoid(2);
> Q := quo< FM | a^3=1, b^3=1, (a*b)^4=1, (a*b^2)^5 = 1 >;
> M := RWSMonoid(Q);
> print Order(M);
1080
> IsConfluent(M);
true
```

Example: Order of a Free Abelian Group (ex-541f65)

We construct the 2-generator free abelian group and compute its order. The result `Infinity` indicates that the group has infinite order.

```
> FM<a,A,b,B> := FreeMonoid(4);
> Q := quo< FM | a*A=1, A*a=1, b*B=1, B*b=1, B*a*b=a>;
> M := RWSMonoid(Q);
> Order(M);
Infinity
```

Example: Finiteness of the Weyl Group E8 (ex-2a5004)

We construct the Weyl group E_8 and test whether or not it has finite order.

```
> FM<a,b,c,d,e,f,g,h> := FreeMonoid(8);
> Q := quo< FM | a^2=1, b^2=1, c^2=1, d^2=1, e^2=1, f^2=1, g^2=1,
>         h^2=1, b*a*b=a*b*a, c*a=a*c, d*a=a*d, e*a=a*e, f*a=a*f,
>         g*a=a*g, h*a=a*h, c*b*c=b*c*b, d*b=b*d, e*b=b*e, f*b=b*f,
>         g*b=b*g, h*b=b*h, d*c*d=c*d*c, e*c*e=c*e*c, f*c=c*f,
>         g*c=c*g, h*c=c*h, e*d=d*e, f*d=d*f, g*d=d*g, h*d=d*h,
>         f*e*f=e*f*e, g*e=e*g, h*e=e*h, g*f*g=f*g*f, h*f=f*h,
>         h*g*h=g*h*g>;
> M := RWSMonoid(Q);
> print IsFinite(M);
```

(continues on next page)

(continued from previous page)

```
true
> isf, ord := IsFinite(M);
> print isf, ord;
true 696729600
```

87.3.3 Construction of a Word

`Identity(M): MonRWS -> MonRWSElt`

`Id(M): MonRWS -> MonRWSElt`

`M ! 1: MonRWS, RngIntElt -> MonRWSElt`

Construct the identity word in M.

`M ! [ i1, ..., is ]: MonRWS, [ RngIntElt ] -> MonRWSElt`

Given a rewrite monoid M defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[1, r]$, construct the word $M.i_1 * M.i_2 * \cdots * M.i_s$.

Example: Constructing Words in a Rewrite Monoid (ex-00230f)

We construct the Fibonacci group $F(2, 7)$, and it's identity.

```
> FM<a,A,b,B,c,C,d,D,e,E,f,F,g,G> := FreeMonoid(14);
> Q := quo< FM | a*A=1, A*a=1, b*B=1, B*b=1, c*C=1, C*c=1,
>                d*D=1, D*d=1, e*E=1, E*e=1, f*F=1, F*f=1, g*G=1, G*g=1,
>                a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b>;
> M := RWSMonoid(Q : TidyInt := 1000);
> print Id(M);
Id(M)
> print M!1;
Id(M)
> Order(M);
29
```

87.3.4 Arithmetic with Words

Having constructed a rewrite monoid M one can perform arithmetic with words in M. Assuming we have $u, v \in M$ then the product $u * v$ will be computed as follows:

(i) The product $w = u * v$ is formed as a product in the appropriate free monoid.

(ii) The word w is reduced using the reduction machine associated with M.

If M is confluent, then w will be the unique minimal word that represents $u * v$ under the ordering of M. If M is not confluent, then there are some pairs of words which are equal in M, but which reduce to distinct words, and hence w will not be a unique normal form. Note that:

(i) Reduction of w can cause an increase in the length of w. At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.

(ii) The implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v: MonRWSElt, MonRWSElt -> MonRWSElt`

Product of the words w and v.

`u ^ n: MonRWSElt, RngIntElt -> MonRWSElt`

The n-th power of the word w, where n is a positive or zero integer.

`u eq v: MonRWSElt, MonRWSElt -> BoolElt`

Given words w and v belonging to the same monoid, return true if w and v reduce to the same normal form, false otherwise. If M is confluent this tests for equality. If M is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v: MonRWSElt, MonRWSElt -> BoolElt`

Given words w and v belonging to the same monoid, return false if w and v reduce to the same normal form, true otherwise. If M is confluent this tests for non-equality. If M is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w): MonRWSElt -> BoolElt`

`IsIdentity(w): MonRWSElt -> BoolElt`

Returns `true` if the word w is the identity word.

`# u: MonRWSElt -> RngIntElt`

The length of the word w.

`ElementToSequence(u): MonRWSElt -> [ RngIntElt ]`

`Eltseq(u): MonRWSElt -> [ RngIntElt ]`

The sequence Q obtained by decomposing the element u of a rewrite monoid into its constituent generators. Suppose u is a word in the rewrite monoid M. If $u = M.i_1 \cdots M.i_m$, then $Q[j] = i_j$, for $j = 1, \ldots, m$.

Example: Word Arithmetic in a Fibonacci Monoid (ex-3c8630)

We illustrate the word operations by applying them to elements of the Fibonacci monoid $FM(2, 5)$.

```
> FM<a,b,c,d,e> := FreeMonoid(5);
> Q:=quo< FM | a*b=c, b*c=d, c*d=e, d*e=a, e*a=b >;
> M<a,b,c,d,e> := RWSMonoid(Q);
> a*b*c*d;
b^2
> (c*d)^4 eq a;
true
> IsIdentity(a^0);
true
> IsIdentity(b^2*e);
false
```

87.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter Mappings. This section describes some special aspects of homomorphisms whose domain is a rewrite monoid.

87.4.1 General Remarks

Monoids in the category *MonRWS* currently are accepted as codomains only for monoid homomorphisms, whose codomain is a rewrite monoid as well.

87.4.2 Construction of Homomorphisms

`hom< M -> N | S >: Struct, Struct -> Map`

Returns the homomorphism from the rewrite group M to the monoid N defined by the expression S which must be the one of the following:

(i) A list, sequence, or indexed set containing the images of the n generators $M.1, \ldots, M.n$ of M. Here, the i-th element of S is interpreted as the image of $M.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set, or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i \text{ -> } y_i$, where x_i is a generator of M and $y_i \in N$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of M. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor. Presently, N must be either a rewrite monoid or a group, and it is not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of M.

87.5 Set Operations

`Random(M, n): MonRWS, RngIntElt -> MonRWSElt`

A random word of length at most n in the generators of M.

`Random(M): MonRWS -> MonRWSElt`

A random word (of length at most the order of M) in the generators of M.

`Representative(M): MonRWS -> MonRWSElt`

`Rep(M): MonRWS -> MonRWSElt`

An element chosen from M.

`Set(M, a, b): MonRWS, RngIntElt, RngIntElt -> SetEnum`

Search: MonStgElt	Default: "DFS"

Create the set of words, w, in M with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Set(M): MonRWS -> SetEnum`

Search: MonStgElt	Default: "DFS"

Create the set of words that is the carrier set of M. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Seq(M, a, b): MonRWS, RngIntElt, RngIntElt -> SeqEnum`

Search: MonStgElt	Default: "DFS"

Create the sequence S of words, w, in M with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

`Seq(M): MonRWS -> SeqEnum`

Search: MonStgElt	Default: "DFS"

Create a sequence S of words from the carrier set of M. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Example: Element Sets of a Rewrite Monoid (ex-e2ec60)

We construct the group D_{22}, together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FM<a,b,c,d,e,f> := FreeMonoid(6);
> Q := quo< FM | a^2=1, f^2=1,
>                d*a=a*c, e*b=b*f, d*c=c*e, d*f=a*d, a*e=e*b, b*f*c=f >;
> M<a,b,c,d,e,f> := RWSMonoid(Q);
> print Order(M);
22
> print Representative(M);
Id(M)
> print Random(M);
c * e
> print Random(M, 5);
d
> Set(M);
{ a * c, e, a * d, f, a * e, a * f, Id(M), a * c * e, b * a,
  b * d, a * d * b, b * e, c * e, a * b * a, d * b, a * b * d,
  a, a * b * e, b, c, a * b, d }
> Seq(M : Search := "BFS");
[ Id(M), a, b, c, d, e, f, a * b, a * c, a * d, a * e, a * f,
  b * a, b * d, b * e, c * e, d * b, a * b * a, a * b * d,
  a * b * e, a * c * e, a * d * b ]
```

87.6 Conversion to a Finitely Presented Monoid

There is a standard way to convert a rewrite monoid into a finitely presented monoid using the function *Relations*. This is shown in the following example.

Example: Converting a Rewrite Monoid to an FP-Monoid (ex-c939dc)

We construct the Fibonacci monoid $FM(2,4)$ as a rewrite monoid, and then convert it into a finitely presented monoid.

```
> FM<a,b,c,d> := FreeMonoid(4);
> Q := quo< FM | a*b=c, b*c=d, c*d=a, d*a=b >;
> M := RWSMonoid(Q);
> Order(M);
11
> P<w,x,y,z> := quo < FM | Relations(M) >;
> P;
Finitely presented monoid
Relations
    w * x = y
    x * y = z
    y * z = w
    z * w = x
    y^2 = w * z
    z^2 = x * w
    z * y = x^2
    y * x = w^2
```

(continues on next page)

(continued from previous page)

```
    x * w * z = x^2
    w^3 = w * z
    x * w^2 = x * z
    x^2 * z = x
    w^2 * y = w
    x^3 = x * w
    x^2 * w = z
    z * x = x * z
    y * w = w * y
    w^2 * z = y
    x * w * y = x
```

87.7 Bibliography

- [Hol97] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.
- [Sim94] Charles C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

References

[AAG99] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Math. Res. Lett.*, 6(3-4):287–291, 1999.

[AR84] D. G. Arrell and E. F. Robertson. A modified Todd-Coxeter algorithm. In *Computational group theory (Durham, 1982)*, pages 27–32. Academic Press, London, 1984.

[Art47] E. Artin. Theory of braids. *Ann. of Math. (2)*, 48:101–126, 1947.

[BKL98] Joan Birman, Ki Hyoung Ko, and Sang Jin Lee. A new approach to the word and conjugacy problems in the braid groups. *Adv. Math.*, 139(2):322–353, 1998.

[BJT97] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.

[CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.

[CLGM+95] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[CHN11] M. Conder, G. Havas, and M. Newman. On one-relator quotients of the modular group. In *Proc. Groups St Andrews 2009 in Bath*, number 387 in London Mathematical Society Lecture Note Series, 183–197. Cambridge University Press, 2011.

[Deh02] Patrick Dehornoy. Groupes de Garside. *Ann. Sci. École Norm. Sup. (4)*, 35(2):267–306, 2002.

[Eic01] Bettina Eick. On the Fitting subgroup of a polycyclic-by-finite group and its applications. *J. Algebra*, 242(1):176–187, 2001.

[ERM94] Elsayed A. El-Rifai and H. R. Morton. Algorithms for positive braids. *Quart. J. Math. Oxford Ser. (2)*, 45(180):479–497, 1994.

[ECH+92] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word processing in groups*. Jones and Bartlett Publishers, Boston, MA, 1992. ISBN 0-86720-244-0.

[Fab09] Anna Fabianska. *Algorithmic analysis of presentations of groups and modules*. Dissertation, RWTH Aachen University, 2009.

[FGM03] Nuno Franco and Juan González-Meneses. Conjugacy problem for braid groups and Garside groups. *J. Algebra*, 266(1):112–132, 2003.

[GKT+02] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, and U. Vishne. Length-based conjugacy search in the braid group. Preprint; available at http://www.arxiv.org/math.GR/0209267, 2002.

[Gar69] F. A. Garside. The braid group and other groups. *Quart. J. Math. Oxford Ser. (2)*, 20:235–254, 1969.

[GH] P. Gaudry, R. Harley. "Counting Points on Hyperelliptic Curves over Finite Fields." 313–332.

[Geb02] Volker Gebhardt. Efficient collection in infinite polycyclic groups. *J. Symbolic Comput.*, 34(3):213–228, 2002.

[Geb03] Volker Gebhardt. A new approach to the conjugacy problem in Garside groups. Preprint; available at http://www.arxiv.org/math.GT/0306199, 2003.

[GM02] Juan González-Meneses. Improving an algorithm to solve Multiple Simultaneous Conjugacy Problems in braid groups. Preprint; available at http://www.arxiv.org/math.GT/0212150, 2002.

[Hav91] G. Havas. Coset enumeration strategies. In *ISSAC'91*, 191–199. ACM Press, 1991.

[HH10] G. Havas and D.F. Holt. On Coxeter's families of group presentations. *J. Algebra*, 324(5):1076–1082, 2010.

[HKRR84] George Havas, P. E. Kenne, J. S. Richardson, and E. F. Robertson. A Tietze transformation program. In *Computational group theory (Durham, 1982)*, pages 69–73. Academic Press, London, 1984.

[HS03] D. Hofheinz and R. Steinwandt. A practical attack on some braid group based cryptographic primitives. In *Public Key Cryptography, 6th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2003*, volume 2567 of LNCS, pages 187–198. Springer, 2003.

[HEO05] D. F. Holt, B. Eick, and E. O`Brien. *Handbook of Computational Group Theory*. Chapman and Hall/CRC, 2005.

[Hol97] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.

[HRR17] Derek F. Holt, Sarah Rees, and Claas E. Röver. *Groups, languages and automata*. Volume 88 of London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 2017.

[Hug02] J. Hughes. A linear algebraic attack on the AAFG1 braid group cryptosystem. In *The 7th Australasian Conference on Information Security and Privacy, ACISP 2002*, volume 2384 of LNCS, pages 176–189. Springer, 2002.

[Jam12] Sebastian Jambor. *An L3-U3-quotient algorithm for finitely presented groups*. PhD Thesis, RWTH Aachen University, 2012.

[Jam15] Sebastian Jambor. An L_2-quotient algorithm for finitely presented groups on arbitrarily many generators. *J. Algebra*, 423:1109–1142, 2015. URL: https://doi.org/10.1016/j.jalgebra.2014.08.058, doi:10.1016/j.jalgebra.2014.08.058.

[KLC+00] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park. New public-key cryptosystem using braid groups. In *Advances in cryptology—CRYPTO 2000*, pages 166–183. Springer, Berlin, 2000.

[LP03] E. Lee and J. H. Park. Cryptanalysis of the public key encryption based on braid groups. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of LNCS, pages 477–490. Springer, 2003.

[LL02] Sang Jin Lee and Eonkyung Lee. Potential Weakness of the Commutator Key Agreement Protocol Based on Braid Groups. In *Advances in Cryptology—EuroCrypt 2002*, volume 2332 of LNCS, pages 14–28. Springer, 2002.

[Lo98] Eddie H. Lo. Finding intersections and normalizers in finitely generated nilpotent groups. *J. Symbolic Comput.*, 25(1):45–59, 1998.

[MKS76] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial group theory*. Dover Publications Inc., New York, revised edition, 1976. Presentations of groups in terms of generators and relations.

[Mar05] Joseph Marshall. Computational problems in hyperbolic groups. *Internat. J. Algebra Comput.*, 15(1):1–13, 2005. URL: https://doi.org/10.1142/S0218196705002086, doi:10.1142/S0218196705002086.

[NO96] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.

[Nic96] Werner Nickel. Computing nilpotent quotients of finitely presented groups. In *Geometric and computational perspectives on infinite groups (Minneapolis, MN and New Brunswick, NJ, 1994)*, pages 175–191. Amer. Math. Soc., Providence, RI, 1996.

[PF09] W. Plesken and A. Fabianska. An L2-quotient algorithm for finitely presented groups. *J. Algebra*, 322(3):914–935, 2009.

[Ram] Colin Ramsay. ACE. https://staff.itee.uq.edu.au/havas/.

[Sim94] Charles C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, Cambridge, 1994. ISBN 0-521-43213-8.

[Tes98a] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.

[Tes98b] E. Teske. “Better Random Walks for Pollard's Rho Method.” 1998.

Index

Symbols

(g$_1$, ..., g$_n$) (*literal*), 205
(u, v) (*literal*), 28, 280, 294
(u$_1$, ..., u$_r$) (*literal*), 280, 294
(u$_1$, ..., u$_n$) (*literal*), 28
<i, w> @ T (*literal*), 122
G (*operation*), 15, 215, 278, 291
M (*operation*), 324
P (*operation*), 151, 262
V (*operation*), 122
u (*operation*), 244, 280, 294, 306, 312, 326
w (*operation*), 27
- u (*operation*), 9
3-Quotient of a Group of Exponent 9 (*example*), 66

A

A ! 0 (*operation*), 8
A ! e (*operation*), 7
A ! g (*operation*), 7
A ! n (*operation*), 7
A ! [a$_1$, ... ,a$_n$] (*operation*), 7
A . i (*operation*), 3, 6
A / B (*operation*), 11
Abelian Group as an FP-Group (*example*), 42
Abelian Group from a Presentation (*example*), 2
Abelian Group from Cyclic Factors (*example*), 3
Abelian Quotient Invariants Modulo 2 (*example*), 157
Abelian Quotient Invariants of a Subgroup (*example*), 65
AbelianBasis(A) (*function*), 15
AbelianFPGroup([n$_1$,...,n$_r$]) (*function*), 89
AbelianGroup(A: parameters) (*function*), 6
AbelianGroup(G) (*function*), 11, 213
AbelianGroup(GrpAb, Q) (*function*), 11
AbelianGroup(GrpGPC, Q) (*function*), 212
AbelianGroup(Q) (*function*), 11
AbelianGroup([n$_1$,...,n$_r$]) (*function*), 3
AbelianGroup< X | R > (*constructor*), 2
AbelianInvariants(A) (*function*), 15
Abelianised Relation Matrix from Word Weights (*example*), 27
AbelianQuotient(G) (*function*), 11, 155, 225
AbelianQuotientInvariants(G) (*function*), 155, 225
AbelianQuotientInvariants(G, n) (*function*), 155
AbelianQuotientInvariants(G, T) (*function*), 155
AbelianQuotientInvariants(G, T, n) (*function*), 156
AbelianQuotientInvariants(H) (*function*), 155
AbelianQuotientInvariants(H, n) (*function*), 156
Access Functions for BKL Braids (*example*), 245
Access Functions for the Von Dyck Group (*example*), 290
Accessing a Coset Table (*example*), 123
Accessing Rewrite Group Information (*example*), 277
Accessing Rewrite Monoid Information (*example*), 323
Action(V) (*function*), 122
AddGenerator(G) (*function*), 82
AddGenerator(G, w) (*function*), 82
AddGenerator(S) (*function*), 316
AddGenerator(S, w) (*function*), 316
AddGroupRelations(Q, R) (*function*), 186
AddGroupRelations(Q, r) (*function*), 191
Adding Excluded Conjugates (*example*), 124
Adding Group Relations (*example*), 186
Adding Group Relations in Characteristic 3 (*example*), 187
Adding Ring Relations (*example*), 187
AddRedundantGenerators(G, Q) (*function*), 306
AddRelation(G, g) (*function*), 82
AddRelation(G, g, i) (*function*), 82
AddRelation(G, r) (*function*), 82
AddRelation(G, r, i) (*function*), 82
AddRelation(S, r) (*function*), 316
AddRelation(S, r, i) (*function*), 316
AddRelator(~P, w) (*procedure*), 106
AddRingRelations(Q, R) (*function*), 187
AddSubgroupGenerator(~P, w) (*procedure*), 106
Agemo(G, i) (*function*), 18
AllHomomorphisms(G, H) (*function*), 21
Alt(GrpFP, n) (*function*), 89
AlternatingFPGroup(n) (*function*), 89
AQInvariants(G) (*function*), 155, 225
AQInvariants(G, n) (*function*), 155
AQInvariants(G, T) (*function*), 155
AQInvariants(G, T, n) (*function*), 156
AQInvariants(H) (*function*), 155

AQInvariants(H, n) (*function*), 156
AQPrimes(G) (*function*), 157
AreIdentical(u, v: parameters) (*function*), 252
Arithmetic with Braids in Normal Form (*example*), 251
Automatic Group of the Trefoil Knot (*example*), 289
Automatic Structure of a Surface Group (*example*), 286
Automatic Structure of the Infinite Dihedral Group (*example*), 46
AutomaticGroup(F: parameters) (*function*), 286, 287
Automorphism Group of a Free Group (*example*), 33
Automorphism Group of a Free Subgroup (*example*), 34
AutomorphismGroup(F) (*function*), 33
AutomorphismGroup(G) (*function*), 22

B

B ! 1 (*operation*), 239
B ! p (*operation*), 240
B ! T (*operation*), 240
B ! [i_1, ..., i_k] (*operation*), 240
B ! [p_1, ...,p_k] (*operation*), 240
B ! [T_1, ..., T_k] (*operation*), 240
B . i (*operation*), 239
B . T (*operation*), 239
B @ f (*literal*), 266
BaseElement(P) (*function*), 261
Binary Tetrahedral Group (*example*), 39, 79
Boolean Predicates for Braids (*example*), 254
Bounding Definitions of Pcp Generators (*example*), 165
Braid Group onto PSL(2, 16) (*example*), 151
Braid Group onto Sym(5) (*example*), 148
Braid Group onto Symmetric Group (*example*), 266
BraidFPGroup(n) (*function*), 89
BraidGroup(GrpBrd, n: parameters) (*function*), 238
BraidGroup(n: parameters) (*function*), 238
Breaking Up a Relator (*example*), 302
Burau Representation of a Braid Group (*example*), 270
BurauRepresentation(B) (*function*), 269
BurauRepresentation(B, p) (*function*), 270

C

C * D (*operation*), 121
C * g (*operation*), 121
C1 eq C2 (*operation*), 121
C1 ne C2 (*operation*), 122
CanContinueEnumeration(P) (*function*), 108
CanonicalFactorRepresentation(u: parameters) (*function*), 244
CanonicalLength(u: parameters) (*function*), 245
CanRedoEnumeration(P) (*function*), 107
Cayley Graph of a Dihedral Group (*example*), 145
Cayley Graph of a Finite FP-Group (*example*), 58
CayleyGraph(G) (*function*), 144
Center(G) (*function*), 12, 222
Centraliser(F, H) (*function*), 32
Centraliser(F,x) (*function*), 32
Centraliser(G, a) (*function*), 12
Centraliser(G, g) (*function*), 219
Centraliser(G, H) (*function*), 219
Centralizer(F, H) (*function*), 32
Centralizer(F,x) (*function*), 32
Centralizer(G, a) (*function*), 12
Centralizer(G, g) (*function*), 219
Centralizer(G, H) (*function*), 219
Centre(G) (*function*), 12, 222
CFP(u: parameters) (*function*), 244
CharacterTable(G) (*function*), 21
ChiefSeries(G) (*function*), 18
Class 2 Nilpotent Quotient (*example*), 68, 168
Class 7 Quotient of R(2, 5) (*example*), 165
Codomain(f) (*function*), 148, 266
Coercion into a Subgroup (*example*), 210
Collect(P, Q) (*function*), 162
CollectRelations(~P) (*procedure*), 161
CommutatorSubgroup(G) (*function*), 12, 99
CommutatorSubgroup(G, H, K) (*function*), 12, 219
CommutatorSubgroup(H, K) (*function*), 12, 219
Complete(~P) (*procedure*), 262
CompositionSeries(G) (*function*), 18
Computing L2-Quotients (*example*), 181
Computing L3- and U3-Quotients (*example*), 191
Confluence of the Weyl Group E8 (*example*), 278
Confluent Submonoid of a Nilpotent Group (*example*), 322
Conjugacy Classes via Low Index Subgroups (*example*), 51, 131
Conjugacy in a Nilpotent Wreath Product (*example*), 225
Conjugate(H, g) (*function*), 219
Conjugate(H, u) (*function*), 137
Consistency(~P, k: parameters) (*procedure*), 161
Consistency(~P: parameters) (*procedure*), 161
Constructing Words in a Rewrite Group (*example*), 279
Constructing Words in a Rewrite Monoid (*example*), 325
Constructing Words in an Automatic Group (*example*), 293

ContinueEnumeration(~P: parameters) (*procedure*), 107
Converting a Polycyclic Group to an FP-Group (*example*), 93
Converting a Rewrite Group to an FP-Group (*example*), 283
Converting a Rewrite Monoid to an FP-Monoid (*example*), 328
Copying a Coset Enumeration Process (*example*), 112
Core(G, H) (*function*), 12, 137, 220
Coset Action of a Polycyclic Group (*example*), 218
Coset Table in the Infinite Dihedral Group (*example*), 117
Coset Table of an Index 15 Subgroup (*example*), 48
CosetAction(G, H) (*function*), 119, 218
CosetAction(V) (*function*), 119
CosetEnumerationProcess(G, H: parameters) (*function*), 103
CosetImage(G, H) (*function*), 119, 218
CosetImage(V) (*function*), 119
CosetKernel(G, H) (*function*), 119, 218
CosetKernel(V) (*function*), 119
CosetSatisfying(P : parameters) (*function*), 108
CosetSatisfying(T, S: parameters) (*function*), 125
CosetSatisfying(V, S: parameters) (*function*), 125
CosetSpace(G, H: parameters) (*function*), 121
CosetsSatisfying(P : parameters) (*function*), 108
CosetsSatisfying(T, S: parameters) (*function*), 125
CosetsSatisfying(V, S: parameters) (*function*), 125
CosetTable(G, H) (*function*), 217
CosetTable(G, H: parameters) (*function*), 117
CosetTable(P) (*function*), 109
CosetTableToPermutationGroup(G, T) (*function*), 117
CosetTableToRepresentation(G, T) (*function*), 117
Coxeter Group as an FP-Group (*example*), 94
Coxeter's Parametrised Families of Groups (*example*), 79
Coxeter's Parametrised Family of Groups (*example*), 40
CoxeterFPGroup(t) (*function*), 89
CoxeterFPGroup(W) (*function*), 94
Creating a Free Group (*example*), 26
Creating a Subgroup of an Abelian Group (*example*), 9
Creating an SLP-Group (*example*), 305
Creating and Representing Elements (*example*), 8
Creating Braid Group Elements (*example*), 241
Cycle(~u: parameters) (*procedure*), 250
Cycle(u: parameters) (*function*), 250
CyclicFPGroup(n) (*function*), 89
CyclicGroup(GrpGPC, n) (*function*), 212

D

Darstellungsgruppe(G) (*function*), 90
Decycle(~u: parameters) (*procedure*), 251
Decycle(u: parameters) (*function*), 250
DefinesHomomorphism(P) (*function*), 151
Defining S4 as a Quotient Group (*example*), 38
DeleteGenerator(G, x) (*function*), 82
DeleteGenerator(S, y) (*function*), 316
DeleteRelation(G, g) (*function*), 82
DeleteRelation(G, i) (*function*), 82
DeleteRelation(G, r) (*function*), 82
DeleteRelation(S, i) (*function*), 316
DeleteRelation(S, r) (*function*), 316
Depth(x) (*function*), 204
Derived Subgroup via Coset Enumeration (*example*), 124
DerivedGroup(G) (*function*), 12, 99, 223
DerivedLength(G) (*function*), 19, 222
DerivedSeries(G) (*function*), 19, 223
DerivedSubgroup(G) (*function*), 12, 99, 223
Dihedral Group in Different Categories (*example*), 208
DihedralFPGroup(n) (*function*), 89
DihedralGroup(GrpGPC, n) (*function*), 212
Direct and Free Products of FP-Groups (*example*), 43
Direct Product of FP-Groups (*example*), 91
Direct Product with the Infinite Dihedral Group (*example*), 213
DirectProduct(G, H) (*function*), 90, 212
DirectProduct(Q) (*function*), 90
DirectProduct(R, S) (*function*), 316
DirectSum(A, B) (*function*), 11
Discrete Logarithm for Quadratic Forms (*example*), 14
Display(P) (*procedure*), 162
Display(P, DisplayLevel) (*procedure*), 162
Domain(f) (*function*), 148, 266
Double Cosets in the Infinite Dihedral Group (*example*), 125
DoubleCoset(G, H, g, K) (*function*), 125
DoubleCosets(G, H, K) (*function*), 125
Dual(G) (*function*), 22

E

e @ f (*literal*), 266
EcheloniseWord(~P) (*function*), 162
EcheloniseWord(~P, ~r) (*function*), 162
EFAModuleMaps(G) (*function*), 226
EFAModules(G) (*function*), 226
EFASeries(G) (*function*), 223

Element Sets of a Rewrite Group (*example*), 282
Element Sets of a Rewrite Monoid (*example*), 328
ElementaryAbelianGroup(GrpGPC, p, n) (*function*), 212
ElementaryAbelianQuotient(G, p) (*function*), 15, 155, 225
ElementaryAbelianSeries(G) (*function*), 19
Elements(P) (*function*), 262
ElementToSequence(g) (*function*), 8
ElementToSequence(u) (*function*), 280, 294, 317, 326
ElementToSequence(u: parameters) (*function*), 244
ElementToSequence(w) (*function*), 27
ElementToSequence(x) (*function*), 204
Eliminate(~P: parameters) (*procedure*), 86
Eliminate(U, x, v) (*function*), 30, 80
Eliminate(u, x, v) (*function*), 30, 80, 316
EliminateGenerators(~P: parameters) (*procedure*), 86
EliminateRedundancy(~P) (*procedure*), 162
Eltseq(g) (*function*), 8
Eltseq(u) (*function*), 280, 294, 317, 326
Eltseq(u: parameters) (*function*), 244
Eltseq(w) (*function*), 27
Eltseq(x) (*function*), 204
Error for Large Relator Pieces (*example*), 301
Error for Short Relators (*example*), 300
Evaluate(u, G) (*function*), 307
Evaluate(u, Q) (*function*), 307
Evaluate(v, G) (*function*), 307
Evaluate(v, Q) (*function*), 307
Excluded Conjugates in a Coset Table (*example*), 114
ExcludedConjugate(P) (*function*), 109
ExcludedConjugates(P) (*function*), 109
ExcludedConjugates(T) (*function*), 122
ExcludedConjugates(V) (*function*), 122
ExistsCosetSatisfying(P : parameters) (*function*), 109
ExistsExcludedConjugate(P) (*function*), 110
ExistsNormalisingCoset(P) (*function*), 110
ExistsNormalizingCoset(P) (*function*), 110
ExplicitCoset(V, i) (*function*), 122
Exponent(G) (*function*), 15
ExponentLaw(~P : parameters) (*procedure*), 161
ExponentLaw(~P, Start, Fin: parameters) (*procedure*), 161
ExponentSum(w, x) (*function*), 27
Extending a 3-Quotient Interactively (*example*), 163
ExtractGenerators(P) (*function*), 133
ExtractGroup(P) (*function*), 133, 162
ExtraSpecialFPGroup(p, n : parameters) (*function*), 89
ExtraSpecialGroup(GrpGPC, p, n : parameters) (*function*), 212

F

f(B) (*literal*), 266
f(e) (*literal*), 266
f(H) (*literal*), 147
f(w) (*literal*), 147
FactoredIndex(G, H) (*function*), 19, 215
FactoredIndex(G, H: parameters) (*function*), 101
FactoredOrder(G) (*function*), 15, 215
FactoredOrder(G: parameters) (*function*), 95
FactoredOrder(P) (*function*), 163
Finding Homomorphisms onto Finite Groups (*example*), 150
Finite Quotients of the Modular Group (*example*), 183
Finitely Presented Monoid (*example*), 314
Finiteness of a Triple Cover of A6 (*example*), 292
Finiteness of the Fibonacci Group F(9) (*example*), 197
Finiteness of the Weyl Group E8 (*example*), 324
Fitting Subgroup via Semisimple Modules (*example*), 230
FittingGroup(G) (*function*), 12, 223
FittingLength(G) (*function*), 223
FittingSeries(G) (*function*), 223
FittingSubgroup(G) (*function*), 12, 223
FP-Group from a Permutation Group (*example*), 91
FP-Group Isomorphic to GL(3,4) (*example*), 44
FP-Group on Strong Generators (*example*), 92
FPGroup(A) (*function*), 12
FPGroup(G) (*function*), 91, 93, 214, 291
FPGroup< X | R > (*constructor*), 79
FPGroupStrong(G) (*function*), 92
FPGroupStrong(G, N) (*function*), 92
FrattiniSubgroup(G) (*function*), 18
Free Abelian Group (*example*), 1
Free Nilpotent Groups (*example*), 168
Free Nilpotent Groups via Nilpotent Quotient (*example*), 68
Free Semigroup (*example*), 311
FreeAbelianGroup(GrpGPC, n) (*function*), 212
FreeAbelianGroup(n) (*function*), 1
FreeAbelianQuotient(G) (*function*), 15, 225
FreeGenerators(H) (*function*), 32
FreeGroup(n) (*function*), 26
FreeMonoid(n) (*function*), 311
FreeNilpotentGroup(r, e) (*function*), 212
FreeProduct(G, H) (*function*), 90
FreeProduct(Q) (*function*), 90
FreeProduct(R, S) (*function*), 316
FreeSemigroup(n) (*function*), 311

FundamentalElement(B: parameters) (*function*), 239

G

G ! 1 (*operation*), 26, 204, 279, 293, 306
G ! g (*operation*), 17, 210, 310
G ! Q (*operation*), 204, 312
G ! [i_1, ..., i_s] (*operation*), 26, 279, 293
g * h (*operation*), 205
g *:= h (*operation*), 205
G . i (*operation*), 80, 215, 277, 290, 306
G / H (*operation*), 78
g / h (*operation*), 205
G / N (*operation*), 211
g /:= h (*operation*), 205
g @@ f (*literal*), 147
g ^ h (*operation*), 205
g ^ n (*operation*), 205
g ^:= h (*operation*), 205
g ^:= n (*operation*), 205
G eq H (*operation*), 16, 216
g eq h (*operation*), 205
g in C (*operation*), 121
g in G (*operation*), 16, 216, 308
G ne H (*operation*), 16, 216
g ne h (*operation*), 206
g notin C (*operation*), 121
g notin G (*operation*), 16, 216, 308
GCD(S: parameters) (*function*), 256
Gcd(S: parameters) (*function*), 256
GCD(u, v: parameters) (*function*), 255
Gcd(u, v: parameters) (*function*), 255
GeneratingWords(G, H) (*function*), 137
GeneratorNumber(w) (*function*), 27
GeneratorOrder(G) (*function*), 291
Generators(A) (*function*), 3, 6
Generators(B: parameters) (*function*), 239
Generators(G) (*function*), 80, 215, 277, 290, 306
Generators(H, G) (*function*), 215
Generators(M) (*function*), 323
Generators(S) (*function*), 314
GeneratorStructure(P) (*procedure*), 162
GeneratorStructure(P, Start, Fin) (*procedure*), 162
Generic Abelian Group of Units (*example*), 5
GenericAbelianGroup(U: parameters) (*function*), 4
GetElementPrintFormat(B) (*function*), 239
GetForceCFP(B) (*function*), 238
GetMatrices(Q) (*function*), 182, 191
GetPresentation(B) (*function*), 238
GModule(G, A) (*function*), 227
GModule(G, A, B) (*function*), 195, 227
GModule(G, A, B, p) (*function*), 195, 227
GModule(G, A, p) (*function*), 195, 227
GModulePrimes(G, A) (*function*), 194, 227
GModulePrimes(G, A, B) (*function*), 194, 227
GPCGroup(G) (*function*), 214
GreatestCommonDivisor(S: parameters) (*function*), 256
GreatestCommonDivisor(u, v: parameters) (*function*), 255
Group Generated by Three Involutions (*example*), 79
Group(P) (*function*), 87, 110
Group(V) (*function*), 122
Groups with Infinitely Many L2-Quotients (*example*), 74
Growth Function of a Dihedral Group (*example*), 297
GrowthFunction(G) (*function*), 297

H

H ! g (*operation*), 17, 210
H * g (*operation*), 121
H + K (*operation*), 17
H @ f (*literal*), 147
H @@ f (*literal*), 147
H ^ G (*operation*), 137, 219
H ^ g (*operation*), 219
H ^ u (*operation*), 137
H eq K (*operation*), 31, 141
H meet K (*operation*), 17, 32, 137, 219
H meet:= K (*operation*), 17, 219
H ne K (*operation*), 141
H notsubset G (*operation*), 16, 216
H notsubset K (*operation*), 141
H subset G (*operation*), 16, 216
H subset K (*operation*), 141
H2_G_QmodZ(G) (*function*), 22
HasClosedCosetTable(P) (*function*), 109
HasComplement(G, U) (*function*), 20
HasCompleteCosetTable(P) (*function*), 109
HasComputableAbelianQuotient(G) (*function*), 156
HasComputableLCS(G) (*function*), 223
HasFiniteAbelianQuotient(G) (*function*), 157
HasFiniteAQ(G) (*function*), 157
HasFiniteIndex(F, H) (*function*), 31
HasInfiniteComputableAbelianQuotient(G) (*function*), 156
HasInfinitePSL2Quotient(G) (*function*), 189
HasPositiveH1Dimension(G, phi : parameters) (*function*), 97
HaspQuotientDefinitions(G) (*function*), 160
HasValidCosetTable(P) (*function*), 109
HasValidIndex(P) (*function*), 110
HirschNumber(G) (*function*), 215
Hom Functor on Abelian Groups (*example*), 21
Hom(G, H) (*function*), 21
hom< A -> B | L> (*constructor*), 23
hom< A -> G | S > (*constructor*), 297
hom< B -> G | S : parameters > (*constructor*), 266
hom< G -> H | L: parameters> (*constructor*), 307

hom< M -> N | S > (*constructor*), 326
hom< P -> G | S : parameters> (*constructor*), 211
hom< P -> G | S > (*constructor*), 147
hom< R -> G | S > (*constructor*), 283
HomGenerators(G, H) (*function*), 21
Homomorphism of D4 onto Z2 x Z2 (*example*), 212
Homomorphism onto A5 (*example*), 61
Homomorphism(A, B, X, Y) (*function*), 23
Homomorphism(P) (*function*), 151
Homomorphisms of Generic Abelian Groups (*example*), 24
Homomorphisms with Redundant Generators (*example*), 307
Homomorphisms(F, G : parameters) (*function*), 149, 150
Homomorphisms(F, G, A : parameters) (*function*), 149, 150
Homomorphisms(G, H) (*function*), 21
HomomorphismsProcess(F, G : parameters) (*function*), 150
HomomorphismsProcess(F, G, A : parameters) (*function*), 150
Hyperbolicity from Group Relators (*example*), 300
Hyperbolicity of a One-Relator Group (*example*), 300
Hyperbolicity with Several Red Relators (*example*), 303
Hypercenter(G) (*function*), 12
Hypercentre(G) (*function*), 12

I

Id(A) (*function*), 8
Id(B) (*function*), 239
Id(G) (*function*), 26, 204, 279, 293, 306
Id(M) (*function*), 312, 325
ideal<S | L_1, ..., L_r> (*constructor*), 315
Identifying PSL(3, 7):2 from a Presentation (*example*), 199
Identity(A) (*function*), 8
Identity(B) (*function*), 239
Identity(G) (*function*), 26, 204, 279, 293, 306
Identity(M) (*function*), 325
Image(f) (*function*), 148, 266
Implicit Coset Enumeration (*example*), 116
InclusionMap(G, H) (*function*), 210
Index(F, H) (*function*), 31
Index(G, H) (*function*), 19, 215
Index(G, H: parameters) (*function*), 101
Index(P) (*function*), 110
IndexedCoset(V, C) (*function*), 122
IndexedCoset(V, w) (*function*), 122
Induced Homomorphism for Co1 (*example*), 119
InducedPermutation(u) (*function*), 244
Infimum(u: parameters) (*function*), 245
Infinite and Finite L2-Quotients (*example*), 190
Infinitely Many L3-Quotients (*example*), 192
Interactive Braid Conjugacy Testing (*example*), 262
Interactive Coset Table for Harada-Norton (*example*), 111
Interactive Simplification of the Fibonacci Group F(7) (*example*), 87
Interactively Extending a Class 6 Quotient (*example*), 166
Intersections of Subgroups of a Nilpotent Group (*example*), 221
Inverse(~u) (*procedure*), 250
Inverse(u) (*function*), 250
Inverse(w) (*function*), 280, 294
InverseAutomorphismFreeGroup(F, Q) (*function*), 33
IsAbelian(G) (*function*), 220
IsAutomaticGroup(F: parameters) (*function*), 286, 287
IsCentral(G, H) (*function*), 220
IsComplete(V) (*function*), 122
IsConfluent(G) (*function*), 278
IsConfluent(M) (*function*), 324
IsConjugate(F, H, K) (*function*), 32
IsConjugate(F, x, y) (*function*), 32
IsConjugate(G, g, h) (*function*), 225
IsConjugate(G, H, K) (*function*), 141, 220, 225
IsConjugate(G, x, y) (*function*), 299
IsConjugate(u, v: parameters) (*function*), 253
IsConsistent(G) (*function*), 209
IsCyclic(G) (*function*), 19, 220
IsElementaryAbelian(G) (*function*), 19, 220
IsEmpty(P) (*function*), 133, 151, 262
IsEmptySimpleQuotientProcess(P) (*function*), 180
IsEmptyWord(u: parameters) (*function*), 252
IsFinite(G) (*function*), 16, 220, 278, 291
IsFinite(M) (*function*), 324
IsFinite(x) (*function*), 205
IsFree(G) (*function*), 19
IsGE(u, v: parameters) (*function*), 253
IsGe(u, v: parameters) (*function*), 253
IsHyperbolic(F, red, green) (*function*), 299
IsHyperbolic(G) (*function*), 298
IsId(g) (*function*), 206
IsId(u) (*function*), 14
IsId(u: parameters) (*function*), 253
IsId(w) (*function*), 280, 294, 326
IsIdenticalPresentation(G, H) (*function*), 209
IsIdentity(g) (*function*), 206
IsIdentity(u) (*function*), 14
IsIdentity(u: parameters) (*function*), 253
IsIdentity(w) (*function*), 280, 294, 326
IsIdentity(w, D1) (*function*), 304
IsInfinite(G) (*function*), 16

IsInfiniteFPGroup(G : parameters) (*function*), 96
IsIsomorphic(G, H) (*function*), 22
IsLarge(G, L, U:parameters) (*function*), 98
IsLE(u, v: parameters) (*function*), 253
IsLe(u, v: parameters) (*function*), 253
IsMaximal(G, H) (*function*), 19, 142
IsMixed(G) (*function*), 19
IsNeat(G, H) (*function*), 19
IsNilpotent(G) (*function*), 220
IsNormal(G, H) (*function*), 141, 220
iso< A -> B | L> (*constructor*), 23
Isomorphism Search for Hillman's Groups (*example*), 62, 154
Isomorphism Search for Neumann's Groups (*example*), 63, 154
Isomorphism(A, B, X, Y) (*function*), 23
IsOne(u) (*function*), 313
IsPerfect(G) (*function*), 156, 220
IspGroup(G) (*function*), 19
IsProductOfParallelDescendingCycles(p) (*function*), 240
IsPure(G, H) (*function*), 19
IsSatisfied(U, E) (*function*), 147
IsSelfNormalising(G, H) (*function*), 221
IsSelfNormalizing(G, H) (*function*), 142, 221
IsSimple(G) (*function*), 220
IsSimple(u: parameters) (*function*), 252
IsSoluble(G) (*function*), 220
IsSolvable(G) (*function*), 220
IsSubgroup(H, K) (*function*), 31
IsSubgroup(H,G) (*function*), 16, 216
IsSubgroup(H,K) (*function*), 141
IsSuperSummitRepresentative(u: parameters) (*function*), 253
IsUltraSummitRepresentative(u: parameters) (*function*), 253
IsValid(P) (*function*), 133, 151

J

Jacobi(~P, c, b, a) (*function*), 162
Jacobi(~P, c, b, a, ~r) (*function*), 162

K

K ! g (*operation*), 17, 210
Kernel(f) (*function*), 148
KG-Modules of a Finitely Presented Group (*example*), 195

L

L2-Quotients from a Coxeter Matrix (*example*), 185
L2-Quotients of Coxeter Presentations (*example*), 184
L2-Quotients of FP-Groups (*example*), 73
L2-Quotients with Exact Orders (*example*), 185
L2Quotients(G) (*function*), 181, 185
L2Quotients(M) (*function*), 185
L3Quotients(G: parameters) (*function*), 191
Largest 2-Generator Group of Exponent 5 (*example*), 159
Largest 2-Quotient of Class 6 (*example*), 66, 158
Largest 3-Quotient of Class 6 (*example*), 159
Lattice Operations on Braids (*example*), 257
LCM(S: parameters) (*function*), 257
Lcm(S: parameters) (*function*), 257
LCM(u, v: parameters) (*function*), 256
Lcm(u, v: parameters) (*function*), 256
LeadingExponent(x) (*function*), 204
LeadingGenerator(w) (*function*), 27
LeadingGenerator(x) (*function*), 204
LeadingTerm(x) (*function*), 204
LeastCommonMultiple(S: parameters) (*function*), 257
LeastCommonMultiple(u, v: parameters) (*function*), 256
LeftConjugate(~u, v) (*function*), 250
LeftConjugate(u, v) (*function*), 250
LeftCosetSpace(G, H: parameters) (*function*), 121
LeftDiv(u, ~v) (*function*), 250
LeftDiv(u, v) (*function*), 250
LeftGCD(S: parameters) (*function*), 256
LeftGcd(S: parameters) (*function*), 256
LeftGCD(u, v: parameters) (*function*), 255
LeftGcd(u, v: parameters) (*function*), 255
LeftGreatestCommonDivisor(S: parameters) (*function*), 256
LeftGreatestCommonDivisor(u, v: parameters) (*function*), 255
LeftLCM(S: parameters) (*function*), 257
LeftLcm(S: parameters) (*function*), 257
LeftLCM(u, v: parameters) (*function*), 256
LeftLcm(u, v: parameters) (*function*), 256
LeftLeastCommonMultiple(S: parameters) (*function*), 257
LeftLeastCommonMultiple(u, v: parameters) (*function*), 256
LeftMixedCanonicalForm(u: parameters) (*function*), 247
LeftNormalForm(~u: parameters) (*procedure*), 247
LeftNormalForm(u: parameters) (*function*), 247
Length, Weight, and Leading Generator of Words (*example*), 29
LHS(r) (*function*), 2, 313
lideal<G | L_1, ..., L_r> (*constructor*), 315
Log(g, d: parameters) (*function*), 14
Low Index Subgroups of a Triangle Group (*example*), 133
Lower Central Series of a Dihedral Group (*example*), 224
LowerCentralSeries(G) (*function*), 19, 223
LowIndexNormalSubgroups(G, n: parameters)

(*function*), 136
LowIndexProcess(G, R : parameters) (*function*), 133
LowIndexProcess(G, R: parameters) (*function*), 133
LowIndexSubgroups(G, R : parameters) (*function*), 130
LowIndexSubgroups(G, R: parameters) (*function*), 130

M

M ! 1 (*operation*), 312, 325
M ! [i_1, ..., i_s] (*operation*), 325
m * u (*operation*), 9
M . i (*operation*), 323
Match(u, v, f) (*function*), 30, 80, 316
Matrix Groups of an L2-Quotient (*example*), 74
Matrix Groups of L2-Quotients (*example*), 182
Maximal Central Extension of a Group (*example*), 90
Maximal Nilpotent Quotient (*example*), 171
Maximal Subgroups of the Hall-Janko Group (*example*), 143
MaximalNumberOfCosets(P) (*function*), 110
MaximalOvergroup(G, H) (*function*), 137
MaximalSubgroups(G) (*function*), 20
Metabelian 5-Quotient (*example*), 67, 159
MinimalElementConjugatingToPositive(x, s: parameters) (*function*), 264
MinimalElementConjugatingToSuperSummit(x, s: parameters) (*function*), 264
MinimalElementConjugatingToUltraSummit(x, s: parameters) (*function*), 264
MinimalOvergroup(G, H) (*function*), 137
MixedCanonicalForm(u: parameters) (*function*), 247
Modifying Limits for a Knot Group (*example*), 288
Monoid Presentation of Fibonacci F(2,7) (*example*), 322
Monoid< generators | relations > (*constructor*), 313
Morphism(H, G) (*function*), 17

N

n * G (*operation*), 18
ncl< G | H > (*constructor*), 219
ncl< G | L > (*constructor*), 99
ncl<G | f> (*constructor*), 99
ncl<G | L> (*constructor*), 209
Neumann's Presentation of the Trivial Monoid (*example*), 321
NextClass(~P : parameters) (*procedure*), 160
NextClass(~P, k : parameters) (*procedure*), 160
NextElement(~P) (*procedure*), 151, 262
NextSimpleQuotient(~P) (*procedure*), 180
NextSubgroup(~P) (*procedure*), 133
NextSubgroup(~P, ~G) (*procedure*), 133
Ngens(A) (*function*), 3, 6
Ngens(B) (*function*), 239
Ngens(G) (*function*), 80, 215, 277, 290, 306
Ngens(M) (*function*), 323
Ngens(P) (*function*), 87
Ngens(S) (*function*), 314
NilpotencyClass(G) (*function*), 223
Nilpotent Quotient of a Baumslag-Solitar Group (*example*), 68
Nilpotent Quotients of Baumslag-Solitar Groups (*example*), 169
NilpotentPresentation(G) (*function*), 223
NilpotentQuotient(G, c: parameters) (*function*), 168
Normal Forms of Braids (*example*), 248
NormalClosure(G, H) (*function*), 137, 219
NormalForm(~u: parameters) (*procedure*), 247
NormalForm(u: parameters) (*function*), 247
Normaliser Element from a Partial Coset Table (*example*), 114
Normaliser(F, H) (*function*), 32
Normaliser(G, H) (*function*), 137, 220
Normalizer(F, H) (*function*), 32
Normalizer(G, H) (*function*), 137, 220
NPCGenerators(G) (*function*), 215
NPCgens(G) (*function*), 215
Nrels(G) (*function*), 277
Nrels(M) (*function*), 323
Nrels(P) (*function*), 87
NuclearRank(G) (*function*), 163
NuclearRank(P) (*function*), 163
NumberingMap(G) (*function*), 16
NumberOfGenerators(A) (*function*), 3, 6
NumberOfGenerators(B) (*function*), 239
NumberOfGenerators(G) (*function*), 80, 215, 277, 290, 306
NumberOfGenerators(M) (*function*), 323
NumberOfGenerators(P) (*function*), 87
NumberOfGenerators(S) (*function*), 314
NumberOfPCGenerators(G) (*function*), 215
NumberOfPCGenerators(P) (*function*), 163
NumberOfRelations(G) (*function*), 277
NumberOfRelations(M) (*function*), 323
NumberOfRelations(P) (*function*), 87
NumberOfStrings(B) (*function*), 239
NumberOfSubgroupsAbelianPGroup(A) (*function*), 20

O

Omega(G, i) (*function*), 18
Operations on a Subgroup of Index 448 (*example*), 52
Operations on Words in a Free Group (*example*), 81
Order of a Finite FP-Group (*example*), 58

Order of a Finitely Presented Group (*example*), 95
Order of a Free Abelian Group (*example*), 324
Order of a Large Soluble Quotient (*example*), 70
Order of a Soluble Quotient (*example*), 173
Order of a Threefold Cover of A6 (*example*), 324
Order of an Infinite Automatic Group (*example*), 291
Order of an Infinite Rewrite Group (*example*), 279
Order(G) (*function*), 15, 215, 278, 291
Order(g, l, u, n, m: parameters) (*function*), 13
Order(g, l, u: parameters) (*function*), 13
Order(G: parameters) (*function*), 95
Order(g: parameters) (*function*), 13
Order(M) (*function*), 324
Order(P) (*function*), 163
Order(x) (*function*), 12, 205
Ordering(G) (*function*), 277
Ordering(M) (*function*), 323
Orders of a Parametrised Family of Groups (*example*), 102
Orders of Abelian Group Elements (*example*), 12

P

Parent(G) (*function*), 231
Parent(r) (*function*), 2
Parent(u) (*function*), 3, 244, 306, 314
Parent(w) (*function*), 27, 279, 293, 323
Parent(x) (*function*), 205
PCExponents(G) (*function*), 215
PCGenerators(G) (*function*), 215
PCGenerators(H, G) (*function*), 215
PCGroup(A) (*function*), 11
PCGroup(G) (*function*), 144, 214
pClass(P) (*function*), 163
pCoveringGroup(~P) (*procedure*), 162
pCoveringGroup(G) (*function*), 162
Permutation Representation of G2(3) (*example*), 49, 120
PermutationGroup(A) (*function*), 12
PermutationGroup(G) (*function*), 144
pMultiplicatorRank(G) (*function*), 163
pMultiplicatorRank(P) (*function*), 163
PolycyclicGroup< x_1, ..., x_n | R : parameters > (*constructor*), 206
PositiveConjugates(u: parameters) (*function*), 259
PositiveConjugatesProcess(u: parameters) (*function*), 261
PowerGroup(G) (*function*), 231
pPrimaryComponent(A, p) (*function*), 15
pPrimaryInvariants(A, p) (*function*), 15
pQuotient(F, p, c: parameters) (*function*), 158
pQuotientDefinitions(G) (*function*), 160
pQuotientProcess(F, p, c: parameters) (*function*), 160
Presentation for an Index 3 Subgroup (*example*), 128
Presentation of an Index 3 Subgroup (*example*), 56
Presentation on New Generators (*example*), 129
PresentationIsSmall(G) (*function*), 209
PresentationLength(G) (*function*), 80
PresentationLength(P) (*function*), 87
PrimaryAbelianBasis(A) (*function*), 15
PrimaryAbelianInvariants(A) (*function*), 15
Primes of Semisimple G-Modules (*example*), 229
Proving a Coxeter Group Infinite (*example*), 45
Proving a Group Infinite (*example*), 96
Proving a Group Is Infinite (*example*), 135
Proving Infiniteness Interactively (*example*), 152
PSL(2,8) as a Quotient of a Triangle Group (*example*), 135
Pullback(f, N) (*function*), 195
Pullback(x, s: parameters) (*function*), 264

Q

quo< F | R > (*constructor*), 77
quo< F | relations > (*constructor*), 315
quo< GrpGPC : F | R : parameters > (*constructor*), 206
quo<F | R> (*constructor*), 11
quo<G | L> (*constructor*), 211
Quotient Constructor Without a Free Group (*example*), 78
Quotient Constructor without a Free Group (*example*), 39
Quotient of a Triangle Group (*example*), 78
Quotient of an FP-Group (*example*), 40

R

Random(A) (*function*), 7
Random(B, m, n: parameters) (*function*), 241
Random(B, r, s, m, n: parameters) (*function*), 240
Random(B: parameters) (*function*), 240
Random(G) (*function*), 17, 217, 281, 295
Random(G, m, n) (*function*), 27
Random(G, max) (*function*), 217
Random(G, n) (*function*), 17, 281, 295
Random(M) (*function*), 327
Random(M, n) (*function*), 327
Random(P) (*function*), 17, 216, 309
Random(S, m, n) (*function*), 317

RandomCFP(B, r, s, m, n: parameters) (*function*), 240
RandomCFP(B: parameters) (*function*), 240
RandomProcess(G) (*function*), 16, 216, 309
RandomWord(B, m, n: parameters) (*function*), 241
RandomWord(B: parameters) (*function*), 241
Recursive Ordering for a Non-Hopfian Group (*example*), 274
Red Relators for a Free Product (*example*), 302
RedoEnumeration(~P: parameters) (*procedure*), 107
RedRelatorsForFreeProduct(groupList, freeFacs) (*function*), 302
ReduceGenerators(G) (*function*), 84
Reducing a Generating Set (*example*), 88
Refining an EFA-Series with Module Maps (*example*), 230
RelationMatrix(A) (*function*), 3
Relations in a Free Abelian Group (*example*), 2
Relations(A) (*function*), 3
Relations(G) (*function*), 80, 277
Relations(M) (*function*), 323
Relations(S) (*function*), 314
Relevant Primes for Module Construction (*example*), 196
Rep(B) (*function*), 239
Rep(G) (*function*), 17, 216, 281, 295, 309
Rep(M) (*function*), 327
Rep(P) (*function*), 262
ReplaceRelation(G, h, g) (*function*), 82
ReplaceRelation(G, h, r) (*function*), 82
ReplaceRelation(G, i, g) (*function*), 82
ReplaceRelation(G, i, r) (*function*), 82
ReplaceRelation(G, s, g) (*function*), 82
ReplaceRelation(G, s, r) (*function*), 82
ReplaceRelation(S, i, r) (*function*), 316
ReplaceRelation(S, r_1, r_2) (*function*), 316
Replacing a Relation (*example*), 83
Representation(g) (*function*), 8
Representation(S, g) (*function*), 8
Representative(B) (*function*), 239
Representative(G) (*function*), 17, 216, 281, 295
Representative(M) (*function*), 327
Representative(P) (*function*), 262
Res_H2_G_QmodZ(U, H2) (*function*), 22
ResumeEnumeration(~P: parameters) (*procedure*), 108
RevertClass(~P) (*procedure*), 162
Rewrite Group for the Von Dyck (2,3,5) Group (*example*), 273
Rewrite Monoid for A4 (*example*), 321
Rewrite System for a Free Nilpotent Group (*example*), 275
Rewrite(G, ~H : parameters) (*procedure*), 127
Rewrite(G, H : parameters) (*function*), 127
Rewrite(H, x) (*function*), 32
Rewriting a Presentation of L2(7) (*example*), 57
RHS(r) (*function*), 2, 313
rideal<G | L_1, ..., L_r> (*constructor*), 315
Right Transversal in the Infinite Dihedral Group (*example*), 217
RightCosetSpace(G, H: parameters) (*function*), 121
RightGCD(S: parameters) (*function*), 256
RightGcd(S: parameters) (*function*), 256
RightGCD(u, v: parameters) (*function*), 256
RightGcd(u, v: parameters) (*function*), 256
RightGreatestCommonDivisor(S: parameters) (*function*), 256
RightGreatestCommonDivisor(u, v: parameters) (*function*), 256
RightLCM(S: parameters) (*function*), 257
RightLcm(S: parameters) (*function*), 257
RightLCM(u, v: parameters) (*function*), 256
RightLcm(u, v: parameters) (*function*), 256
RightLeastCommonMultiple(S: parameters) (*function*), 257
RightLeastCommonMultiple(u, v: parameters) (*function*), 256
RightMixedCanonicalForm(u: parameters) (*function*), 247
RightNormalForm(~u: parameters) (*procedure*), 247
RightNormalForm(u: parameters) (*function*), 247
RightTransversal(G, H) (*function*), 17, 122, 217
RotateWord(u, n) (*function*), 30, 80, 317
RWSGroup(F: parameters) (*function*), 273, 274
RWSMonoid(F: parameters) (*function*), 275
RWSMonoid(Q: parameters) (*function*), 319
r[1] (*indexed*), 2
r[2] (*indexed*), 2

S

S ! [i_1, ... i_s] (*operation*), 312
S . i (*operation*), 314
S notsubset G (*operation*), 16, 216, 308
S subset G (*operation*), 16, 216, 308
Schreier Generators (*example*), 140
SchreierGenerators(G, H : parameters) (*function*), 137
SchreierGraph(G, H) (*function*), 145
SchreierSystem(G, H) (*function*), 138
Search(~P: parameters) (*procedure*), 86
SearchEqual(~P: parameters) (*procedure*), 87
SearchForIsomorphism(F, G, m : parameters) (*function*), 153
Selecting Cosets by a Condition (*example*), 126
Semigroup< generators | relations > (*constructor*), 313

Semisimple Modules of a Polycyclic Group (*example*), 228
SemisimpleEFAModuleMaps(G) (*function*), 228
SemisimpleEFAModules(G) (*function*), 228
SemisimpleEFASeries(G) (*function*), 223
Seq(G) (*function*), 282, 296
Seq(G, a, b) (*function*), 282, 295
Seq(M) (*function*), 327
Seq(M, a, b) (*function*), 327
Set Operations on a Dihedral Group (*example*), 296
Set(G) (*function*), 281, 295
Set(G, a, b) (*function*), 281, 295
Set(M) (*function*), 327
Set(M, a, b) (*function*), 327
SetDisplayLevel(~P, Level) (*procedure*), 162
SetElementPrintFormat(~B, s) (*procedure*), 239
SetForceCFP(~B, b) (*procedure*), 238
SetGlobalTCParameters(: parameters) (*procedure*), 116
SetOptions(~P : parameters) (*procedure*), 86
SetPresentation(~B, s) (*procedure*), 238
SetProcessParameters(~P: parameters) (*procedure*), 107
SetVerbose("KBMAG", v) (*procedure*), 277, 288, 320
SetVerbose("NilpotentQuotient", n) (*procedure*), 171
ShowOptions(~P : parameters) (*procedure*), 86
Simple Quotients of a Perfect Group (*example*), 71, 180
SimpleEpimorphisms(P) (*function*), 180
SimpleQuotientProcess(F, deg1, deg2, ord1, ord2: parameters) (*function*), 180
SimpleQuotients(F, deg1, deg2, ord1, ord2: parameters) (*function*), 179
SimpleQuotients(F, ord1, ord2: parameters) (*function*), 179
SimpleQuotients(F, ord2: parameters) (*function*), 179
Simplify(~P : parameters) (*procedure*), 86
Simplify(G: parameters) (*function*), 84
Simplifying the Fibonacci Group F(8) (*example*), 41, 85
SimplifyLength(~P : parameters) (*procedure*), 86
SimplifyLength(G: parameters) (*function*), 85
SimplifyPresentation(~P : parameters) (*procedure*), 86
SLPGroup(n) (*function*), 305
Small Cancellation Conditions (*example*), 98
SmallCancellationConditions(G) (*function*), 98
Soluble Quotient of a Two-Generator Group (*example*), 69, 172
SolubleQuotient(F : parameters) (*function*), 176
SolubleQuotient(F, n : parameters) (*function*), 173, 176
SolubleQuotient(F, P : parameters) (*function*), 173, 176
SolubleQuotient(G : parameters) (*function*), 172
SolvableQuotient(F : parameters) (*function*), 176
SolvableQuotient(F, n : parameters) (*function*), 173, 176
SolvableQuotient(F, P : parameters) (*function*), 173, 176
SolvableQuotient(G : parameters) (*function*), 172
SpecifyCharacteristic(Q, n) (*function*), 186
SpecifyCharacteristic(Q, p) (*function*), 191
Specifying a Polycyclic Presentation (*example*), 207
Specifying the Characteristic (*example*), 186
Speed of SLP-Group Homomorphisms (*example*), 309
Splitting Relators into Red and Green (*example*), 301
Standard Groups as FP-Groups (*example*), 89
StartEnumeration(~P: parameters) (*procedure*), 107
StartNewClass(~P: parameters) (*procedure*), 160
String Operations on Words (*example*), 30
Structure of a Unit Group (*example*), 7
Structure of the Fibonacci Group F(7) (*example*), 65, 156
sub< G | f > (*constructor*), 99
sub< G | L > (*constructor*), 99
sub<A | L: parameters> (*constructor*), 10
sub<A | L> (*constructor*), 9
sub<G | L> (*constructor*), 209
sub<S | L_1, ..., L_r> (*constructor*), 314
Subgroup Constructions (*example*), 138
Subgroup Defined by a Homomorphism (*example*), 100
Subgroup of a Quadratic Forms Group (*example*), 10
Subgroup of Index 448 (*example*), 47, 100
Subgroup Predicates in a Space Group (*example*), 55, 142
Subgroup Structure of a Wreath Product (*example*), 221
Subgroup(P) (*function*), 111
Subgroup(V) (*function*), 122
SubgroupElements(G, H, n) (*function*), 32
Subgroups of a Free Group (*example*), 32
Subgroups of an Abelian Group of Order 12 (*example*), 20
Subgroups(G:parameters) (*function*), 20
SubnormalSeries(G, H) (*function*), 19
Substitute(u, f, n, v) (*function*), 30, 81, 317
Subword(u, f, n) (*function*), 30, 81, 317
Super Summit Sets of Braids (*example*), 259
Supergroup(F) (*function*), 31

SuperSummitCanonicalLength(u: parameters) (*function*), 245
SuperSummitInfimum(u: parameters) (*function*), 245
SuperSummitProcess(u: parameters) (*function*), 261
SuperSummitRepresentative(u: parameters) (*function*), 259
SuperSummitSet(u: parameters) (*function*), 259
SuperSummitSupremum(u: parameters) (*function*), 245
Supremum(u: parameters) (*function*), 245
Sylow 2-Subgroup of a Unit Group (*example*), 18
Sylow(G, p : parameters) (*function*), 18
SylowSubgroup(G, p : parameters) (*function*), 18
Sym(4) as a Quotient of a Free Group (*example*), 78
Sym(GrpFP, n) (*function*), 89
Symmetric Group as Image of a Braid Group (*example*), 60
Symmetric Group Sym(8) as an FP-Group (*example*), 43
SymmetricFPGroup(n) (*function*), 89
SymmetricRepresentation(B) (*function*), 269

T

T(i, w) (*literal*), 122
Tails(~P, k: parameters) (*procedure*), 160
Tails(~P: parameters) (*procedure*), 160
TietzeProcess(G: parameters) (*function*), 86
Todd-Coxeter Enumeration of 448 Cosets (*example*), 102
Todd-Coxeter for the Harada-Norton Group (*example*), 102
ToddCoxeter(G, H: parameters) (*function*), 101
TorsionFreeRank(A) (*function*), 15
TorsionFreeRank(G) (*function*), 156
TorsionFreeSubgroup(A) (*function*), 15
TorsionInvariants(A) (*function*), 15
TorsionSubgroup(A) (*function*), 15
TotalNumberOfCosets(P) (*function*), 111
Transferring Subgroups and Quotients (*example*), 214
Transport(x, s: parameters) (*function*), 264
Transversal(G, H) (*function*), 17, 122, 138, 217
Transversal(G, H, K) (*function*), 138
Tutte's 8-Cage and the Conder Graph (*example*), 51, 130

U

u * m (*operation*), 9
u * v (*operation*), 28, 249, 280, 293, 306, 312, 325
u *:= v (*operation*), 249
u + v (*operation*), 9
u / v (*operation*), 249, 280, 293
u /:= v (*operation*), 249
u @@ f (*literal*), 266
u - v (*operation*), 9
u ^ m (*operation*), 306
u ^ n (*operation*), 28, 249, 280, 294, 312, 325
u ^ v (*operation*), 28, 250, 280, 294, 306
u ^:= n (*operation*), 250
u ^:= v (*operation*), 250
u eq v (*operation*), 14, 29, 253, 280, 294, 308, 312, 325
u ge v (*operation*), 29, 253, 313
u gt v (*operation*), 29, 313
u in B (*operation*), 252
u in H (*operation*), 141
u in P (*operation*), 262
u le v (*operation*), 29, 253, 313
u lt v (*operation*), 29, 312
u ne v (*operation*), 14, 29, 253, 280, 294, 308, 312, 326
u notin B (*operation*), 252
u notin H (*operation*), 141
u notin P (*operation*), 262
Ultra Summit Sets from Minimal Simple Elements (*example*), 265
UltraSummitProcess(u: parameters) (*function*), 261
UltraSummitRepresentative(u: parameters) (*function*), 259
UltraSummitSet(u: parameters) (*function*), 259
Universe(A) (*procedure*), 6
UnsetGlobalTCParameters() (*procedure*), 116
UpperCentralSeries(G) (*function*), 19, 223
UserGenerators(A) (*function*), 6
UserRepresentation(g) (*function*), 8

V

Varying the Epsilon Parameter (*example*), 303
Verbose Hyperbolicity Testing (*example*), 298

W

w @ f (*literal*), 147
Weight(w, x) (*function*), 27
WhiteheadReduction(F, Q) (*function*), 33
Word Arithmetic in a Fibonacci Group (*example*), 281
Word Arithmetic in a Fibonacci Monoid (*example*), 326
Word Operations in an Automatic Group (*example*), 294
Word Problem via the Dehn Algorithm (*example*), 304
WordAcceptor(G) (*function*), 291
WordAcceptorSize(G) (*function*), 291
WordDifferenceAutomaton(G) (*function*), 291
WordDifferences(G) (*function*), 291
WordDifferenceSize(G) (*function*), 291
WordToSequence(u: parameters) (*function*), 244
Working with a Coset Space (*example*), 123
w_1 = w_2 (*operation*), 1, 313

X

`x in H` (*operation*), 31