Semigroups

Version 3.0.7

J. D. Mitchell
Manuel Delgado
James East
Attila Egri-Nagy
Nicholas Ham
Julius Jonušas
Markus Pfeiffer
Ben Steinberg
Jhevon Smith
Michael Torpey
Wilf A. Wilson

J. D. Mitchell Email: jdm3@st-and.ac.uk
Homepage: http://tinyurl.com/jdmitchell

Abstract

The Semigroups package is a GAP package containing methods for semigroups, monoids, and inverse semigroups. There are particularly efficient methods for semigroups or ideals consisting of transformations, partial permutations, bipartitions, partitioned binary relations, subsemigroups of regular Rees 0-matrix semigroups, and matrices of various semirings including boolean matrices, matrices over finite fields, and certain tropical matrices.

Semigroups contains efficient methods for creating semigroups, monoids, and inverse semigroup, calculating their Green's structure, ideals, size, elements, group of units, small generating sets, testing membership, finding the inverses of a regular element, factorizing elements over the generators, and so on. It is possible to test if a semigroup satisfies a particular property, such as if it is regular, simple, inverse, completely regular, and a variety of further properties.

There are methods for finding presentations for a semigroup, the congruences of a semigroup, the normalizer of a semigroup in a permutation group, the maximal subsemigroups of a finite semigroup, smaller degree partial permutation representations, and the character tables of inverse semigroups. There are functions for producing pictures of the Green's structure of a semigroup, and for drawing graphical representations of certain types of elements.

Copyright

Acknowledgements

I would like to thank P. von Bunau, A. Distler, S. Linton, C. Nehaniv, J. Neubueser, M. R. Quick, E. F. Robertson, and N. Ruskuc for their help and suggestions. Special thanks go to J. Araujo for his mathematical suggestions and to M. Neunhoeffer for his invaluable help in improving the efficiency of the package.

Stuart Burrell contributed methods for checking finiteness of semigroups of matrices of the max-plus and min-plus semirings.

Manuel Delgado and Attila Egri-Nagy contributed to the functions `Splash` (18.1.1) and `DotString` (18.2.1).

James East, Attila Egri-Nagy, and Markus Pfeiffer contributed to the part of the package relating to bipartitions. I would like to thank the University of Western Sydney for their support of the development of this part of the package.

Nick Ham contributed many of the standard examples of bipartition semigroups.

Julius Jonušas contributed the part of the package relating to free inverse semigroups, and contributed to the code for ideals.

Markus Pfeiffer contributed the majority of the code relating to semigroups of matrices over finite fields.

Yann Péresse and Yanhui Wang contributed to the attribute `MunnSemigroup` (8.2.1).

Jhevon Smith and Ben Steinberg contributed the function `CharacterTableOfInverseSemigroup` (15.1.10).

Michael Torpey contributed the part of the package relating to congruences.

Wilf A. Wilson contributed to the part of the package relating maximal subsemigroups and smaller degree partial permutation representations of inverse semigroups. We are also grateful to C. Donoven and R. Hancock for their contribution to the development of the algorithms for maximal subsemigroups and smaller degree partial permutation representations.

We would also like to acknowledge the support of: EPSRC grant number GR/S/56085/01; the Carnegie Trust for the Universities of Scotland for funding the PhD scholarships of J. Jonušas and W. Wilson when they worked on this project; the Engineering and Physical Sciences Research Council (EPSRC) for funding the PhD scholarship of M. Torpey when he worked on this project (EP/M506631/1).

Contents

Chapter 1

The Semigroups package

1.1 Introduction

This is the manual for the Semigroups package for GAP version 3.0.7. Semigroups 3.0.7 is a distant descendant of the Monoid package for GAP 3 by Goetz Pfeiffer, Steve A. Linton, Edmund F. Robertson, and Nik Ruskuc.

Semigroups 3.0.7 contains efficient methods for creating semigroups, monoids, inverse semigroups and their ideals, calculating their Green's structure, size, elements, group of units, minimal ideal, and testing membership, finding the inverses of a regular element, and factorizing elements over the generators, and much more; see below for more details.

There are methods for finding: congruences of semigroups, the normalizer of a semigroup in a permutation group (using the method from [ABMN10]), the maximal subsemigroups of a finite semigroup (based on [GGR68] and described in [DMW16]), smaller degree partial permutation representations (based on [Sch92]) and the character table of an inverse semigroup. There are functions for producing pictures of the Green's structure of a semigroup (inspired by sgpviz), and for drawing graphical representations of the elements of certain semigroups.

Many standard examples of semigroups and classes of semigroups are provided; see Section 8. Semigroups also provides functions to read and write collections of elements of a semigroup to a file; see `ReadGenerators` (19.1.1) and `WriteGenerators` (19.1.2).

There are functions in Semigroups to define and manipulate free inverse semigroups and free bands; this part of the package was written by Julius Jonušas; see Chapters 10.

From Version 3.0.0, Semigroups includes a copy of the libsemigroups C++ library which contains an implementation of the Froidure-Pin Algorithm.

MINOR NOTE OF CAUTION: Semigroups contains different methods for some GAP library functions, and so you might notice that GAP behaves differently when Semigroups is loaded. For more details about semigroups in GAP or Green's relations in particular, see (**Reference: Semigroups**) or (**Reference: Green's Relations**).

If you find a bug or an issue with the package, then report this using the issue tracker.

1.2 Overview

This manual is organised as follows:

Part I: generators

the different types of generators that are introduced in Semigroups are described in Chapters 3, 4, and 5. These include `Bipartition` (3.2.1), `PBR` (4.2.1), and `Matrix` (5.1.5), which supplement those already defined in the GAP library, such as `Transformation` (**Reference: Transformation (for an image list)**) or `PartialPerm` (**Reference: PartialPerm (for a domain and image)**).

Part II: semigroups and ideals
functions and operations for creating semigroups, monoids, and their ideals, in general, and various options, are described in Chapters 6 and 7.

Part III: standard examples and constructions
standard examples of semigroups, such as `FullBooleanMatMonoid` (8.6.1) or `UniformBlockBijectionMonoid` (8.3.8), are described in Chapter 8, and standard constructions, such as `TrivialSemigroup` (9.1.1), `RightZeroSemigroup` (9.1.5), are described in Chapter 9.

Part IV: further classes of semigroups and monoids
free objects in the categories of inverse semigroups, and bands, are described in Chapter 10, and graph inverse semigroups, which are a generalisation of polycyclic monoids, are described in Chapter 11.

Part V: the structure of a semigroup or monoid
the functionality of the Semigroups package for determining various structural properties of a given semigroup or monoid are described in Chapters 12, 13, and 14. Attributes and properties specific to inverse semigroups are described in Chapter 15.

Part VI: congruences, quotients, and homomorphisms
methods for creating and manipulating congruences and homomorphisms are described by Chapters 16 and 17.

Part VII: utilities and helper functions
functions for reading and writing semigroups and their elements, and for visualising semigroups, and some of their elements, can be found in Chapters 18 and 19.

Chapter 2

Installing Semigroups

2.1 For those in a hurry

In this section we give a brief description of how to start using Semigroups.

It is assumed that you have a working copy of GAP with version number 4.9.0 or higher. The most up-to-date version of GAP and instructions on how to install it can be obtained from the main GAP webpage `http://www.gap-system.org`.

The following is a summary of the steps that should lead to a successful installation of Semigroups:

- ensure that the IO package version 4.4.4 or higher is available. IO must be compiled before Semigroups can be loaded.

- ensure that the Orb package version 4.7.5 or higher is available. Orb and Semigroups both perform better if Orb is compiled.

- ensure that the Digraphs package version 0.7.1 or higher is available. Digraphs must be compiled before Semigroups can be loaded.

- ensure that the genss package version 1.5 or higher is available.

- download the package archive `semigroups-3.0.7.tar.gz` from the Semigroups package webpage.

- unzip and untar the file, this should create a directory called `semigroups-3.0.7`.

- locate the `pkg` directory of your GAP directory, which contains the directories `lib`, `doc` and so on. Move the directory `semigroups-3.0.7` into the `pkg` directory.

- from version 3.0.0, it is necessary to compile the Semigroups package. Semigroups uses the libsemigroups C++ library, which requires a compiler implementing the C++11 standard. Inside the `pkg/semigroups-3.0.7` directory, type

  ```
  ./configure
  make
  ```

  Further information about this step can be found in Section 2.3.

- start GAP in the usual way (i.e. type gap at the command line).
- type LoadPackage("semigroups");

PLEASE NOTE THAT *from version 3.0.0: Semigroups can only be loaded if it has been compiled.*

If you want to check that the package is working correctly, you should run some of the tests described in Section 2.5.

2.2 Package dependencies

The Semigroups package is written in GAP and C++ code and requires the Orb, IO, Digraphs and genss packages. The Orb package is used to efficiently compute components of actions, which underpin many of the features of Semigroups. The IO package is used to read and write elements of a semigroup to a file. The genss package is used in a non-deterministic version of the operation Normalizer (13.11.1) and in calculating the stabiliser of a Rees 0-matrix semigroup's matrix. The Digraphs package is used for in variety of ways in the Semigroups package, in particular, to apply standard graph theoretic algorithms to certain data structures.

2.3 Compiling the kernel module

As of version 3.0.0, the Semigroups package has a GAP kernel component in C/C++ which must be compiled. This component contains low-level functions relating to the enumeration of certain types of semigroups, and it is not possible to use the Semigroups package without compiling it.

To compile the kernel component inside the pkg/semigroups-3.0.7 directory, type

```
./configure
make
```

If you installed the package in another pkg directory other than the standard pkg directory in your GAP installation, then you have to do two things. Firstly during compilation you have to use the option -with-gaproot=PATH of the configure script where PATH is a path to the main GAP root directory (if not given the default ../.. is assumed).

If you installed GAP on several architectures, you must execute the configure/make step for each of the architectures. You can either do this immediately after configuring and compiling GAP itself on this architecture, or alternatively set the environment variable CONFIGNAME to the name of the configuration you used when compiling GAP before running ./configure. Note however that your compiler choice and flags (environment variables CC and CFLAGS) need to be chosen to match the setup of the original GAP compilation. For example you have to specify 32-bit or 64-bit mode correctly!

2.4 Rebuilding the documentation

The Semigroups package comes complete with pdf, html, and text versions of the documentation. However, you might find it necessary, at some point, to rebuild the documentation. To rebuild the documentation use the SemigroupsMakeDoc (2.4.1).

2.4.1 SemigroupsMakeDoc

▷ `SemigroupsMakeDoc()` (function)

Returns: Nothing.

This function should be called with no argument to compile the Semigroups documentation.

2.5 Testing your installation

In this section we describe how to test that Semigroups is working as intended. To quickly test that Semigroups is installed correctly use `SemigroupsTestInstall` (2.5.1) - this will take a few seconds. For more extensive tests use `SemigroupsTestStandard` (2.5.2) - this may take several minutes. Finally, for lengthy benchmarking tests use `SemigroupsTestExtreme` (2.5.3) - this may take more than half an hour.

If something goes wrong, then please review the instructions in Section 2.1 and ensure that Semigroups has been properly installed. If you continue having problems, please use the issue tracker to report the issues you are having.

2.5.1 SemigroupsTestInstall

▷ `SemigroupsTestInstall()` (function)

Returns: `true` or `false`.

This function should be called with no argument to test your installation of Semigroups is working correctly. These tests should take no more than a few seconds to complete. To more comprehensively test that Semigroups is installed correctly use `SemigroupsTestStandard` (2.5.2).

2.5.2 SemigroupsTestStandard

▷ `SemigroupsTestStandard()` (function)

Returns: A list indicating which tests passed and failed and the time take to run each file.

This function should be called with no argument to comprehensively test that Semigroups is working correctly. These tests should take no more than a few minutes to complete. To quickly test that Semigroups is installed correctly use `SemigroupsTestInstall` (2.5.1).

Each test file is run twice, once when the methods for `IsActingSemigroup` (6.1.3) are enabled and once when they are disabled.

2.5.3 SemigroupsTestExtreme

▷ `SemigroupsTestExtreme()` (function)

Returns: A list indicating which tests passed and failed and the time take to run each file.

This function should be called with no argument to run some long-running tests, which could be used to benchmark Semigroups or test your hardware. These tests should take no more than around half an hour to complete. To quickly test that Semigroups is installed correctly use `SemigroupsTestInstall` (2.5.1), or to test all aspects of the package use `SemigroupsTestStandard` (2.5.2).

Each test file is run twice, once when the methods for semigroups satisfying `IsActingSemigroup` (6.1.3) are enabled and once when they are disabled.

2.6 More information during a computation

2.6.1 InfoSemigroups

▷ `InfoSemigroups` (info class)

`InfoSemigroups` is the info class of the Semigroups package. The info level is initially set to 0 and no info messages are displayed. To increase the amount of information displayed during a computation increase the info level to 2 or 3. To stop all info messages from being displayed, set the info level to 0. See also (**Reference: Info Functions**) and `SetInfoLevel` (**Reference: InfoLevel**).

Chapter 3

Bipartitions and blocks

In this chapter we describe the functions in Semigroups for creating and manipulating bipartitions and semigroups of bipartitions. We begin by describing what these objects are.

A *partition* of a set X is a set of pairwise disjoint non-empty subsets of X whose union is X. A partition of X is the collection of equivalence classes of an equivalence relation on X, and vice versa.

Let $n \in \mathbb{N}$, let **n**$= \{1,2,\ldots,n\}$, and let $-$**n**$= \{-1,-2,\ldots,-n\}$.

The *partition monoid* of degree n is the set of all partitions of **n**∪-**n** with a multiplication we describe below. To avoid conflict with other uses of the word "partition" in GAP, and to reflect their definition, we have opted to refer to the elements of the partition monoid as *bipartitions* of degree n; we will do so from this point on.

Let x be any bipartition of degree n. Then x is a set of pairwise disjoint non-empty subsets of **n**∪-**n** whose union is **n**∪-**n**; these subsets are called the *blocks* of x. A block containing elements of both **n** and -**n** is called a *transverse block*. If i, j∈**n**∪-**n** belong to the same block of a bipartition x, then we write (i, j)∈x.

Let x and y be bipartitions of degree n. Their product xy can be described as follows. Define **n**'$= \{1',2',\ldots,n'\}$. From x, create a partition x' of the set **n**∪**n**' by replacing each negative point -i in a block of x by the point i', and create from y a partition y' of the set **n**'∪-**n** by replacing each positive point i in a block of y by the point i'. Then define a relation on the set **n**∪**n**'∪-**n**, where i and j are related if they are related in either x' or y', and let p be the transitive closure of this relation. Finally, define xy to be the bipartition of degree n defined by the restriction of the equivalence relation p to the set **n**∪-**n**.

Equivalently, the product xy is defined to be the bipartition where i,j∈**n**∪-**n** (we assume without loss of generality that $i \geq j$) belong to the same block of xy if either:

- i=j,
- i, j ∈ **n** and (i,j)∈ x, or
- i, j ∈ -**n** and (i,j)∈ y;

or there exists $r \in \mathbb{N}$ and $k(1),k(2),\ldots,k(r) \in$ **n** , and one of the following holds:

- $r = 2s-1$ for some $s \geq 1$, i∈**n**, j∈ -**n** and

$$(i,-k(1)) \in x,\ (k(1),k(2)) \in y,\ (-k(2),-k(3)) \in x,\ \ldots,$$

$$\ldots,\ (-k(2s-2),-k(2s-1)) \in x,\ (k(2s-1),j) \in y;$$

- $r=2s$ for some $s \geq 1$, and either $i,j\in$**n**, and

  $(i,-k(1)) \in x,\ (k(1),k(2)) \in y,\ (-k(2),-k(3)) \in x,\ \ldots,(k(2s-1),k(2s)) \in y,\ (-k(2s),j) \in x,$

  or $i,j\in$-**n**, and

  $(i,k(1)) \in y,\ (-k(1),-k(2)) \in x,\ (k(2),k(3)) \in y,\ \ldots,(-k(2s-1),-k(2s)) \in x,\ (k(2s),j) \in y.$

This multiplication can be shown to be associative, and so the collection of all bipartitions of any particular degree is a monoid; the identity element of the partition monoid of degree n is the bipartition $\{\{i,-i\} : i \in \mathbf{n}\}$. A bipartition is a unit if and only if each block is of the form $\{i,\text{-}j\}$ for some i, $j\in$**n**. Hence the group of units is isomorphic to the symmetric group on **n**.

Let x be a bipartition of degree n. Then we define x^* to be the bipartition obtained from x by replacing i by -i and -i by i in every block of x for all $i\in$**n**. It is routine to verify that if x and y are arbitrary bipartitions of equal degree, then

$$(x^*)^* = x, \quad xx^*x = x, \quad x^*xx^* = x^*, \quad (xy)^* = y^*x^*.$$

In this way, the partition monoid is a *regular *-semigroup*.

A bipartition x of degree n is called *planar* if there do not exist distinct blocks $A,U \in x$, along with $a,b \in A$ and $u,v \in U$, such that $a < u < b < v$. Define p to be the bipartition of degree n with blocks $\{\{i,-(i+1)\} : i \in \{1,\ldots,n-1\}\}$ and $\{n,-1\}$. Note that p is a unit. A bipartition x of degree n is called *annular* if $x = p^iyp^j$ for some planar bipartition y of degree n, and some integers i and j.

From a graphical perspective, as on Page 873 in [HR05], a bipartition of degree n is planar if it can be represented as a graph without edges crossing inside of the rectangle formed by its vertices **n**∪-**n**. Similarly, as shown in Figure 2 in [Aui12], a bipartition of degree n is annular if it can be represented as a graph without edges crossing inside an annulus.

3.1 The family and categories of bipartitions

3.1.1 IsBipartition

▷ `IsBipartition(obj)` (Category)

Returns: `true` or `false`.

Every bipartition in GAP belongs to the category `IsBipartition`. Basic operations for bipartitions are `RightBlocks` (3.5.5), `LeftBlocks` (3.5.6), `ExtRepOfObj` (3.5.3), `LeftProjection` (3.2.4), `RightProjection` (3.2.5), `StarOp` (3.2.6), `DegreeOfBipartition` (3.5.1), `RankOfBipartition` (3.5.2), multiplication of two bipartitions of equal degree is via `*`.

3.1.2 IsBipartitionCollection

▷ `IsBipartitionCollection(obj)` (Category)

▷ `IsBipartitionCollColl(obj)` (Category)

Returns: `true` or `false`.

Every collection of bipartitions belongs to the category `IsBipartitionCollection`. For example, bipartition semigroups belong to `IsBipartitionCollection`.

Every collection of collections of bipartitions belongs to `IsBipartitionCollColl`. For example, a list of bipartition semigroups belongs to `IsBipartitionCollColl`.

3.2 Creating bipartitions

There are several ways of creating bipartitions in GAP, which are described in this section.

3.2.1 Bipartition

▷ `Bipartition(blocks)` (function)

Returns: A bipartition.

`Bipartition` returns the bipartition `x` with equivalence classes *blocks*, which should be a list of duplicate-free lists whose union is `[-n .. -1]` union `[1 .. n]` for some positive integer `n`.

`Bipartition` returns an error if the argument does not define a bipartition.

Example
```
gap> x := Bipartition([[1, -1], [2, 3, -3], [-2]]);
<bipartition: [ 1, -1 ], [ 2, 3, -3 ], [ -2 ]>
```

3.2.2 BipartitionByIntRep

▷ `BipartitionByIntRep(list)` (operation)

Returns: A bipartition.

It is possible to create a bipartition using its internal representation. The argument *list* must be a list of positive integers not greater than `n`, of length `2 * n`, and where `i` appears in the list only if `i-1` occurs earlier in the list.

For example, the internal representation of the bipartition with blocks

Example
```
[1, -1], [2, 3, -2], [-3]
```

has internal representation

Example
```
[1, 2, 2, 1, 2, 3]
```

The internal representation indicates that the number `1` is in class `1`, the number `2` is in class `2`, the number `3` is in class `2`, the number `-1` is in class `1`, the number `-2` is in class `2`, and `-3` is in class `3`. As another example, `[1, 3, 2, 1]` is not the internal representation of any bipartition since there is no `2` before the `3` in the second position.

In its first form `BipartitionByIntRep` verifies that the argument *list* is the internal representation of a bipartition.

See also `IntRepOfBipartition` (3.5.4).

Example
```
gap> BipartitionByIntRep([1, 2, 2, 1, 3, 4]);
<bipartition: [ 1, -1 ], [ 2, 3 ], [ -2 ], [ -3 ]>
```

3.2.3 IdentityBipartition

▷ `IdentityBipartition(n)` (operation)

Returns: The identity bipartition.

Returns the identity bipartition with degree *n*.

Example
```
gap> IdentityBipartition(10);
<block bijection: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ], [ 4, -4 ],
 [ 5, -5 ], [ 6, -6 ], [ 7, -7 ], [ 8, -8 ], [ 9, -9 ], [ 10, -10 ]>
```

3.2.4 LeftOne (for a bipartition)

▷ `LeftOne(x)` (attribute)

▷ `LeftProjection(x)` (attribute)

Returns: A bipartition.

The `LeftProjection` of a bipartition *x* is the bipartition `x * Star(x)`. It is so-named, since the left and right blocks of the left projection equal the left blocks of *x*.

The left projection `e` of *x* is also a bipartition with the property that `e * x = x`. `LeftOne` and `LeftProjection` are synonymous.

Example

```
gap> x := Bipartition([
>   [1, 4, -1, -2, -6], [2, 3, 5, -4], [6, -3], [-5]]);;
gap> LeftOne(x);
<block bijection: [ 1, 4, -1, -4 ], [ 2, 3, 5, -2, -3, -5 ],
 [ 6, -6 ]>
gap> LeftBlocks(x);
<blocks: [ 1*, 4* ], [ 2*, 3*, 5* ], [ 6* ]>
gap> RightBlocks(LeftOne(x));
<blocks: [ 1*, 4* ], [ 2*, 3*, 5* ], [ 6* ]>
gap> LeftBlocks(LeftOne(x));
<blocks: [ 1*, 4* ], [ 2*, 3*, 5* ], [ 6* ]>
gap> LeftOne(x) * x = x;
true
```

3.2.5 RightOne (for a bipartition)

▷ `RightOne(x)` (attribute)

▷ `RightProjection(x)` (attribute)

Returns: A bipartition.

The `RightProjection` of a bipartition *x* is the bipartition `Star(x) * x`. It is so-named, since the left and right blocks of the right projection equal the right blocks of *x*.

The right projection `e` of *x* is also a bipartition with the property that `x * e = x`. `RightOne` and `RightProjection` are synonymous.

Example

```
gap> x := Bipartition([[1, -1, -4], [2, -2, -3], [3, 4], [5, -5]]);;
gap> RightOne(x);
<block bijection: [ 1, 4, -1, -4 ], [ 2, 3, -2, -3 ], [ 5, -5 ]>
gap> RightBlocks(RightOne(x));
<blocks: [ 1*, 4* ], [ 2*, 3* ], [ 5* ]>
gap> LeftBlocks(RightOne(x));
<blocks: [ 1*, 4* ], [ 2*, 3* ], [ 5* ]>
gap> RightBlocks(x);
<blocks: [ 1*, 4* ], [ 2*, 3* ], [ 5* ]>
gap> x * RightOne(x) = x;
true
```

3.2.6 StarOp (for a bipartition)

▷ `StarOp(x)` (operation)

▷ `Star(x)` (attribute)

Returns: A bipartition.

StarOp returns the unique bipartition g with the property that: x * g * x = x, RightBlocks(x) = LeftBlocks(g), and LeftBlocks(x) = RightBlocks(g). The star g can be obtained from x by changing the sign of every integer in the external representation of x.

Example
```
gap> x := Bipartition([[1, -4], [2, 3, 4], [5], [-1], [-2, -3], [-5]]);
<bipartition: [ 1, -4 ], [ 2, 3, 4 ], [ 5 ], [ -1 ], [ -2, -3 ],
 [ -5 ]>
gap> y := Star(x);
<bipartition: [ 1 ], [ 2, 3 ], [ 4, -1 ], [ 5 ], [ -2, -3, -4 ],
 [ -5 ]>
gap> x * y * x = x;
true
gap> LeftBlocks(x) = RightBlocks(y);
true
gap> RightBlocks(x) = LeftBlocks(y);
true
```

3.2.7 RandomBipartition

▷ RandomBipartition([rs,]n) (operation)

▷ RandomBlockBijection([rs,]n) (operation)

Returns: A bipartition.

If n is a positive integer, then RandomBipartition returns a random bipartition of degree n, and RandomBlockBijection returns a random block bijection of degree n.

If the optional first argument rs is a random source, then this is used to generate the bipartition returned by RandomBipartition and RandomBlockBijection.

Note that neither of these functions has a uniform distribution.

Example
```
gap> x := RandomBipartition(6);
<bipartition: [ 1, 2, 3, 4 ], [ 5 ], [ 6, -2, -3, -4 ], [ -1, -5 ], [ -6 ]>
gap> x := RandomBlockBijection(4);
<block bijection: [ 1, 4, -2 ], [ 2, -4 ], [ 3, -1, -3 ]>
```

3.3 Changing the representation of a bipartition

It is possible that a bipartition can be represented as another type of object, or that another type of GAP object can be represented as a bipartition. In this section, we describe the functions in the Semigroups package for changing the representation of bipartition, or for changing the representation of another type of object to that of a bipartition.

The operations AsPermutation (3.3.5), AsPartialPerm (3.3.4), AsTransformation (3.3.3) can be used to convert bipartitions into permutations, partial permutations, or transformations where appropriate.

3.3.1 AsBipartition

▷ AsBipartition(x[, n]) (operation)

Returns: A bipartition.

`AsBipartition` returns the bipartition, permutation, transformation, or partial permutation *x*, as a bipartition of degree *n*.

There are several possible arguments for `AsBipartition`:

permutations
: If *x* is a permutation and *n* is a positive integer, then `AsBipartition(x, n)` returns the bipartition on `[1 .. n]` with classes `[i, i ^ x]` for all `i = 1 .. n`.

  If no positive integer *n* is specified, then the largest moved point of *x* is used as the value for *n*; see `LargestMovedPoint` (**Reference: LargestMovedPoint (for a permutation)**).

transformations
: If *x* is a transformation and *n* is a positive integer such that *x* is a transformation of `[1 .. n]`, then `AsTransformation` returns the bipartition with classes $(i)f^{-1} \cup \{i\}$ for all `i` in the image of *x*.

  If the positive integer *n* is not specified, then the degree of *x* is used as the value for *n*.

partial permutations
: If *x* is a partial permutation and *n* is a positive integer, then `AsBipartition` returns the bipartition with classes `[i, i ^ x]` for `i` in `[1 .. n]`. Thus the degree of the returned bipartition is the maximum of *n* and the values `i ^ x` where `i` in `[1 .. n]`.

  If the optional argument *n* is not present, then the default value of the maximum of the largest moved point and the largest image of a moved point of *x* plus `1` is used.

bipartitions
: If *x* is a bipartition and *n* is a non-negative integer, then `AsBipartition` returns a bipartition corresponding to *x* with degree *n*.

  If *n* equals the degree of *x*, then *x* is returned. If *n* is less than the degree of *x*, then this function returns the bipartition obtained from *x* by removing the values exceeding *n* or less than -*n* from the blocks of *x*. If *n* is greater than the degree of *x*, then this function returns the bipartition with the same blocks as *x* and the singleton blocks `i` and `-i` for all `i` greater than the degree of *x*

pbrs
: If *x* is a pbr satisfying `IsBipartitionPBR` (4.5.8) and *n* is a non-negative integer, then `AsBipartition` returns the bipartition corresponding to *x* with degree *n*.

Example

```
gap> x := Transformation([3, 5, 3, 4, 1, 2]);;
gap> AsBipartition(x, 5);
<bipartition: [ 1, 3, -3 ], [ 2, -5 ], [ 4, -4 ], [ 5, -1 ], [ -2 ]>
gap> AsBipartition(x);
<bipartition: [ 1, 3, -3 ], [ 2, -5 ], [ 4, -4 ], [ 5, -1 ],
 [ 6, -2 ], [ -6 ]>
gap> AsBipartition(x, 10);
<bipartition: [ 1, 3, -3 ], [ 2, -5 ], [ 4, -4 ], [ 5, -1 ],
 [ 6, -2 ], [ 7, -7 ], [ 8, -8 ], [ 9, -9 ], [ 10, -10 ], [ -6 ]>
gap> AsBipartition((1, 3)(2, 4));
<block bijection: [ 1, -3 ], [ 2, -4 ], [ 3, -1 ], [ 4, -2 ]>
gap> AsBipartition((1, 3)(2, 4), 10);
<block bijection: [ 1, -3 ], [ 2, -4 ], [ 3, -1 ], [ 4, -2 ],
 [ 5, -5 ], [ 6, -6 ], [ 7, -7 ], [ 8, -8 ], [ 9, -9 ], [ 10, -10 ]>
```

```
gap> x := PartialPerm([1, 2, 3, 4, 5, 6], [6, 7, 1, 4, 3, 2]);;
gap> AsBipartition(x, 11);
<bipartition: [ 1, -6 ], [ 2, -7 ], [ 3, -1 ], [ 4, -4 ], [ 5, -3 ],
 [ 6, -2 ], [ 7 ], [ 8 ], [ 9 ], [ 10 ], [ 11 ], [ -5 ], [ -8 ],
 [ -9 ], [ -10 ], [ -11 ]>
gap> AsBipartition(x);
<bipartition: [ 1, -6 ], [ 2, -7 ], [ 3, -1 ], [ 4, -4 ], [ 5, -3 ],
 [ 6, -2 ], [ 7 ], [ -5 ]>
gap> AsBipartition(Transformation([1, 1, 2]), 1);
<block bijection: [ 1, -1 ]>
gap> x := Bipartition([[1, 2, -2], [3], [4, 5, 6, -1],
>                       [-3, -4, -5, -6]]);;
gap> AsBipartition(x, 0);
<empty bipartition>
gap> AsBipartition(x, 2);
<bipartition: [ 1, 2, -2 ], [ -1 ]>
gap> AsBipartition(x, 8);
<bipartition: [ 1, 2, -2 ], [ 3 ], [ 4, 5, 6, -1 ], [ 7 ], [ 8 ],
 [ -3, -4, -5, -6 ], [ -7 ], [ -8 ]>
gap> x := PBR(
> [[-1, 1, 2, 3, 4], [-1, 1, 2, 3, 4],
>  [-1, 1, 2, 3, 4], [-1, 1, 2, 3, 4]],
> [[-1, 1, 2, 3, 4], [-2], [-3], [-4]]);;
gap> AsBipartition(x);
<bipartition: [ 1, 2, 3, 4, -1 ], [ -2 ], [ -3 ], [ -4 ]>
gap> AsBipartition(x, 2);
<bipartition: [ 1, 2, -1 ], [ -2 ]>
gap> AsBipartition(x, 4);
<bipartition: [ 1, 2, 3, 4, -1 ], [ -2 ], [ -3 ], [ -4 ]>
gap> AsBipartition(x, 5);
<bipartition: [ 1, 2, 3, 4, -1 ], [ 5 ], [ -2 ], [ -3 ], [ -4 ],
 [ -5 ]>
gap> AsBipartition(x, 0);
<empty bipartition>
```

3.3.2 AsBlockBijection

▷ `AsBlockBijection(x[, n])` (operation)

Returns: A block bijection.

When the argument x is a partial perm and n is a positive integer which is greater than the maximum of the degree and codegree of x, this function returns a block bijection corresponding to x. This block bijection has the same non-singleton classes as `g := AsBipartition(x, n)` and one additional class which is the union the singleton classes of g.

If the optional second argument n is not present, then the maximum of the degree and codegree of x plus 1 is used by default. If the second argument n is not greater than this maximum, then an error is given.

This is the value at x of the embedding of the symmetric inverse monoid into the dual symmetric inverse monoid given in the FitzGerald-Leech Theorem [FL98].

When the argument x is a partial perm bipartition (see `IsPartialPermBipartition` (3.5.15)) then this operation returns `AsBlockBijection(AsPartialPerm(x)[, n])`.

Example

```
gap> x := PartialPerm([1, 2, 3, 6, 7, 10], [9, 5, 6, 1, 7, 8]);
[2,5][3,6,1,9][10,8](7)
gap> AsBipartition(x, 11);
<bipartition: [ 1, -9 ], [ 2, -5 ], [ 3, -6 ], [ 4 ], [ 5 ],
 [ 6, -1 ], [ 7, -7 ], [ 8 ], [ 9 ], [ 10, -8 ], [ 11 ], [ -2 ],
 [ -3 ], [ -4 ], [ -10 ], [ -11 ]>
gap> AsBlockBijection(x, 10);
Error, Semigroups: AsBlockBijection (for a partial perm and pos int):
the 2nd argument must be strictly greater than the maximum of the
degree and codegree of the 1st argument,
gap> AsBlockBijection(x, 11);
<block bijection: [ 1, -9 ], [ 2, -5 ], [ 3, -6 ],
 [ 4, 5, 8, 9, 11, -2, -3, -4, -10, -11 ], [ 6, -1 ], [ 7, -7 ],
 [ 10, -8 ]>
gap> x := Bipartition([[1, -3], [2], [3, -2], [-1]]);;
gap> IsPartialPermBipartition(x);
true
gap> AsBlockBijection(x);
<block bijection: [ 1, -3 ], [ 2, 4, -1, -4 ], [ 3, -2 ]>
```

3.3.3 AsTransformation (for a bipartition)

▷ AsTransformation(*x*) (attribute)

Returns: A transformation.

When the argument *x* is a bipartition, that mathematically defines a transformation, this function returns that transformation. A bipartition *x* defines a transformation if and only if its right blocks are the image list of a permutation of `[1 .. n]` where `n` is the degree of *x*.

See `IsTransBipartition` (3.5.12).

Example

```
gap> x := Bipartition([[1, -3], [2, -2], [3, 5, 10, -7],
>                      [4, -12], [6, 7, -6], [8, -5], [9, -11],
>                      [11, 12, -10], [-1], [-4], [-8], [-9]]);;
gap> AsTransformation(x);
Transformation( [ 3, 2, 7, 12, 7, 6, 6, 5, 11, 7, 10, 10 ] )
gap> IsTransBipartition(x);
true
gap> x := Bipartition([[1, 5], [2, 4, 8, 10],
>                      [3, 6, 7, -1, -2], [9, -4, -6, -9],
>                      [-3, -5], [-7, -8], [-10]]);;
gap> AsTransformation(x);
Error, Semigroups: AsTransformation (for a bipartition):
the argument does not define a transformation,
```

3.3.4 AsPartialPerm (for a bipartition)

▷ AsPartialPerm(*x*) (operation)

Returns: A partial perm.

When the argument *x* is a bipartition that mathematically defines a partial perm, this function returns that partial perm.

A bipartition *x* defines a partial perm if and only if its numbers of left and right blocks both equal its degree.

See `IsPartialPermBipartition` (3.5.15).

Example
```
gap> x := Bipartition([[1, -4], [2, -2], [3, -10], [4, -5],
>                     [5, -9], [6], [7], [8, -6], [9, -3], [10, -8],
>                     [-1], [-7]]);;
gap> IsPartialPermBipartition(x);
true
gap> AsPartialPerm(x);
[1,4,5,9,3,10,8,6](2)
gap> x := Bipartition([[1, -2, -4], [2, 3, 4, -3], [-1]]);;
gap> IsPartialPermBipartition(x);
false
gap> AsPartialPerm(x);
Error, Semigroups: AsPartialPerm (for a bipartition):
the argument does not define a partial perm,
```

3.3.5 AsPermutation (for a bipartition)

▷ `AsPermutation(x)` (attribute)

Returns: A permutation.

When the argument *x* is a bipartition that mathematically defines a permutation, this function returns that permutation.

A bipartition *x* defines a permutation if and only if its numbers of left, right, and transverse blocks all equal its degree.

See `IsPermBipartition` (3.5.14).

Example
```
gap> x := Bipartition([[1, -6], [2, -4], [3, -2], [4, -5],
>                     [5, -3], [6, -1]]);;
gap> IsPermBipartition(x);
true
gap> AsPermutation(x);
(1,6)(2,4,5,3)
gap> AsBipartition(last) = x;
true
```

3.4 Operators for bipartitions

f `*` *g*

returns the composition of *f* and *g* when *f* and *g* are bipartitions.

f `<` *g*

returns `true` if the internal representation of *f* is lexicographically less than the internal representation of *g* and `false` if it is not.

f `=` *g*

returns `true` if the bipartition *f* equals the bipartition *g* and returns `false` if it does not.

3.4.1 PartialPermLeqBipartition

▷ `PartialPermLeqBipartition(x, y)` (operation)

Returns: `true` or `false`.

If x and y are partial perm bipartitions, i.e. they satisfy `IsPartialPermBipartition` (3.5.15), then this function returns `AsPartialPerm(x) < AsPartialPerm(y)`.

3.4.2 NaturalLeqPartialPermBipartition

▷ `NaturalLeqPartialPermBipartition(x, y)` (operation)

Returns: `true` or `false`.

The *natural partial order* $\leq$ on an inverse semigroup `S` is defined by `s` $\leq$ `t` if there exists an idempotent `e` in `S` such that `s = et`. Hence if x and y are partial perm bipartitions, then $x \leq y$ if and only if `AsPartialPerm(x)` is a restriction of `AsPartialPerm(y)`.

`NaturalLeqPartialPermBipartition` returns `true` if `AsPartialPerm(x)` is a restriction of `AsPartialPerm(y)` and `false` if it is not. Note that since this is a partial order and not a total order, it is possible that x and y are incomparable with respect to the natural partial order.

3.4.3 NaturalLeqBlockBijection

▷ `NaturalLeqBlockBijection(x, y)` (operation)

Returns: `true` or `false`.

The *natural partial order* $\leq$ on an inverse semigroup `S` is defined by `s` $\leq$ `t` if there exists an idempotent `e` in `S` such that `s = et`. Hence if x and y are block bijections, then $x \leq y$ if and only if x contains y.

`NaturalLeqBlockBijection` returns `true` if x is contained in y and `false` if it is not. Note that since this is a partial order and not a total order, it is possible that x and y are incomparable with respect to the natural partial order.

Example

```
gap> x := Bipartition([[1, 2, -3], [3, -1, -2], [4, -4],
>                      [5, -5], [6, -6], [7, -7],
>                      [8, -8], [9, -9], [10, -10]]);;
gap> y := Bipartition([[1, -2], [2, -1], [3, -3],
>                      [4, -4], [5, -5], [6, -6], [7, -7],
>                      [8, -8], [9, -9], [10, -10]]);;
gap> z := Bipartition([Union([1 .. 10], [-10 .. -1])]);;
gap> NaturalLeqBlockBijection(x, y);
false
gap> NaturalLeqBlockBijection(y, x);
false
gap> NaturalLeqBlockBijection(z, x);
true
gap> NaturalLeqBlockBijection(z, y);
true
```

3.4.4 PermLeftQuoBipartition

▷ `PermLeftQuoBipartition(x, y)` (operation)

Returns: A permutation.

If x and y are bipartitions with equal left and right blocks, then `PermLeftQuoBipartition` returns the permutation of the indices of the right blocks of x (and y) induced by `Star(x) * y`.

`PermLeftQuoBipartition` verifies that x and y have equal left and right blocks, and returns an error if they do not.

Example
```
gap> x := Bipartition([[1, 4, 6, 7, 8, 10], [2, 5, -1, -2, -8],
>                      [3, -3, -6, -7, -9], [9, -4, -5], [-10]]);;
gap> y := Bipartition([[1, 4, 6, 7, 8, 10], [2, 5, -3, -6, -7, -9],
>                      [3, -4, -5], [9, -1, -2, -8], [-10]]);;
gap> PermLeftQuoBipartition(x, y);
(1,2,3)
gap> Star(x) * y;
<bipartition: [ 1, 2, 8, -3, -6, -7, -9 ], [ 3, 6, 7, 9, -4, -5 ],
 [ 4, 5, -1, -2, -8 ], [ 10 ], [ -10 ]>
```

3.5 Attributes for bipartitons

In this section we describe various attributes that a bipartition can possess.

3.5.1 DegreeOfBipartition

▷ `DegreeOfBipartition(x)` (attribute)
▷ `DegreeOfBipartitionCollection(x)` (attribute)

Returns: A positive integer.

The degree of a bipartition is, roughly speaking, the number of points where it is defined. More precisely, if x is a bipartition defined on `2 * n` points, then the degree of x is `n`.

The degree of a collection *coll* of bipartitions of equal degree is just the degree of any (and every) bipartition in *coll*. The degree of collection of bipartitions of unequal degrees is not defined.

Example
```
gap> x := Bipartition([[1, 7, -3, -8], [2, 6],
>                      [3], [4, -7, -9], [5, 9, -2],
>                      [8, -1, -4, -6], [-5]]);;
gap> DegreeOfBipartition(x);
9
gap> S := BrauerMonoid(5);
<regular bipartition *-monoid of degree 5 with 3 generators>
gap> IsBipartitionCollection(S);
true
gap> DegreeOfBipartitionCollection(S);
5
```

3.5.2 RankOfBipartition

▷ `RankOfBipartition(x)` (attribute)
▷ `NrTransverseBlocks(x)` (attribute)

Returns: The rank of a bipartition.

When the argument is a bipartition x, `RankOfBipartition` returns the number of blocks of x containing both positive and negative entries, i.e. the number of transverse blocks of x.

`NrTransverseBlocks` is just a synonym for `RankOfBipartition`.

Example
```
gap> x := Bipartition([[1, 2, 6, 7, -4, -5, -7], [3, 4, 5, -1, -3],
>                     [8, -9], [9, -2], [-6], [-8]]);
<bipartition: [ 1, 2, 6, 7, -4, -5, -7 ], [ 3, 4, 5, -1, -3 ],
 [ 8, -9 ], [ 9, -2 ], [ -6 ], [ -8 ]>
gap> RankOfBipartition(x);
4
```

3.5.3 ExtRepOfObj (for a bipartition)

▷ `ExtRepOfObj(x)` (operation)

Returns: A partition of `[1 .. 2 * n]`.

If `n` is the degree of the bipartition `x`, then `ExtRepOfObj` returns the partition of `[-n .. -1]` union `[1 .. n]` corresponding to `x` as a sorted list of duplicate-free lists.

Example
```
gap> x := Bipartition([[1, 5, -3], [2, 4, -2, -4], [3, -1, -5]]);
<block bijection: [ 1, 5, -3 ], [ 2, 4, -2, -4 ], [ 3, -1, -5 ]>
gap> ExtRepOfObj(x);
[ [ 1, 5, -3 ], [ 2, 4, -2, -4 ], [ 3, -1, -5 ] ]
```

3.5.4 IntRepOfBipartition

▷ `IntRepOfBipartition(x)` (attribute)

Returns: A list of positive integers.

If `x` is a bipartition with degree `n`, then `IntRepOfBipartition` returns the *internal representation* of `x`: a list of length `2 * n` containing positive integers which correspond to the blocks of `x`.

If `i` is in `[1 .. n]`, then `list[i]` refers to the point `i`; if `i` is in `[n + 1 .. 2 * n]`, then `list[i]` refers to the point `n - i` (a negative point). Two points lie in the same block of the bipartition if and only if their entries in the list are equal.

See also `BipartitionByIntRep` (3.2.2).

Example
```
gap> x := Bipartition([[1, -3], [3, 4], [2, -1, -2], [-4]]);
<bipartition: [ 1, -3 ], [ 2, -1, -2 ], [ 3, 4 ], [ -4 ]>
gap> IntRepOfBipartition(x);
[ 1, 2, 3, 3, 2, 2, 1, 4 ]
```

3.5.5 RightBlocks

▷ `RightBlocks(x)` (attribute)

Returns: The right blocks of a bipartition.

`RightBlocks` returns the right blocks of the bipartition `x`.

The *right blocks* of a bipartition `x` are just the intersections of the blocks of `x` with `[-n .. -1]` where `n` is the degree of `x`, the values in transverse blocks are positive, and the values in non-transverse blocks are negative.

The right blocks of a bipartition are GAP objects in their own right, and are not simply a list of blocks of `x`; see 3.6 for more information.

The significance of this notion lies in the fact that bipartitions `x` and `y` are $\mathscr{L}$-related in the partition monoid if and only if they have equal right blocks.

Example
```
gap> x := Bipartition([[1, 4, 7, 8, -4], [2, 3, 5, -2, -7],
>                      [6, -1], [-3], [-5, -6, -8]]);;
gap> RightBlocks(x);
<blocks: [ 1* ], [ 2*, 7* ], [ 3 ], [ 4* ], [ 5, 6, 8 ]>
gap> LeftBlocks(x);
<blocks: [ 1*, 4*, 7*, 8* ], [ 2*, 3*, 5* ], [ 6* ]>
```

3.5.6 LeftBlocks

▷ `LeftBlocks(x)` (attribute)

Returns: The left blocks of a bipartition.

`LeftBlocks` returns the left blocks of the bipartition `x`.

The *left blocks* of a bipartition `x` are just the intersections of the blocks of `x` with `[1..n]` where `n` is the degree of `x`, the values in transverse blocks are positive, and the values in non-transverse blocks are negative.

The left blocks of a bipartition are GAP objects in their own right, and are not simply a list of blocks of `x`; see 3.6 for more information.

The significance of this notion lies in the fact that bipartitions `x` and y are $\mathscr{R}$-related in the partition monoid if and only if they have equal left blocks.

Example
```
gap> x := Bipartition([[1, 4, 7, 8, -4], [2, 3, 5, -2, -7],
>                      [6, -1], [-3], [-5, -6, -8]]);;
gap> RightBlocks(x);
<blocks: [ 1* ], [ 2*, 7* ], [ 3 ], [ 4* ], [ 5, 6, 8 ]>
gap> LeftBlocks(x);
<blocks: [ 1*, 4*, 7*, 8* ], [ 2*, 3*, 5* ], [ 6* ]>
```

3.5.7 NrLeftBlocks

▷ `NrLeftBlocks(x)` (attribute)

Returns: A non-negative integer.

When the argument is a bipartition `x`, `NrLeftBlocks` returns the number of left blocks of `x`, i.e. the number of blocks of `x` intersecting `[1 .. n]` non-trivially.

Example
```
gap> x := Bipartition([[1, 2, 3, 4, 5, 6, 8], [7, -2, -3],
>                      [-1, -4, -7, -8], [-5, -6]]);;
gap> NrLeftBlocks(x);
2
gap> LeftBlocks(x);
<blocks: [ 1, 2, 3, 4, 5, 6, 8 ], [ 7* ]>
```

3.5.8 NrRightBlocks

▷ `NrRightBlocks(x)` (attribute)

Returns: A non-negative integer.

When the argument is a bipartition `x`, `NrRightBlocks` returns the number of right blocks of `x`, i.e. the number of blocks of `x` intersecting `[-n .. -1]` non-trivially.

Example
```
gap> x := Bipartition([[1, 2, 3, 4, 6, -2, -7], [5, -1, -3, -8],
>                      [7, -4, -6], [8], [-5]]);;
gap> RightBlocks(x);
<blocks: [ 1*, 3*, 8* ], [ 2*, 7* ], [ 4*, 6* ], [ 5 ]>
gap> NrRightBlocks(x);
4
```

3.5.9 NrBlocks (for blocks)

▷ `NrBlocks(blocks)` (attribute)
▷ `NrBlocks(f)` (attribute)

Returns: A positive integer.

If *blocks* is some blocks or *f* is a bipartition, then `NrBlocks` returns the number of blocks in *blocks* or *f*, respectively.

Example
```
gap> blocks := BlocksNC([[-1, -2, -3, -4], [-5], [6]]);
<blocks: [ 1, 2, 3, 4 ], [ 5 ], [ 6* ]>
gap> NrBlocks(blocks);
3
gap> x := Bipartition([
>   [1, 5], [2, 4, -2, -4], [3, 6, -1, -5, -6], [-3]]);
<bipartition: [ 1, 5 ], [ 2, 4, -2, -4 ], [ 3, 6, -1, -5, -6 ],
 [ -3 ]>
gap> NrBlocks(x);
4
```

3.5.10 DomainOfBipartition

▷ `DomainOfBipartition(x)` (attribute)

Returns: A list of positive integers.

If *x* is a bipartition, then `DomainOfBipartition` returns the domain of *x*. The *domain* of *x* consists of those numbers `i` in `[1 .. n]` such that `i` is contained in a transverse block of *x*, where `n` is the degree of *x* (see `DegreeOfBipartition` (3.5.1)).

Example
```
gap> x := Bipartition([[1, 2], [3, 4, 5, -5], [6, -6],
>                      [-1, -2, -3], [-4]]);
<bipartition: [ 1, 2 ], [ 3, 4, 5, -5 ], [ 6, -6 ], [ -1, -2, -3 ],
 [ -4 ]>
gap> DomainOfBipartition(x);
[ 3, 4, 5, 6 ]
```

3.5.11 CodomainOfBipartition

▷ `CodomainOfBipartition(x)` (attribute)

Returns: A list of positive integers.

If *x* is a bipartition, then `CodomainOfBipartition` returns the codomain of *x*. The *codomain* of *x* consists of those numbers `i` in `[-n .. -1]` such that `i` is contained in a transverse block of *x*, where `n` is the degree of *x* (see `DegreeOfBipartition` (3.5.1)).

Example
```
gap> x := Bipartition([[1, 2], [3, 4, 5, -5], [6, -6],
>                     [-1, -2, -3], [-4]]);
<bipartition: [ 1, 2 ], [ 3, 4, 5, -5 ], [ 6, -6 ], [ -1, -2, -3 ],
 [ -4 ]>
gap> CodomainOfBipartition(x);
[ -5, -6 ]
```

3.5.12 IsTransBipartition

▷ IsTransBipartition(*x*) (property)

Returns: `true` or `false`.

If the bipartition *x* defines a transformation, then `IsTransBipartition` returns `true`, and if not, then `false` is returned.

A bipartition *x* defines a transformation if and only if the number of left blocks equals the number of transverse blocks and the number of right blocks equals the degree.

Example
```
gap> x := Bipartition([[1, 4, -2], [2, 5, -6], [3, -7],
>                     [6, 7, -9], [8, 9, -1], [10, -5],
>                     [-3], [-4], [-8], [-10]]);;
gap> IsTransBipartition(x);
true
gap> x := Bipartition([[1, 4, -3, -6], [2, 5, -4, -5],
>                     [3, 6, -1], [-2]]);;
gap> IsTransBipartition(x);
false
gap> Number(PartitionMonoid(3), IsTransBipartition);
27
```

3.5.13 IsDualTransBipartition

▷ IsDualTransBipartition(*x*) (property)

Returns: `true` or `false`.

If the star of the bipartition *x* defines a transformation, then `IsDualTransBipartition` returns `true`, and if not, then `false` is returned.

A bipartition is the dual of a transformation if and only if its number of right blocks equals its number of transverse blocks and its number of left blocks equals its degree.

Example
```
gap> x := Bipartition([[1, -8, -9], [2, -1, -4], [3],
>                     [4], [5, -10], [6, -2, -5], [7, -3],
>                     [8], [9, -6, -7], [10]]);;
gap> IsDualTransBipartition(x);
true
gap> x := Bipartition([[1, 4, -3, -6], [2, 5, -4, -5],
>                     [3, 6, -1], [-2]]);;
gap> IsTransBipartition(x);
false
gap> Number(PartitionMonoid(3), IsDualTransBipartition);
27
```

3.5.14 IsPermBipartition

▷ `IsPermBipartition(x)` (property)

Returns: `true` or `false`.

If the bipartition *x* defines a permutation, then `IsPermBipartition` returns `true`, and if not, then `false` is returned.

A bipartition is a permutation if its numbers of left, right, and transverse blocks all equal its degree.

Example

```
gap> x := Bipartition([
>   [1, 4, -1], [2, -3], [3, 6, -5], [5, -2, -4, -6]]);;
gap> IsPermBipartition(x);
false
gap> x := Bipartition([[1, -3], [2, -4], [3, -6], [4, -1],
>                      [5, -5], [6, -2], [7, -8], [8, -7]]);;
gap> IsPermBipartition(x);
true
```

3.5.15 IsPartialPermBipartition

▷ `IsPartialPermBipartition(x)` (property)

Returns: `true` or `false`.

If the bipartition *x* defines a partial permutation, then `IsPartialPermBipartition` returns `true`, and if not, then `false` is returned.

A bipartition *x* defines a partial permutation if and only if the numbers of left and right blocks of *x* equal the degree of *x*.

Example

```
gap> x := Bipartition([
>   [1, 4, -1], [2, -3], [3, 6, -5], [5, -2, -4, -6]]);;
gap> IsPartialPermBipartition(x);
false
gap> x := Bipartition([[1, -3], [2], [-4], [3, -6], [4, -1],
>                      [5, -5], [6, -2], [7, -8], [8, -7]]);;
gap> IsPermBipartition(x);
false
gap> IsPartialPermBipartition(x);
true
```

3.5.16 IsBlockBijection

▷ `IsBlockBijection(x)` (property)

Returns: `true` or `false`.

If the bipartition *x* induces a bijection from the quotient of `[1 .. n]` by the blocks of *f* to the quotient of `[-n .. -1]` by the blocks of *f*, then `IsBlockBijection` return `true`, and if not, then it returns `false`.

A bipartition is a block bijection if and only if its number of blocks, left blocks and right blocks are equal.

Example

```
gap> x := Bipartition([[1, 4, 5, -2], [2, 3, -1], [6, -5, -6],
>                      [-3, -4]]);;
gap> IsBlockBijection(x);
```

```
false
gap> x := Bipartition([[1, 2, -3], [3, -1, -2], [4, -4], [5, -5]]);;
gap> IsBlockBijection(x);
true
```

3.5.17 IsUniformBlockBijection

▷ IsUniformBlockBijection(*x*) (property)

Returns: true or false.

If the bipartition *x* is a block bijection where every block contains an equal number of positive and negative entries, then IsUniformBlockBijection returns true, and otherwise it returns false.

Example
```
gap> x := Bipartition([[1, 2, -3, -4], [3, -5], [4, -6],
> [5, -7], [6, -8], [7, -9], [8, -1], [9, -2]]);;
gap> IsBlockBijection(x);
true
gap> x := Bipartition([[1, 2, -3], [3, -1, -2], [4, -4],
> [5, -5]]);;
gap> IsUniformBlockBijection(x);
false
```

3.5.18 CanonicalBlocks

▷ CanonicalBlocks(*blocks*) (attribute)

Returns: Blocks of a bipartition.

If *blocks* is the blocks of a bipartition, then the function CanonicalBlocks returns a canonical representative of *blocks*.

In particular, let C(n) be a largest class such that any element of C(n) is blocks of a bipartition of degree n and such that for every pair of elements x and y of C(n) the number of signed, and similarly unsigned, blocks of any given size in both x and y are the same. Then CanonicalBlocks returns a canonical representative of a class C(n) containing *blocks* where n is the degree of *blocks*.

Example
```
gap> B := BlocksNC([[-1, -3], [2, 4, 7], [5, 6]]);
<blocks: [ 1, 3 ], [ 2*, 4*, 7* ], [ 5*, 6* ]>
gap> CanonicalBlocks(B);
<blocks: [ 1*, 2*, 3* ], [ 4, 5 ], [ 6*, 7* ]>
```

3.6 Creating blocks and their attributes

As described above the left and right blocks of a bipartition characterise Green's $\mathscr{R}$- and $\mathscr{L}$-relation of the partition monoid; see LeftBlocks (3.5.6) and RightBlocks (3.5.5). The left or right blocks of a bipartition are GAP objects in their own right.

In this section, we describe the functions in the Semigroups package for creating and manipulating the left or right blocks of a bipartition.

3.6.1 IsBlocks

▷ `IsBlocks(obj)` (Category)

Returns: `true` or `false`.

Every blocks object in GAP belongs to the category `IsBlocks`. Basic operations for blocks are `ExtRepOfObj` (3.6.3), `RankOfBlocks` (3.6.4), `DegreeOfBlocks` (3.6.5), `OnRightBlocks` (3.7.1), and `OnLeftBlocks` (3.7.2).

3.6.2 BlocksNC

▷ `BlocksNC(classes)` (function)

Returns: A blocks.

This function makes it possible to create a GAP object corresponding to the left or right blocks of a bipartition without reference to any bipartitions.

`BlocksNC` returns the blocks with equivalence classes *classes*, which should be a list of duplicate-free lists consisting solely of positive or negative integers, where the union of the absolute values of the lists is `[1 .. n]` for some `n`. The blocks with positive entries correspond to transverse blocks and the classes with negative entries correspond to non-transverse blocks.

This method function does not check that its arguments are valid, and should be used with caution.

Example
```
gap> BlocksNC([[1], [2], [-3, -6], [-4, -5]]);
<blocks: [ 1* ], [ 2* ], [ 3, 6 ], [ 4, 5 ]>
```

3.6.3 ExtRepOfObj (for a blocks)

▷ `ExtRepOfObj(blocks)` (operation)

Returns: A list of integers.

If `n` is the degree of a bipartition with left or right blocks *blocks*, then `ExtRepOfObj` returns the partition corresponding to *blocks* as a sorted list of duplicate-free lists.

Example
```
gap> blocks := BlocksNC([[1, 6], [2, 3, 7], [4, 5], [-8]]);;
gap> ExtRepOfObj(blocks);
[ [ 1, 6 ], [ 2, 3, 7 ], [ 4, 5 ], [ -8 ] ]
```

3.6.4 RankOfBlocks

▷ `RankOfBlocks(blocks)` (attribute)
▷ `NrTransverseBlocks(blocks)` (attribute)

Returns: A non-negative integer.

When the argument *blocks* is the left or right blocks of a bipartition, `RankOfBlocks` returns the number of blocks of *blocks* containing only positive entries, i.e. the number of transverse blocks in *blocks*.

`NrTransverseBlocks` is a synonym of `RankOfBlocks` in this context.

Example
```
gap> blocks := BlocksNC([[-1, -2, -4, -6], [3, 10, 12], [5, 7],
>                        [8], [9], [-11]]);;
gap> RankOfBlocks(blocks);
4
```

3.6.5 DegreeOfBlocks

▷ DegreeOfBlocks(*blocks*) (attribute)

Returns: A non-negative integer.

The degree of *blocks* is the number of points n where it is defined, i.e. the union of the blocks in *blocks* will be [1 .. n] after taking the absolute value of every element.

Example
```
gap> blocks := BlocksNC([[-1, -11], [2], [3, 5, 6, 7], [4, 8], [9, 10],
>                        [12]]);;
gap> DegreeOfBlocks(blocks);
12
```

3.6.6 ProjectionFromBlocks

▷ ProjectionFromBlocks(*blocks*) (attribute)

Returns: A bipartition.

When the argument *blocks* is the left or right blocks of a bipartition, this operation returns the unique bipartition whose left and right blocks are equal to *blocks*.

If *blocks* is the left blocks of a bipartition x, then this operation returns a bipartition equal to the left projection of x. The analogous statement holds when *blocks* is the right blocks of a bipartition.

Example
```
gap> x := Bipartition([[1], [2, -2, -3], [3], [-1]]);
<bipartition: [ 1 ], [ 2, -2, -3 ], [ 3 ], [ -1 ]>
gap> ProjectionFromBlocks(LeftBlocks(x));
<bipartition: [ 1 ], [ 2, -2 ], [ 3 ], [ -1 ], [ -3 ]>
gap> LeftProjection(x);
<bipartition: [ 1 ], [ 2, -2 ], [ 3 ], [ -1 ], [ -3 ]>
gap> ProjectionFromBlocks(RightBlocks(x));
<bipartition: [ 1 ], [ 2, 3, -2, -3 ], [ -1 ]>
gap> RightProjection(x);
<bipartition: [ 1 ], [ 2, 3, -2, -3 ], [ -1 ]>
```

3.7 Actions on blocks

Bipartitions act on left and right blocks in several ways, which are described in this section.

3.7.1 OnRightBlocks

▷ OnRightBlocks(*blocks*, *x*) (operation)

Returns: The blocks of a bipartition.

OnRightBlocks returns the right blocks of the product g * *x* where g is any bipartition whose right blocks are equal to *blocks*.

Example
```
gap> x := Bipartition([[1, 4, 5, 8], [2, 3, 7], [6, -3, -4, -5],
>                      [-1, -2, -6], [-7, -8]]);;
gap> y := Bipartition([[1, 5], [2, 4, 8, -2], [3, 6, 7, -3, -4],
>                      [-1, -6, -8], [-5, -7]]);;
gap> RightBlocks(y * x);
<blocks: [ 1, 2, 6 ], [ 3*, 4*, 5* ], [ 7, 8 ]>
```

```
gap> OnRightBlocks(RightBlocks(y), x);
<blocks: [ 1, 2, 6 ], [ 3*, 4*, 5* ], [ 7, 8 ]>
```

3.7.2 OnLeftBlocks

▷ `OnLeftBlocks(blocks, x)` (operation)

Returns: The blocks of a bipartition.

`OnLeftBlocks` returns the left blocks of the product `x * y` where `y` is any bipartition whose left blocks are equal to `blocks`.

Example
```
gap> x := Bipartition([[1, 5, 7, -1, -3, -4, -6], [2, 3, 6, 8],
>                      [4, -2, -5, -8], [-7]]);;
gap> y := Bipartition([[1, 3, -4, -5], [2, 4, 5, 8], [6, -1, -3],
>                      [7, -2, -6, -7, -8]]);;
gap> LeftBlocks(x * y);
<blocks: [ 1*, 4*, 5*, 7* ], [ 2, 3, 6, 8 ]>
gap> OnLeftBlocks(LeftBlocks(y), x);
<blocks: [ 1*, 4*, 5*, 7* ], [ 2, 3, 6, 8 ]>
```

3.8 Semigroups of bipartitions

Semigroups and monoids of bipartitions can be created in the usual way in GAP using the functions `Semigroup` (**Reference: Semigroup**) and `Monoid` (**Reference: Monoid**); see Chapter 6 for more details.

It is possible to create inverse semigroups and monoids of bipartitions using `InverseSemigroup` (**Reference: InverseSemigroup**) and `InverseMonoid` (**Reference: InverseMonoid**) when the argument is a collection of block bijections or partial perm bipartions; see `IsBlockBijection` (3.5.16) and `IsPartialPermBipartition` (3.5.15). Note that every bipartition semigroup in Semigroups is finite.

3.8.1 IsBipartitionSemigroup

▷ `IsBipartitionSemigroup(S)` (filter)

▷ `IsBipartitionMonoid(S)` (filter)

Returns: `true` or `false`.

A *bipartition semigroup* is simply a semigroup consisting of bipartitions. An object `obj` is a bipartition semigroup in GAP if it satisfies `IsSemigroup` (**Reference: IsSemigroup**) and `IsBipartitionCollection` (3.1.2).

A *bipartition monoid* is a monoid consisting of bipartitions. An object `obj` is a bipartition monoid in GAP if it satisfies `IsMonoid` (**Reference: IsMonoid**) and `IsBipartitionCollection` (3.1.2).

Note that it is possible for a bipartition semigroup to have a multiplicative neutral element (i.e. an identity element) but not to satisfy `IsBipartitionMonoid`. For example,

Example
```
gap> x := Bipartition([
> [1, 4, -2], [2, 5, -6], [3, -7], [6, 7, -9], [8, 9, -1],
> [10, -5], [-3], [-4], [-8], [-10]]);;
gap> S := Semigroup(x, One(x));
<commutative bipartition monoid of degree 10 with 1 generator>
```

```
gap> IsMonoid(S);
true
gap> IsBipartitionMonoid(S);
true
gap> S := Semigroup([
> Bipartition([
>   [1, -3], [2, -8], [3, 8, -1], [4, -4], [5, -5], [6, -6],
>   [7, -7], [9, 10, -10], [-2], [-9]]),
> Bipartition([
>   [1, -1], [2, -2], [3, -3], [4, -4], [5, -5], [6, -6],
>   [7, -7], [8, -8], [9, 10, -10], [-9]])]);;
gap> One(S);
fail
gap> MultiplicativeNeutralElement(S);
<bipartition: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ], [ 4, -4 ], [ 5, -5 ],
 [ 6, -6 ], [ 7, -7 ], [ 8, -8 ], [ 9, 10, -10 ], [ -9 ]>
gap> IsMonoid(S);
false
```

In this example `S` cannot be converted into a monoid using `AsMonoid` (**Reference: AsMonoid**) since the `One` (**Reference: One**) of any element in `S` differs from the multiplicative neutral element.

For more details see `IsMagmaWithOne` (**Reference: IsMagmaWithOne**).

3.8.2 IsBlockBijectionSemigroup

▷ `IsBlockBijectionSemigroup(S)` (property)

▷ `IsBlockBijectionMonoid(S)` (filter)

Returns: `true` or `false`.

A *block bijection semigroup* is simply a semigroup consisting of block bijections. A *block bijection monoid* is a monoid consisting of block bijections.

An object in GAP is a block bijection monoid if it satisfies `IsMonoid` (**Reference: IsMonoid**) and `IsBlockBijectionSemigroup`.

See `IsBlockBijection` (3.5.16).

3.8.3 IsPartialPermBipartitionSemigroup

▷ `IsPartialPermBipartitionSemigroup(S)` (property)

▷ `IsPartialPermBipartitionMonoid(S)` (filter)

Returns: `true` or `false`.

A *partial perm bipartition semigroup* is simply a semigroup consisting of partial perm bipartitions. A *partial perm bipartition monoid* is a monoid consisting of partial perm bipartitions.

An object in GAP is a partial perm bipartition monoid if it satisfies `IsMonoid` (**Reference: IsMonoid**) and `IsPartialPermBipartitionSemigroup`.

See `IsPartialPermBipartition` (3.5.15).

3.8.4 IsPermBipartitionGroup

▷ `IsPermBipartitionGroup(S)` (property)

Returns: `true` or `false`.

A *perm bipartition group* is simply a semigroup consisting of perm bipartitions.
See `IsPermBipartition` (3.5.14).

3.8.5 DegreeOfBipartitionSemigroup

▷ `DegreeOfBipartitionSemigroup(S)` (attribute)

Returns: A non-negative integer.

The *degree* of a bipartition semigroup *S* is just the degree of any (and every) element of *S*.

Example
```
gap> DegreeOfBipartitionSemigroup(JonesMonoid(8));
8
```

Chapter 4

Partitioned binary relations (PBRs)

In this chapter we describe the functions in Semigroups for creating and manipulating partitioned binary relations, henceforth referred to by their acronym PBRs. We begin by describing what these objects are.

PBRs were introduced in the paper [MM11] as, roughly speaking, the maximum generalization of bipartitions and related objects. Although, mathematically, bipartitions are a special type of PBR, in Semigroups bipartitions and PBRs are currently distinct types of objects. It is possible to change the representation from bipartition to PBR, and from PBR to bipartition, when appropriate; see Section 4.3 for more details. The reason for this distinct is largely historical, bipartition appeared in the literature, and in the Semigroups package, before PBRs.

4.1 The family and categories of PBRs

4.1.1 IsPBR

▷ `IsPBR(obj)` (Category)

Returns: `true` or `false`.

Every PBR in GAP belongs to the category `IsPBR`. Basic operations for PBRs are `DegreeOfPBR` (4.5.2), `ExtRepOfObj` (4.5.3), `PBRNumber` (4.5.4), `NumberPBR` (4.5.4), `StarOp` (4.5.1), and multiplication of two PBRs of equal degree is via `*`.

4.1.2 IsPBRCollection

▷ `IsPBRCollection(obj)` (Category)

▷ `IsPBRCollColl(obj)` (Category)

Returns: `true` or `false`.

Every collection of PBRs belongs to the category `IsPBRCollection`. For example, PBR semigroups belong to `IsPBRCollection`.

Every collection of collections of PBRs belongs to `IsPBRCollColl`. For example, a list of PBR semigroups belongs to `IsPBRCollColl`.

4.2 Creating PBRs

There are several ways of creating PBRs in GAP, which are described in this section.

4.2.1 PBR

▷ `PBR(left, right)` (operation)

Returns: A PBR.

The arguments *left* and *right* of this function should each be a list of length `n` whose entries are lists of integers in the ranges `[-n .. -1]` and `[1 .. n]` for some `n` greater than 0.

Given such an argument, `PBR` returns the PBR `x` where:

- for each `i` in the range `[1 .. n]` there is an edge from `i` to every `j` in *left[i]*;
- for each `i` in the range `[-n .. -1]` there is an edge from `i` to every `j` in *right[-i]*;

`PBR` returns an error if the argument does not define a PBR.

Example
```
gap> PBR([[-3, -2, -1, 2, 3], [-1], [-3, -2, 1, 2]],
>        [[-2, -1, 1, 2, 3], [3], [-3, -2, -1, 1, 3]]);
PBR([ [ -3, -2, -1, 2, 3 ], [ -1 ], [ -3, -2, 1, 2 ] ],
  [ [ -2, -1, 1, 2, 3 ], [ 3 ], [ -3, -2, -1, 1, 3 ] ])
```

4.2.2 RandomPBR

▷ `RandomPBR(n[, p])` (operation)

Returns: A PBR.

If *n* is a positive integer and *p* is an float between `0` and `1`, then `RandomPBR` returns a random PBR of degree *n* where the probability of there being an edge from `i` to `j` is approximately `p`.

If the optional second argument is not present, then a random value *p* is used (chosen with uniform probability).

Example
```
gap> RandomPBR(6);
PBR(
  [ [ -5, 1, 2, 3 ], [ -6, -3, -1, 2, 5 ], [ -5, -2, 2, 3, 5 ],
      [ -6, -4, -1, 2, 3, 6 ], [ -4, -1, 2, 4 ],
      [ -5, -3, -1, 1, 2, 3, 5 ] ],
  [ [ -6, -4, -2, 1, 3, 5, 6 ], [ -5, -2, 1, 2, 3, 5 ],
      [ -6, -5, -2, 1, 5 ], [ -6, -5, -3, -2, 1, 3, 4 ],
      [ -6, -5, -4, -2, 3, 5 ], [ -6, -4, -2, -1, 1, 2, 6 ] ])
```

4.2.3 EmptyPBR

▷ `EmptyPBR(n)` (operation)

Returns: A PBR.

If *n* is a positive integer, then `EmptyPBR` returns the PBR of degree *n* with no edges.

Example
```
gap> x := EmptyPBR(3);
PBR([ [  ], [  ], [  ] ], [ [  ], [  ], [  ] ])
gap> IsEmptyPBR(x);
true
```

4.2.4 IdentityPBR

▷ `IdentityPBR(n)` (operation)

Returns: A PBR.

If `n` is a positive integer, then `IdentityPBR` returns the identity PBR of degree `n`. This PBR has $2n$ edges: specifically, for each `i` in the ranges `[1 .. n]` and `[-n .. -1]`, the identity PBR has an edge from `i` to `-i`.

Example
```
gap> x := IdentityPBR(3);
PBR([ [ -1 ], [ -2 ], [ -3 ] ], [ [ 1 ], [ 2 ], [ 3 ] ])
gap> IsIdentityPBR(x);
true
```

4.2.5 UniversalPBR

▷ `UniversalPBR(n)` (operation)

Returns: A PBR.

If `n` is a positive integer, then `UniversalPBR` returns the PBR of degree `n` with `4 * n ^ 2` edges, i.e. every possible edge.

Example
```
gap> x := UniversalPBR(2);
PBR([ [ -2, -1, 1, 2 ], [ -2, -1, 1, 2 ] ],
  [ [ -2, -1, 1, 2 ], [ -2, -1, 1, 2 ] ])
gap> IsUniversalPBR(x);
true
```

4.3 Changing the representation of a PBR

It is possible that a PBR can be represented as another type of object, or that another type of GAP object can be represented as a PBR. In this section, we describe the functions in the Semigroups package for changing the representation of PBR, or for changing the representation of another type of object to that of a PBR.

The operations `AsPermutation` (4.3.4), `AsPartialPerm` (4.3.3), `AsTransformation` (4.3.2), `AsBipartition` (3.3.1), `AsBooleanMat` (5.3.2) can be used to convert PBRs into permutations, partial permutations, transformations, bipartitions, and boolean matrices where appropriate.

4.3.1 AsPBR

▷ `AsPBR(x[, n])` (operation)

Returns: A PBR.

`AsPBR` returns the boolean matrix, bipartition, transformation, partial permutation, or permutation `x` as a PBR of degree `n`.

There are several possible arguments for `AsPBR`:

bipartitions

If `x` is a bipartition and `n` is a positive integer, then `AsPBR` returns a PBR corresponding to `x` with degree `n`. The resulting PBR has an edge from `i` to `j` whenever `i` and `j` belong to the same block of `x`.

If the optional second argument *n* is not specified, then degree of the bipartition *x* is used by default.

boolean matrices

If *x* is a boolean matrix of even dimension `2 * m` and *n* is a positive integer, then `AsPBR` returns a PBR corresponding to *x* with degree *n*. If the optional second argument *n* is not specified, then dimension of the boolean matrix *x* is used by default.

transformations, partial perms, permutations

If *x* is a transformation, partial perm, or permutation and *n* is a positive integer, then `AsPBR` is a synonym for `AsPBR(AsBipartition(x, n))`. If the optional second argument *n* is not specified, then `AsPBR` is a synonym for `AsPBR(AsBipartition(x))`. See `AsBipartition` (3.3.1) for more details.

Example

```
gap> x := Bipartition([[1, 2, -1], [3, -2], [4, -3, -4]]);
<block bijection: [ 1, 2, -1 ], [ 3, -2 ], [ 4, -3, -4 ]>
gap> AsPBR(x, 2);
PBR([ [ -1, 1, 2 ], [ -1, 1, 2 ] ], [ [ -1, 1, 2 ], [ -2 ] ])
gap> AsPBR(x, 5);
PBR([ [ -1, 1, 2 ], [ -1, 1, 2 ], [ -2, 3 ], [ -4, -3, 4 ], [  ] ],
  [ [ -1, 1, 2 ], [ -2, 3 ], [ -4, -3, 4 ], [ -4, -3, 4 ], [  ] ])
gap> AsPBR(x);
PBR([ [ -1, 1, 2 ], [ -1, 1, 2 ], [ -2, 3 ], [ -4, -3, 4 ] ],
  [ [ -1, 1, 2 ], [ -2, 3 ], [ -4, -3, 4 ], [ -4, -3, 4 ] ])
gap> mat := Matrix(IsBooleanMat, [[1, 0, 0, 1],
>                                 [0, 1, 1, 0],
>                                 [1, 0, 1, 1],
>                                 [0, 0, 0, 1]]);;
gap> AsPBR(mat);
PBR([ [ -2, 1 ], [ -1, 2 ] ], [ [ -2, -1, 1 ], [ -2 ] ])
gap> AsPBR(mat, 2);
PBR([ [ 1 ] ], [ [ -1 ] ])
gap> AsPBR(mat, 6);
PBR([ [ -2, 1 ], [ -1, 2 ], [  ] ], [ [ -2, -1, 1 ], [ -2 ], [  ] ])
gap> x := Transformation([2, 2, 1]);;
gap> AsPBR(x);
PBR([ [ -2 ], [ -2 ], [ -1 ] ], [ [ 3 ], [ 1, 2 ], [  ] ])
gap> AsPBR(x, 2);
PBR([ [ -2 ], [ -2 ] ], [ [  ], [ 1, 2 ] ])
gap> AsPBR(x, 4);
PBR([ [ -2 ], [ -2 ], [ -1 ], [ -4 ] ],
  [ [ 3 ], [ 1, 2 ], [  ], [ 4 ] ])
gap> x := PartialPerm([4, 3]);
[1,4][2,3]
gap> AsPBR(x);
PBR([ [ -4 ], [ -3 ], [  ], [  ] ], [ [  ], [  ], [ 2 ], [ 1 ] ])
gap> AsPBR(x, 2);
PBR([ [  ], [  ] ], [ [  ], [  ] ])
gap> AsPBR(x, 5);
PBR([ [ -4 ], [ -3 ], [  ], [  ], [  ] ],
  [ [  ], [  ], [ 2 ], [ 1 ], [  ] ])
gap> x := (1, 3)(2, 4);
```

```
(1,3)(2,4)
gap> AsPBR(x);
PBR([ [ -3, 1 ], [ -4, 2 ], [ -1, 3 ], [ -2, 4 ] ],
  [ [ -1, 3 ], [ -2, 4 ], [ -3, 1 ], [ -4, 2 ] ])
gap> AsPBR(x, 5);
PBR([ [ -3, 1 ], [ -4, 2 ], [ -1, 3 ], [ -2, 4 ], [ -5, 5 ] ],
  [ [ -1, 3 ], [ -2, 4 ], [ -3, 1 ], [ -4, 2 ], [ -5, 5 ] ])
```

4.3.2 AsTransformation (for a PBR)

▷ `AsTransformation(x)` (attribute)

Returns: A transformation.

When the argument `x` is a PBR which satisfies `IsTransformationPBR` (4.5.9), then this attribute returns that transformation.

Example
```
gap> x := PBR([[-3], [-3], [-2]], [[], [3], [1, 2]]);;
gap> IsTransformationPBR(x);
true
gap> AsTransformation(x);
Transformation( [ 3, 3, 2 ] )
gap> x := PBR([[1], [1, 2]], [[-2, -1], [-2, -1]]);;
gap> AsTransformation(x);
Error, Semigroups: AsTransformation: usage,
the argument <x> must be a transformation PBR,
```

4.3.3 AsPartialPerm (for a PBR)

▷ `AsPartialPerm(x)` (operation)

Returns: A partial perm.

When the argument `x` is a PBR which satisfies `IsPartialPermPBR` (4.5.11), then this function returns that partial perm.

Example
```
gap> x := PBR([[-1, 1], [-3, 2], [-4, 3], [4], [5]],
>             [[-1, 1], [-2], [-3, 2], [-4, 3], [-5]]);;
gap> IsPartialPermPBR(x);
true
gap> AsPartialPerm(x);
[2,3,4](1)
```

4.3.4 AsPermutation (for a PBR)

▷ `AsPermutation(x)` (attribute)

Returns: A permutation.

When the argument `x` is a PBR which satisfies `IsPermPBR` (4.5.12), then this attribute returns that permutation.

Example
```
gap> x := PBR([[-1, 1], [-4, 2], [-2, 3], [-3, 4]],
>             [[-1, 1], [-2, 3], [-3, 4], [-4, 2]]);;
gap> IsPermPBR(x);
true
```

```
gap> AsPermutation(x);
(2,4,3)
```

4.4 Operators for PBRs

`x * y`
: returns the product of `x` and `y` when `x` and `y` are PBRs.

`x < y`
: returns `true` if the degree of `x` is less than the degree of `y`, or the degrees are equal and the out-neighbours of `x` (as a list of list of positive integers) is lexicographically less than the out-neighbours of `y`.

`x = y`
: returns `true` if the PBR `x` equals the PBR `y` and returns `false` if it does not.

4.5 Attributes for PBRs

In this section we describe various attributes that a PBR can possess.

4.5.1 StarOp (for a PBR)

▷ `StarOp(x)` (operation)
▷ `Star(x)` (attribute)

Returns: A PBR.

`StarOp` returns the unique PBR `y` obtained by exchanging the positive and negative numbers in `x` (i.e. multiplying `ExtRepOfObj` (4.5.3) by `-1` and swapping its first and second components).

Example
```
gap> x := PBR([[], [-1], []], [[-3, -2, 2, 3], [-2, 1], []]);;
gap> Star(x);
PBR([ [ -3, -2, 2, 3 ], [ -1, 2 ], [  ] ], [ [  ], [ 1 ], [  ] ])
```

4.5.2 DegreeOfPBR

▷ `DegreeOfPBR(x)` (attribute)
▷ `DegreeOfPBRCollection(x)` (attribute)

Returns: A positive integer.

The degree of a PBR is, roughly speaking, the number of points where it is defined. More precisely, if `x` is a PBR defined on `2 * n` points, then the degree of `x` is `n`.

The degree of a collection `coll` of PBRs of equal degree is just the degree of any (and every) PBR in `coll`. The degree of collection of PBRs of unequal degrees is not defined.

Example
```
gap> x := PBR([[-2], [-2, -1, 2, 3], [-1, 1, 2, 3]],
>             [[-1, 1], [2, 3], [-3, 2, 3]]);
PBR([ [ -2 ], [ -2, -1, 2, 3 ], [ -1, 1, 2, 3 ] ],
  [ [ -1, 1 ], [ 2, 3 ], [ -3, 2, 3 ] ])
gap> DegreeOfPBR(x);
3
```

```
gap> S := FullPBRMonoid(2);
<pbr monoid of degree 2 with 10 generators>
gap> DegreeOfPBRCollection(S);
2
```

4.5.3 ExtRepOfObj (for a PBR)

▷ ExtRepOfObj(*x*) (operation)

Returns: A pair of lists of lists of integers.

If n is the degree of the PBR *x*, then ExtRepOfObj returns the argument required by PBR (4.2.1) to create a PBR equal to *x*, i.e. PBR(ExtRepOfObj(*x*)) returns a PBR equal to *x*.

Example

```
gap> x := PBR([[-1, 1], [-2, 2]],
>             [[-2, -1, 1], [-1, 1, 2]]);
PBR([ [ -1, 1 ], [ -2, 2 ] ], [ [ -2, -1, 1 ], [ -1, 1, 2 ] ])
gap> ExtRepOfObj(x);
[ [ [ -1, 1 ], [ -2, 2 ] ], [ [ -2, -1, 1 ], [ -1, 1, 2 ] ] ]
```

4.5.4 PBRNumber

▷ PBRNumber(*m*, *n*) (operation)

▷ NumberPBR(*mat*) (operation)

Returns: A PBR, or a positive integer.

These functions implement a bijection from the set of all PBRs of degree *n* and the numbers [1 .. 2 ^ (4 * *n* ^ 2)].

More precisely, if *m* and *n* are positive integers such that *m* is at most 2 ^ (4 * *n* ^ 2), then PBRNumber returns the *m*th PBR of degree *n*.

If *mat* is a PBR of degree *n*, then NumberPBR returns the number in [1 .. 2 ^ (4 * *n* ^ 2)] that corresponds to *mat*.

Example

```
gap> S := FullPBRMonoid(1);
<pbr monoid of degree 1 with 4 generators>
gap> List(S, NumberPBR);
[ 3, 15, 5, 7, 8, 1, 4, 11, 13, 16, 6, 2, 9, 12, 14, 10 ]
```

4.5.5 IsEmptyPBR

▷ IsEmptyPBR(*x*) (property)

Returns: true or false.

A PBR is EMPTY if it has no edges. IsEmptyPBR returns true if the PBR *x* is empty and false if it is not.

Example

```
gap> x := PBR([[]], [[]]);;
gap> IsEmptyPBR(x);
true
gap> x := PBR([[-2, 1], [2]], [[-1], [-2, 1]]);
PBR([ [ -2, 1 ], [ 2 ] ], [ [ -1 ], [ -2, 1 ] ])
gap> IsEmptyPBR(x);
false
```

4.5.6 IsIdentityPBR

▷ IsIdentityPBR(x) (property)

Returns: true or false.

A PBR of degree n is the IDENTITY PBR of degree n if it is the identity of the full PBR monoid of degree n. The identity PBR of degree n has 2n edges. Specifically, for each i in the ranges [1 .. n] and [-n .. -1], the identity PBR has an edge from i to -i.

IsIdentityPBR returns true is the PBR x is an identity PBR and false if it is not.

Example
```
gap> x := PBR([[-2], [-1]], [[1], [2]]);
PBR([ [ -2 ], [ -1 ] ], [ [ 1 ], [ 2 ] ])
gap> IsIdentityPBR(x);
false
gap> x := PBR([[-1]], [[1]]);
PBR([ [ -1 ] ], [ [ 1 ] ])
gap> IsIdentityPBR(x);
true
```

4.5.7 IsUniversalPBR

▷ IsUniversalPBR(x) (property)

Returns: true or false.

A PBR of degree n is UNIVERSAL if it has 4 * n ^ 2 edges, i.e. every possible edge.

Example
```
gap> x := PBR([[]], [[]]);
PBR([ [  ] ], [ [  ] ])
gap> IsUniversalPBR(x);
false
gap> x := PBR([[-2, 1], [2]], [[-1], [-2, 1]]);
PBR([ [ -2, 1 ], [ 2 ] ], [ [ -1 ], [ -2, 1 ] ])
gap> IsUniversalPBR(x);
false
gap> x := PBR([[-1, 1]], [[-1, 1]]);
PBR([ [ -1, 1 ] ], [ [ -1, 1 ] ])
gap> IsUniversalPBR(x);
true
```

4.5.8 IsBipartitionPBR

▷ IsBipartitionPBR(x) (property)

▷ IsBlockBijectionPBR(x) (property)

Returns: true or false.

If the PBR x defines a bipartition, then IsBipartitionPBR returns true, and if not, then it returns false.

A PBR x defines a bipartition if and only if when considered as a boolean matrix it is an equivalence.

If x satisfies IsBipartitionPBR and when considered as a bipartition it is a block bijection, then IsBlockBijectionPBR returns true.

Example
```
gap> x := PBR([[-1, 3], [-1, 3], [-2, 1, 2, 3]],
>             [[-2, -1, 2], [-2, -1, 1, 2, 3],
>              [-2, -1, 1, 2]]);
PBR([ [ -1, 3 ], [ -1, 3 ], [ -2, 1, 2, 3 ] ],
  [ [ -2, -1, 2 ], [ -2, -1, 1, 2, 3 ], [ -2, -1, 1, 2 ] ])
gap> IsBipartitionPBR(x);
false
gap> x := PBR([[-2, -1, 1], [2, 3], [2, 3]],
>             [[-2, -1, 1], [-2, -1, 1], [-3]]);
PBR([ [ -2, -1, 1 ], [ 2, 3 ], [ 2, 3 ] ],
  [ [ -2, -1, 1 ], [ -2, -1, 1 ], [ -3 ] ])
gap> IsBipartitionPBR(x);
true
gap> IsBlockBijectionPBR(x);
false
```

4.5.9 IsTransformationPBR

▷ `IsTransformationPBR(x)` (property)

Returns: `true` or `false`.

If the PBR *x* defines a transformation, then `IsTransformationPBR` returns `true`, and if not, then `false` is returned.

A PBR *x* defines a transformation if and only if it satisfies `IsBipartitionPBR` (4.5.8) and when it is considered as a bipartition it satisfies `IsTransBipartition` (3.5.12).

With this definition, `AsPBR` (4.3.1) and `AsTransformation` (4.3.2) define mutually inverse isomorphisms from the full transformation monoid of degree `n` to the submonoid of the full PBR monoid of degree `n` consisting of all the elements satisfying `IsTransformationPBR`.

Example
```
gap> x := PBR([[-3], [-1], [-3]], [[2], [], [1, 3]]);
PBR([ [ -3 ], [ -1 ], [ -3 ] ], [ [ 2 ], [  ], [ 1, 3 ] ])
gap> IsTransformationPBR(x);
true
gap> x := AsTransformation(x);
Transformation( [ 3, 1, 3 ] )
gap> AsPBR(x) * AsPBR(x) = AsPBR(x ^ 2);
true
gap> Number(FullPBRMonoid(1), IsTransformationPBR);
1
gap> x := PBR([[-2, -1, 2], [-2, 1, 2]], [[-1, 1], [-2]]);
PBR([ [ -2, -1, 2 ], [ -2, 1, 2 ] ], [ [ -1, 1 ], [ -2 ] ])
gap> IsTransformationPBR(x);
false
```

4.5.10 IsDualTransformationPBR

▷ `IsDualTransformationPBR(x)` (property)

Returns: `true` or `false`.

If the PBR *x* defines a dual transformation, then `IsDualTransformationPBR` returns `true`, and if not, then `false` is returned.

A PBR *x* defines a dual transformation if and only if `Star(x)` satisfies `IsTransformationPBR` (4.5.9).

Example
```
gap> x := PBR([[-3, 1, 3], [-1, 2], [-3, 1, 3]],
>             [[-1, 2], [-2], [-3, 1, 3]]);
PBR([ [ -3, 1, 3 ], [ -1, 2 ], [ -3, 1, 3 ] ],
  [ [ -1, 2 ], [ -2 ], [ -3, 1, 3 ] ])
gap> IsDualTransformationPBR(x);
false
gap> IsDualTransformationPBR(Star(x));
true
gap> Number(FullPBRMonoid(1), IsDualTransformationPBR);
1
```

4.5.11 IsPartialPermPBR

▷ `IsPartialPermPBR(x)` (property)

Returns: `true` or `false`.

If the PBR *x* defines a partial permutation, then `IsPartialPermPBR` returns `true`, and if not, then `false` is returned.

A PBR *x* defines a partial perm if and only if it satisfies `IsBipartitionPBR` (4.5.8) and and when it is considered as a bipartition it satisfies `IsPartialPermBipartition` (3.5.15).

With this definition, `AsPBR` (4.3.1) and `AsPartialPerm` (4.3.3) define mutually inverse isomorphisms from the symmetric inverse monoid of degree `n` to the submonoid of the full PBR monoid of degree `n` consisting of all the elements satisfying `IsPartialPermPBR`.

Example
```
gap> x := PBR([[-1, 1], [2]], [[-1, 1], [-2]]);
PBR([ [ -1, 1 ], [ 2 ] ], [ [ -1, 1 ], [ -2 ] ])
gap> IsPartialPermPBR(x);
true
gap> x := PartialPerm([3, 1]);
[2,1,3]
gap> AsPBR(x) * AsPBR(x) = AsPBR(x ^ 2);
true
gap> Number(FullPBRMonoid(1), IsPartialPermPBR);
2
```

4.5.12 IsPermPBR

▷ `IsPermPBR(x)` (property)

Returns: `true` or `false`.

If the PBR *x* defines a permutation, then `IsPermPBR` returns `true`, and if not, then `false` is returned.

A PBR *x* defines a permutation if and only if it satisfies `IsBipartitionPBR` (4.5.8) and and when it is considered as a bipartition it satisfies `IsPermBipartition` (3.5.14).

With this definition, `AsPBR` (4.3.1) and `AsPermutation` (4.3.4) define mutually inverse isomorphisms from the symmetric group of degree `n` to the subgroup of the full PBR monoid of degree `n` consisting of all the elements satisfying `IsPermPBR` (i.e. the `GroupOfUnits` (13.8.1) of the full PBR monoid of degree `n`).

Example
```
gap> x := PBR([[-2, 1], [-4, 2], [-1, 3], [-3, 4]],
> [[-1, 3], [-2, 1], [-3, 4], [-4, 2]]);;
gap> IsPermPBR(x);
true
gap> x := (1, 5)(2, 4, 3);
(1,5)(2,4,3)
gap> y := (1, 4, 3)(2, 5);
(1,4,3)(2,5)
gap> AsPBR(x) * AsPBR(y) = AsPBR(x * y);
true
gap> Number(FullPBRMonoid(1), IsPermPBR);
1
```

4.6 Semigroups of PBRs

Semigroups and monoids of PBRs can be created in the usual way in GAP using the functions `Semigroup` (**Reference: Semigroup**) and `Monoid` (**Reference: Monoid**); see Chapter 6 for more details.

It is possible to create inverse semigroups and monoids of PBRs using `InverseSemigroup` (**Reference: InverseSemigroup**) and `InverseMonoid` (**Reference: InverseMonoid**) when the argument is a collection of PBRs satisfying `IsBipartitionPBR` (4.5.8) and when considered as bipartitions, the collection satisfies `IsGeneratorsOfInverseSemigroup`.

Note that every PBR semigroup in Semigroups is finite.

4.6.1 IsPBRSemigroup

▷ `IsPBRSemigroup(S)` (filter)

▷ `IsPBRMonoid(S)` (filter)

Returns: `true` or `false`.

A *PBR semigroup* is simply a semigroup consisting of PBRs. An object *obj* is a PBR semigroup in GAP if it satisfies `IsSemigroup` (**Reference: IsSemigroup**) and `IsPBRCollection` (4.1.2).

A *PBR monoid* is a monoid consisting of PBRs. An object *obj* is a PBR monoid in GAP if it satisfies `IsMonoid` (**Reference: IsMonoid**) and `IsPBRCollection` (4.1.2).

Note that it is possible for a PBR semigroup to have a multiplicative neutral element (i.e. an identity element) but not to satisfy `IsPBRMonoid`. For example,

Example
```
gap> x := PBR([[-2, -1, 3], [-2, 2], [-3, -2, 1, 2, 3]],
>             [[-3, -2, -1, 2, 3], [-3, -2, -1, 2, 3], [-1]]);;
gap> S := Semigroup(x, One(x));
<commutative pbr monoid of degree 3 with 1 generator>
gap> IsMonoid(S);
true
gap> IsPBRMonoid(S);
true
gap> S := Semigroup([
> PBR([[-2, 1], [-3, 2], [-1, 3], [-4, 4, 5], [-4, 4, 5]],
>     [[-1, 3], [-2, 1], [-3, 2], [-4, 4, 5], [-5]]),
> PBR([[-2, 1], [-1, 2], [-3, 3], [-4, 4, 5], [-4, 4, 5]],
```

```
>      [[-1, 2], [-2, 1], [-3, 3], [-4, 4, 5], [-5]]),
>  PBR([[-1, 1, 3], [-2, 2], [-1, 1, 3], [-4, 4, 5], [-4, 4, 5]],
>      [[-1, 1, 3], [-2, 2], [-3], [-4, 4, 5], [-5]])]);
<pbr semigroup of degree 5 with 3 generators>
gap> One(S);
fail
gap> MultiplicativeNeutralElement(S);
PBR([ [ -1, 1 ], [ -2, 2 ], [ -3, 3 ], [ -4, 4, 5 ], [ -4, 4, 5 ] ],
  [ [ -1, 1 ], [ -2, 2 ], [ -3, 3 ], [ -4, 4, 5 ], [ -5 ] ])
gap> IsPBRMonoid(S);
false
```

In this example `S` cannot be converted into a monoid using `AsMonoid` (**Reference: AsMonoid**) since the `One` (**Reference: One**) of any element in `S` differs from the multiplicative neutral element.

For more details see `IsMagmaWithOne` (**Reference: IsMagmaWithOne**).

4.6.2 DegreeOfPBRSemigroup

▷ `DegreeOfPBRSemigroup(S)` (attribute)

Returns: A non-negative integer.

The *degree* of a PBR semigroup *S* is just the degree of any (and every) element of *S*.

Example

```
gap> S := Semigroup(
>  PBR([[-1, 1], [-2, 2], [-3, 3]],
>      [[-1, 1], [-2, 2], [-3, 3]]),
>  PBR([[1, 2], [1, 2], [-3, 3]],
>      [[-2, -1], [-2, -1], [-3, 3]]),
>  PBR([[-1, 1], [2, 3], [2, 3]],
>      [[-1, 1], [-3, -2], [-3, -2]]));
<pbr semigroup of degree 3 with 3 generators>
gap> DegreeOfPBRSemigroup(S);
3
```

Chapter 5

Matrices over semirings

In this chapter we describe the functionality in Semigroups for creating matrices over semirings. ONLY SQUARE MATRICES ARE CURRENTLY SUPPORTED. We use the term MATRIX to mean SQUARE MATRIX everywhere in this manual.

For reference, matrices over the following semirings are currently supported:

the Boolean semiring
: the set $\{0,1\}$ where $0+0=0$, $0+1=1+1=1+0=1$, $1\cdot 0=0\cdot 0=0\cdot 1=0$, and $1\cdot 1=1$.

the max-plus semiring
: the set of integers and negative infinity $\mathbb{Z}\cup\{-\infty\}$ with operations max and plus.

the min-plus semiring
: the set of integers and infinity $\mathbb{Z}\cup\{\infty\}$ with operations min and plus;

tropical max-plus semirings
: the set $\{-\infty,0,1,\ldots,t\}$ for some threshold t with operations max and plus;

tropical min-plus semirings
: the set $\{0,1,\ldots,t,\infty\}$ for some threshold t with operations min and plus;

the semiring $\mathbb{N}_{t,p}$
: the semiring $\mathbb{N}_{t,p}=\{0,1,\ldots,t,t+1,\ldots,t+p-1\}$ for some threshold t and period p under addition and multiplication modulo the congruence $t=t+p$;

the integers
: the usual ring of integers;

finite fields
: the finite fields `GF(q^d)` for prime `q` and some positive integer `d`.

With the exception of matrices of finite fields, semigroups of matrices in Semigroups are of the second type described in Section 1.1. In other words, a version of the Froidure-Pin Algorithm [FP97] is used to compute semigroups of these types, i.e it is possible that all of the elements of such a semigroup are enumerated and stored in the memory of your computer.

5.1 Creating matrices over semirings

In this section we describe the two main operations for creating matrices over semirings in Semigroups, and the categories, attributes, and operations which apply to every matrix over one of the semirings given at the start of this chapter.

There are several special methods for boolean matrices, which can be found in Section 5.3. There are also several special methods for finite fields, which can be found in section 5.4.

5.1.1 IsMatrixOverSemiring

▷ `IsMatrixOverSemiring(obj)` (Category)

Returns: `true` or `false`.

Every matrix over a semiring in Semigroups is a member of the category `IsMatrixOverSemiring`, which is a subcategory of `IsMultiplicativeElementWithOne` (**Reference: IsMultiplicativeElementWithOne**), `IsAssociativeElement` (**Reference: IsAssociativeElement**), and `IsPositionalObjectRep`; see (**Reference: Representation**).

Every matrix over a semiring in Semigroups is a square matrix.

Basic operations for matrices over semirings are: `DimensionOfMatrixOverSemiring` (5.1.3), `TransposedMat` (**Reference: TransposedMat**), and `One` (**Reference: One**).

5.1.2 IsMatrixOverSemiringCollection

▷ `IsMatrixOverSemiringCollection(obj)` (Category)
▷ `IsMatrixOverSemiringCollColl(obj)` (Category)

Returns: `true` or `false`.

Every collection of matrices over the same semiring belongs to the category `IsMatrixOverSemiringCollection`. For example, semigroups of matrices over a semiring belong to `IsMatrixOverSemiringCollection`.

Every collection of collections of matrices over the same semiring belongs to the category `IsMatrixOverSemiringCollColl`. For example, a list of semigroups of matrices over semirings belongs to `IsMatrixOverSemiringCollColl`.

5.1.3 DimensionOfMatrixOverSemiring

▷ `DimensionOfMatrixOverSemiring(mat)` (attribute)

Returns: A positive integer.

If *mat* is a matrix over a semiring (i.e. belongs to the category `IsMatrixOverSemiring` (5.1.1)), then *mat* is a square `n` by `n` matrix. `DimensionOfMatrixOverSemiring` returns the dimension `n` of *mat*.

Example
```
gap> x := BooleanMat([[1, 0, 0, 1],
>                     [0, 1, 1, 0],
>                     [1, 0, 1, 1],
>                     [0, 0, 0, 1]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [0, 1, 1, 0], [1, 0, 1, 1],
  [0, 0, 0, 1]])
gap> DimensionOfMatrixOverSemiring(x);
4
```

5.1.4 DimensionOfMatrixOverSemiringCollection

▷ `DimensionOfMatrixOverSemiringCollection(coll)` (attribute)

Returns: A positive integer.

If *coll* is a collection of matrices over a semiring (i.e. belongs to the category `IsMatrixOverSemiringCollection` (5.1.2)), then the elements of *coll* are square `n` by `n` matrices. `DimensionOfMatrixOverSemiringCollection` returns the dimension `n` of these matrices.

Example
```
gap> x := BooleanMat([[1, 0, 0, 1],
>                     [0, 1, 1, 0],
>                     [1, 0, 1, 1],
>                     [0, 0, 0, 1]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [0, 1, 1, 0], [1, 0, 1, 1],
  [0, 0, 0, 1]])
gap> DimensionOfMatrixOverSemiringCollection(Semigroup(x));
4
```

5.1.5 Matrix (for a filter and a matrix)

▷ `Matrix(filt, mat[, threshold[, period]])` (operation)

▷ `Matrix(semiring, mat)` (operation)

Returns: A matrix over semiring.

This operation can be used to construct a matrix over a semiring in Semigroups.

In its first form, the first argument *filt* specifies the filter to be used to create the matrix, the second argument *mat* is a GAP matrix (i.e. a list of lists) compatible with *filt*, the third and fourth arguments *threshold* and *period* (if required) must be positive integers.

filt
: This must be one of the filters given in Section 5.1.8.

mat
: This must be a list of `n` lists each of length `n` (i.e. a square matrix), consisting of elements belonging to the underlying semiring described by *filt*, and *threshold* and *period* if present. An error is given if *mat* is not compatible with the other arguments.

  For example, if *filt* is `IsMaxPlusMatrix`, then the entries of *mat* must belong to the max-plus semiring, i.e. they must be integers or $-\infty$.

  The supported semirings are fully described at the start of this chapter.

threshold
: If *filt* is any of `IsTropicalMaxPlusMatrix` (5.1.8), `IsTropicalMinPlusMatrix` (5.1.8), or `IsNTPMatrix` (5.1.8), then this argument specifies the threshold of the underlying semiring of the matrix being created.

period
: If *filt* is `IsNTPMatrix` (5.1.8), then this argument specifies the period of the underlying semiring of the matrix being created.

In its second form, the arguments should be a semiring *semiring* and matrix *mat* with entries in *semiring*. Currently, the only supported semirings are finite fields of prime order, and the integers `Integers` (**Reference: Integers**).

The function `BooleanMat` (5.3.1) is provided for specifically creating boolean matrices.

Example

```
gap> Matrix(IsBooleanMat, [[1, 0, 0, 0],
>                          [0, 0, 0, 0],
>                          [1, 1, 1, 1],
>                          [1, 0, 1, 1]]);
Matrix(IsBooleanMat, [[1, 0, 0, 0], [0, 0, 0, 0], [1, 1, 1, 1],
  [1, 0, 1, 1]])
gap> Matrix(IsMaxPlusMatrix, [[4, 0, -2],
>                             [1, -3, 0],
>                             [5, -1, -4]]);
Matrix(IsMaxPlusMatrix, [[4, 0, -2], [1, -3, 0], [5, -1, -4]])
gap> Matrix(IsMinPlusMatrix, [[-1, infinity],
>                             [1, -1]]);
Matrix(IsMinPlusMatrix, [[-1, infinity], [1, -1]])
gap> Matrix(IsTropicalMaxPlusMatrix, [[3, 2, 4],
>                                     [3, 1, 1],
>                                     [-infinity, 1, 1]],
>           9);
Matrix(IsTropicalMaxPlusMatrix, [[3, 2, 4], [3, 1, 1],
  [-infinity, 1, 1]], 9)
gap> Matrix(IsTropicalMinPlusMatrix, [[1, 1, 1],
>                                     [0, 3, 0],
>                                     [1, 1, 3]],
>           9);
Matrix(IsTropicalMinPlusMatrix, [[1, 1, 1], [0, 3, 0], [1, 1, 3]], 9)
gap> Matrix(IsNTPMatrix, [[0, 0, 0],
>                         [2, 0, 1],
>                         [2, 2, 2]],
>           2, 1);
Matrix(IsNTPMatrix, [[0, 0, 0], [2, 0, 1], [2, 2, 2]], 2, 1)
gap> Matrix(IsIntegerMatrix, [[-1, -2, 0],
>                             [0, 3, -1],
>                             [1, 0, -3]]);
Matrix(IsIntegerMatrix, [[-1, -2, 0], [0, 3, -1], [1, 0, -3]])
gap> Matrix(Integers, [[-1, -2, 0],
>                      [0, 3, -1],
>                      [1, 0, -3]]);
Matrix(IsIntegerMatrix, [[-1, -2, 0], [0, 3, -1], [1, 0, -3]])
```

5.1.6 AsMatrix (for a filter and a matrix)

▷ `AsMatrix(filt, mat)` (operation)

▷ `AsMatrix(filt, mat, threshold)` (operation)

▷ `AsMatrix(filt, mat, threshold, period)` (operation)

Returns: A matrix.

This operation can be used to change the representation of certain matrices over semirings. If *mat* is a matrix over a semiring (in the category `IsMatrixOverSemiring` (5.1.1)), then `AsMatrix` returns a new matrix corresponding to *mat* of the type specified by the filter *filt*, and if applicable the arguments *threshold* and *period*. The dimension of the matrix *mat* is not changed by this operation.

The version of the operation with arguments *filt* and *mat* can be applied to:

- `IsMinPlusMatrix` (5.1.8) and a tropical min-plus matrix (i.e. convert a tropical min-plus matrix to a (non-tropical) min-plus matrix);
- `IsMaxPlusMatrix` (5.1.8) and a tropical max-plus matrix;
- `IsIntegerMatrix` (5.1.8) and an ntp matrix.

The version of the operation with arguments *filt*, *mat*, and *threshold* can be applied to:

- `IsTropicalMinPlusMatrix` (5.1.8), a tropical min-plus or min-plus matrix, and a value for the threshold of the resulting matrix.
- `IsTropicalMaxPlusMatrix` (5.1.8) and a tropical max-plus, or max-plus matrix, and a value for the threshold of the resulting matrix.

The version of the operation with arguments *filt*, *mat*, *threshold*, and *period* can be applied to `IsNTPMatrix` (5.1.8) and an ntp matrix, or integer matrix.

When converting matrices with negative entries to an ntp, tropical max-plus, or tropical min-plus matrix, the entry is replaced with its absolute value.

When converting non-tropical matrices to tropical matrices entries higher than the specified threshold are reduced to the threshold.

Example

```
gap> mat := Matrix(IsTropicalMinPlusMatrix, [[0, 1, 3],
>                                             [1, 1, 6],
>                                             [0, 4, 2]], 10);;
gap> AsMatrix(IsMinPlusMatrix, mat);
Matrix(IsMinPlusMatrix, [[0, 1, 3], [1, 1, 6], [0, 4, 2]])
gap> mat := Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
>                                             [0, 1, 3],
>                                             [4, 1, 0]], 10);;
gap> AsMatrix(IsMaxPlusMatrix, mat);
Matrix(IsMaxPlusMatrix, [[-infinity, -infinity, 3], [0, 1, 3],
  [4, 1, 0]])
gap> mat := Matrix(IsNTPMatrix, [[1, 2, 2],
>                                 [0, 2, 0],
>                                 [1, 3, 0]], 4, 5);;
gap> AsMatrix(IsIntegerMatrix, mat);
Matrix(IsIntegerMatrix, [[1, 2, 2], [0, 2, 0], [1, 3, 0]])
gap> mat := Matrix(IsMinPlusMatrix, [[0, 1, 3], [1, 1, 6], [0, 4, 2]]);;
gap> mat := AsMatrix(IsTropicalMinPlusMatrix, mat, 2);
Matrix(IsTropicalMinPlusMatrix, [[0, 1, 2], [1, 1, 2], [0, 2, 2]], 2)
gap> mat := AsMatrix(IsTropicalMinPlusMatrix, mat, 1);
Matrix(IsTropicalMinPlusMatrix, [[0, 1, 1], [1, 1, 1], [0, 1, 1]], 1)
gap> mat := Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
>                                             [0, 1, 3],
>                                             [4, 1, 0]], 10);;
gap> AsMatrix(IsTropicalMaxPlusMatrix, mat, 4);
Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
  [0, 1, 3], [4, 1, 0]], 4)
gap> mat := Matrix(IsMaxPlusMatrix, [[-infinity, -infinity, 3],
>                                     [0, 1, 3],
>                                     [4, 1, 0]]);;
gap> AsMatrix(IsTropicalMaxPlusMatrix, mat, 10);
```

```
Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
  [0, 1, 3], [4, 1, 0]], 10)
gap> mat := Matrix(IsNTPMatrix, [[0, 1, 0],
>                                [1, 3, 1],
>                                [1, 0, 1]], 10, 10);;
gap> mat := AsMatrix(IsNTPMatrix, mat, 5, 6);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 3, 1], [1, 0, 1]], 5, 6)
gap> mat := AsMatrix(IsNTPMatrix, mat, 2, 6);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 3, 1], [1, 0, 1]], 2, 6)
gap> mat := AsMatrix(IsNTPMatrix, mat, 2, 1);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 2, 1], [1, 0, 1]], 2, 1)
gap> mat := AsMatrix(IsIntegerMatrix, mat);
Matrix(IsIntegerMatrix, [[0, 1, 0], [1, 2, 1], [1, 0, 1]])
gap> AsMatrix(IsNTPMatrix, mat, 1, 2);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 2, 1], [1, 0, 1]], 1, 2)
```

5.1.7 RandomMatrix (for a filter and a matrix)

▷ `RandomMatrix(filt, dim[, threshold[, period]])` (function)
▷ `RandomMatrix(semiring, dim)` (function)

Returns: A matrix over semiring.

This operation can be used to construct a random matrix over a semiring in Semigroups. The usage of `RandomMatrix` is similar to that of `Matrix` (5.1.5).

In its first form, the first argument *filt* specifies the filter to be used to create the matrix, the second argument *dim* is dimension of the matrix, the third and fourth arguments *threshold* and *period* (if required) must be positive integers.

filt
: This must be one of the filters given in Section 5.1.8.

dim
: This must be a positive integer.

threshold
: If *filt* is any of `IsTropicalMaxPlusMatrix` (5.1.8), `IsTropicalMinPlusMatrix` (5.1.8), or `IsNTPMatrix` (5.1.8), then this argument specifies the threshold of the underlying semiring of the matrix being created.

period
: If *filt* is `IsNTPMatrix` (5.1.8), then this argument specifies the period of the underlying semiring of the matrix being created.

In its second form, the arguments should be a semiring *semiring* and dimension *dim*. Currently, the only supported semirings are finite fields of prime order and the integers `Integers` (**Reference: Integers**).

Example
```
gap> RandomMatrix(IsBooleanMat, 3);
Matrix(IsBooleanMat, [[1, 0, 0], [1, 0, 1], [1, 0, 1]])
gap> RandomMatrix(IsMaxPlusMatrix, 2);
Matrix(IsMaxPlusMatrix, [[1, -infinity], [1, 0]])
gap> RandomMatrix(IsMinPlusMatrix, 3);
```

```
Matrix(IsMinPlusMatrix, [[infinity, 2, infinity], [4, 0, -2], [1, -3, 0]])
gap> RandomMatrix(IsTropicalMaxPlusMatrix, 3, 5);
Matrix(IsTropicalMaxPlusMatrix, [[5, 1, 4], [1, -infinity, 1], [1, 0, 2]],
  5)
gap> RandomMatrix(IsTropicalMinPlusMatrix, 3, 2);
Matrix(IsTropicalMinPlusMatrix, [[1, -infinity, -infinity], [1, 1, 1],
  [2, 2, 1]], 2)
gap> RandomMatrix(IsNTPMatrix, 3, 2, 5);
Matrix(IsNTPMatrix, [[1, 1, 1], [1, 1, 0], [3, 0, 1]], 2, 5)
gap> RandomMatrix(IsIntegerMatrix, 2);
Matrix(IsIntegerMatrix, [[1, 3], [0, 0]])
gap> RandomMatrix(Integers, 2);
Matrix(IsIntegerMatrix, [[-1, 0], [0, -1]])
gap> RandomMatrix(GF(5), 1);
Matrix(GF(5), [[Z(5)^0]])
```

5.1.8 Matrix filters

▷ `IsBooleanMat(obj)` (Category)
▷ `IsMatrixOverFiniteField(obj)` (Category)
▷ `IsMaxPlusMatrix(obj)` (Category)
▷ `IsMinPlusMatrix(obj)` (Category)
▷ `IsTropicalMatrix(obj)` (Category)
▷ `IsTropicalMaxPlusMatrix(obj)` (Category)
▷ `IsTropicalMinPlusMatrix(obj)` (Category)
▷ `IsNTPMatrix(obj)` (Category)
▷ `IsIntegerMatrix(obj)` (Category)

Returns: `true` or `false`.

Every matrix over a semiring in Semigroups is a member of one of these categories, which are subcategory of `IsMatrixOverSemiring` (5.1.1).

`IsTropicalMatrix` is a supercategory of `IsTropicalMaxPlusMatrix` and `IsTropicalMinPlusMatrix`.

Basic operations for matrices over semirings include: multiplication via *, `DimensionOfMatrixOverSemiring` (5.1.3), `One` (**Reference: One**), the underlying list of lists used to create the matrix can be accessed using `AsList` (5.1.10), the rows of `mat` can be accessed using `mat[i]` where `i` is between `1` and the dimension of the matrix, it also possible to loop over the rows of a matrix; for tropical matrices `ThresholdTropicalMatrix` (5.1.11); for ntp matrices `ThresholdNTPMatrix` (5.1.12) and `PeriodNTPMatrix` (5.1.12).

For matrices over finite fields see Section 5.4; for Boolean matrices more details can be found in Section 5.3.

5.1.9 Matrix collection filters

▷ `IsBooleanMatCollection(obj)` (Category)
▷ `IsBooleanMatCollColl(obj)` (Category)
▷ `IsMatrixOverFiniteFieldCollection(obj)` (Category)
▷ `IsMatrixOverFiniteFieldCollColl(obj)` (Category)
▷ `IsMaxPlusMatrixCollection(obj)` (Category)

▷ `IsMaxPlusMatrixCollColl(obj)` (Category)
▷ `IsMinPlusMatrixCollection(obj)` (Category)
▷ `IsMinPlusMatrixCollColl(obj)` (Category)
▷ `IsTropicalMatrixCollection(obj)` (Category)
▷ `IsTropicalMaxPlusMatrixCollection(obj)` (Category)
▷ `IsTropicalMaxPlusMatrixCollColl(obj)` (Category)
▷ `IsTropicalMinPlusMatrixCollection(obj)` (Category)
▷ `IsTropicalMinPlusMatrixCollColl(obj)` (Category)
▷ `IsNTPMatrixCollection(obj)` (Category)
▷ `IsNTPMatrixCollColl(obj)` (Category)
▷ `IsIntegerMatrixCollection(obj)` (Category)
▷ `IsIntegerMatrixCollColl(obj)` (Category)

Returns: `true` or `false`.

Every collection of matrices over the same semiring in Semigroups belongs to one of the categories above. For example, semigroups of boolean matrices belong to `IsBooleanMatCollection`.

Similarly, every collection of collections of matrices over the same semiring in Semigroups belongs to one of the categories above.

5.1.10 AsList

▷ `AsList(mat)` (attribute)
▷ `AsMutableList(mat)` (operation)

Returns: A list of lists.

If *mat* is a matrix over a semiring (in the category `IsMatrixOverSemiring` (5.1.1)), then `AsList` returns the underlying list of lists of semiring elements corresponding to *mat*. In this case, the returned list and all of its entries are immutable.

The operation `AsMutableList` returns a mutable copy of the underlying list of lists of the matrix over semiring *mat*.

Example
```
gap> mat := Matrix(IsIntegerMatrix, [[0, 2],
>                                    [3, 5]]);
Matrix(IsIntegerMatrix, [[0, 2], [3, 5]])
gap> AsList(mat);
[ [ 0, 2 ], [ 3, 5 ] ]
gap> mat := Matrix(GF(7), [[Z(7) ^ 3, Z(7) ^ 2],
>                          [Z(7) ^ 4, Z(7)]]);
Matrix(GF(7), [[Z(7)^3, Z(7)^2], [Z(7)^4, Z(7)]])
gap> list := AsList(mat);
[ [ Z(7)^3, Z(7)^2 ], [ Z(7)^4, Z(7) ] ]
gap> IsMutable(list);
false
gap> IsMutable(list[1]);
false
gap> list := AsMutableList(mat);
[ [ Z(7)^3, Z(7)^2 ], [ Z(7)^4, Z(7) ] ]
gap> IsMutable(list);
true
gap> IsMutable(list[1]);
true
```

```
gap> mat = Matrix(BaseDomain(mat), AsList(mat));
true
```

5.1.11 ThresholdTropicalMatrix

▷ ThresholdTropicalMatrix(*mat*) (attribute)

Returns: A positive integer.

If *mat* is a tropical matrix (i.e. belongs to the category IsTropicalMatrix (5.1.8)), then ThresholdTropicalMatrix returns the threshold (i.e. the largest integer) of the underlying semiring.

Example

```
gap> mat := Matrix(IsTropicalMaxPlusMatrix,
> [[0, 3, 0, 2],
>  [1, 1, 1, 0],
>  [-infinity, 1, -infinity, 1],
>  [0, -infinity, 2, -infinity]], 10);
Matrix(IsTropicalMaxPlusMatrix, [[0, 3, 0, 2], [1, 1, 1, 0],
  [-infinity, 1, -infinity, 1], [0, -infinity, 2, -infinity]], 10)
gap> ThresholdTropicalMatrix(mat);
10
gap> mat := Matrix(IsTropicalMaxPlusMatrix,
> [[0, 3, 0, 2],
>  [1, 1, 1, 0],
>  [-infinity, 1, -infinity, 1],
>  [0, -infinity, 2, -infinity]], 3);
Matrix(IsTropicalMaxPlusMatrix, [[0, 3, 0, 2], [1, 1, 1, 0],
  [-infinity, 1, -infinity, 1], [0, -infinity, 2, -infinity]], 3)
gap> ThresholdTropicalMatrix(mat);
3
```

5.1.12 ThresholdNTPMatrix

▷ ThresholdNTPMatrix(*mat*) (attribute)

▷ PeriodNTPMatrix(*mat*) (attribute)

Returns: A positive integer.

An NTP MATRIX is a matrix with entries in a semiring $\mathbb{N}_{t,p} = \{0, 1, \ldots, t, t+1, \ldots, t+p-1\}$ for some threshold t and period p under addition and multiplication modulo the congruence $t = t + p$.

If *mat* is a ntp matrix (i.e. belongs to the category IsNTPMatrix (5.1.8)), then ThresholdNTPMatrix and PeriodNTPMatrix return the threshold and period of the underlying semiring, respectively.

Example

```
gap> mat := Matrix(IsNTPMatrix, [[1, 1, 0],
>                                [2, 1, 0],
>                                [0, 1, 1]],
>                   1, 2);
Matrix(IsNTPMatrix, [[1, 1, 0], [2, 1, 0], [0, 1, 1]], 1, 2)
gap> ThresholdNTPMatrix(mat);
1
gap> PeriodNTPMatrix(mat);
2
```

```
gap> mat := Matrix(IsNTPMatrix, [[2, 1, 3],
>                                [0, 5, 1],
>                                [4, 1, 0]],
>                   3, 4);
Matrix(IsNTPMatrix, [[2, 1, 3], [0, 5, 1], [4, 1, 0]], 3, 4)
gap> ThresholdNTPMatrix(mat);
3
gap> PeriodNTPMatrix(mat);
4
```

5.2 Operators for matrices over semirings

`mat1 * mat2`

returns the product of the matrices *mat1* and *mat2* of equal dimension over the same semiring using the usual matrix multiplication with the operations + and ∗ from the underlying semiring.

`mat1 < mat2`

returns `true` if when considered as a list of rows, the matrix *mat1* is short-lex less than the matrix *mat2*, and `false` if this is not the case. This means that a matrix of lower dimension is less than a matrix of higher dimension.

`mat1 = mat2`

returns `true` if the matrix *mat1* equals the matrix *mat2* (i.e. the entries are equal and the underlying semirings are equal) and returns `false` if it does not.

5.3 Boolean matrices

In this section we describe the operations, properties, and attributes in Semigroups specifically for Boolean matrices. These include:

- `NumberBooleanMat` (5.3.6)
- `Successors` (5.3.5)
- `IsRowTrimBooleanMat` (5.3.9), `IsColTrimBooleanMat` (5.3.9), and `IsTrimBooleanMat` (5.3.9),
- `CanonicalBooleanMat` (5.3.8)
- `IsSymmetricBooleanMat` (5.3.10)
- `IsAntiSymmetricBooleanMat` (5.3.13)
- `IsTransitiveBooleanMat` (5.3.12)
- `IsReflexiveBooleanMat` (5.3.11)
- `IsTotalBooleanMat` (5.3.14)
- `IsOntoBooleanMat` (5.3.14)

- `IsPartialOrderBooleanMat` (5.3.15)
- `IsEquivalenceBooleanMat` (5.3.16)

5.3.1 BooleanMat

▷ `BooleanMat(arg)` (function)

Returns: A boolean matrix.

`BooleanMat` returns the boolean matrix `mat` defined by its argument. The argument can be any of the following:

a matrix with entries `0` and/or `1`
: the argument *arg* is list of `n` lists of length `n` consisting of the values `0` and `1`;

a matrix with entries `true` and/or `false`
: the argument *arg* is list of `n` lists of length `n` consisting of the values `true` and `false`;

successors
: the argument *arg* is list of `n` sublists of consisting of positive integers not greater than `n`. In this case, the entry `j` in the sublist in position `i` of *arg* indicates that the entry in position `(i, j)` of the created boolean matrix is `true`.

`BooleanMat` returns an error if the argument is not one of the above types.

Example
```
gap> x := BooleanMat([[true, false], [true, true]]);
Matrix(IsBooleanMat, [[1, 0], [1, 1]])
gap> y := BooleanMat([[1, 0], [1, 1]]);
Matrix(IsBooleanMat, [[1, 0], [1, 1]])
gap> z := BooleanMat([[1], [1, 2]]);
Matrix(IsBooleanMat, [[1, 0], [1, 1]])
gap> x = y;
true
gap> y = z;
true
gap> Display(x);
1 0
1 1
```

5.3.2 AsBooleanMat

▷ `AsBooleanMat(x[, n])` (operation)

Returns: A boolean matrix.

`AsBooleanMat` returns the pbr, bipartition, permutation, transformation, or partial permutation *x*, as a boolean matrix of dimension *n*.

There are several possible arguments for `AsBooleanMat`:

permutations
: If *x* is a permutation and *n* is a positive integer, then `AsBooleanMat(x, n)` returns the boolean matrix `mat` of dimension *n* such that `mat[i][j] = true` if and only if `j = i ^ x`.

: If no positive integer *n* is specified, then the largest moved point of *x* is used as the value for *n*; see `LargestMovedPoint` (**Reference: LargestMovedPoint (for a permutation)**).

transformations

If `x` is a transformation and `n` is a positive integer such that `x` is a transformation of `[1 .. n]`, then `AsTransformation` returns the boolean matrix `mat` of dimension `n` such that `mat[i][j] = true` if and only if `j = i ^ x`.

If the positive integer `n` is not specified, then the degree of `f` is used as the value for `n`.

partial permutations

If `x` is a partial permutation and `n` is a positive integer such that `i ^ x <= n` for all `i` in `[1 .. n]`, then `AsBooleanMat` returns the boolean matrix `mat` of dimension `n` such that `mat[i][j] = true` if and only if `j = i ^ x`.

If the optional argument `n` is not present, then the default value of the maximum of degree and the codegree of `x` is used.

bipartitions

If `x` is a bipartition and `n` is any non-negative integer, then `AsBooleanMat` returns the boolean matrix `mat` of dimension `n` such that `mat[i][j] = true` if and only if `i` and `j` belong to the same block of `x`.

If the optional argument `n` is not present, then twice the degree of `x` is used by default.

pbrs If `x` is a pbr and `n` is any non-negative integer, then `AsBooleanMat` returns the boolean matrix `mat` of dimension `n` such that `mat[i][j] = true` if and only if `i` and `j` are related in `x`.

If the optional argument `n` is not present, then twice the degree of `x` is used by default.

Example

```
gap> Display(AsBooleanMat((1, 2), 5));
0 1 0 0 0
1 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
gap> Display(AsBooleanMat((1, 2)));
0 1
1 0
gap> x := Transformation([1, 3, 4, 1, 3]);;
gap> Display(AsBooleanMat(x));
1 0 0 0 0
0 0 1 0 0
0 0 0 1 0
1 0 0 0 0
0 0 1 0 0
gap> Display(AsBooleanMat(x, 4));
1 0 0 0
0 0 1 0
0 0 0 1
1 0 0 0
gap> x := PartialPerm([1, 2, 3, 6, 8, 10],
>                     [2, 6, 7, 9, 1, 5]);
[3,7][8,1,2,6,9][10,5]
gap> Display(AsBooleanMat(x));
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
```

```
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
gap> x := Bipartition([[1, 4, -2, -3], [2, 3, 5, -5], [-1, -4]]);
<bipartition: [ 1, 4, -2, -3 ], [ 2, 3, 5, -5 ], [ -1, -4 ]>
gap> y := AsBooleanMat(x);
<10x10 boolean matrix>
gap> Display(y);
1 0 0 1 0 0 1 1 0 0
0 1 1 0 1 0 0 0 0 1
0 1 1 0 1 0 0 0 0 1
1 0 0 1 0 0 1 1 0 0
0 1 1 0 1 0 0 0 0 1
0 0 0 0 0 1 0 0 1 0
1 0 0 1 0 0 1 1 0 0
1 0 0 1 0 0 1 1 0 0
0 0 0 0 0 1 0 0 1 0
0 1 1 0 1 0 0 0 0 1
gap> IsEquivalenceBooleanMat(y);
true
gap> AsBooleanMat(x, 1);
Matrix(IsBooleanMat, [[1]])
gap> Display(AsBooleanMat(x, 1));
1
gap> Display(AsBooleanMat(x, 2));
1 0
0 1
gap> Display(AsBooleanMat(x, 3));
1 0 0
0 1 1
0 1 1
gap> Display(AsBooleanMat(x, 11));
1 0 0 1 0 0 1 1 0 0 0
0 1 1 0 1 0 0 0 0 1 0
0 1 1 0 1 0 0 0 0 1 0
1 0 0 1 0 0 1 1 0 0 0
0 1 1 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1 0 0
1 0 0 1 0 0 1 1 0 0 0
1 0 0 1 0 0 1 1 0 0 0
0 0 0 0 0 1 0 0 1 0 0
0 1 1 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0
gap> x := PBR(
> [[-1, 1], [2, 3], [-3, 2, 3]],
> [[-1, 1, 2], [-3, -1, 1, 3], [-3, -1, 1, 2, 3]]);;
gap> AsBooleanMat(x);
Matrix(IsBooleanMat, [[1, 0, 0, 1, 0, 0], [0, 1, 1, 0, 0, 0],
```

```
  [0, 1, 1, 0, 0, 1], [1, 1, 0, 1, 0, 0], [1, 0, 1, 1, 0, 1],
  [1, 1, 1, 1, 0, 1]])
gap> Display(AsBooleanMat(x));
1 0 0 1 0 0
0 1 1 0 0 0
0 1 1 0 0 1
1 1 0 1 0 0
1 0 1 1 0 1
1 1 1 1 0 1
```

5.3.3 \in

▷ `\in(mat1, mat2)` (operation)

Returns: `true` or `false`.

If *mat1* and *mat2* are boolean matrices, then *mat1* `in` *mat2* returns `true` if the binary relation defined by *mat1* is a subset of that defined by *mat2*.

Example
```
gap> x := BooleanMat([[1, 0, 0, 1], [0, 0, 0, 0],
>                     [1, 0, 1, 1], [0, 1, 1, 1]]);;
gap> y := BooleanMat([[1, 0, 1, 0], [1, 1, 1, 0],
>                     [0, 1, 1, 0], [1, 1, 1, 1]]);;
gap> x in y;
false
gap> y in y;
true
```

5.3.4 OnBlist

▷ `OnBlist(blist, mat)` (function)

Returns: A boolean list.

If *blist* is a boolean list of length `n` and *mat* is boolean matrices of dimension `n`, then `OnBlist` returns the product of *blist* (thought of as a row vector over the boolean semiring) and *mat*.

Example
```
gap> mat := BooleanMat([[1, 0, 0, 1],
>                       [0, 0, 0, 0],
>                       [1, 0, 1, 1],
>                       [0, 1, 1, 1]]);;
gap> blist := BlistList([1 .. 4], [1, 2]);
[ true, true, false, false ]
gap> OnBlist(blist, mat);
[ true, false, false, true ]
```

5.3.5 Successors

▷ `Successors(mat)` (attribute)

Returns: A list of lists of positive integers.

A row of a boolean matrix of dimension `n` can be thought of of as the characteristic function of a subset `S` of `[1 .. n]`, i.e. `i in S` if and only if the `i`th component of the row equals 1. We refer to the subset `S` as the SUCCESSORS of the row.

If *mat* is a boolean matrix, then `Successors` returns the list of successors of the rows of *mat*.

Example
```
gap> mat := BooleanMat([[1, 0, 1, 1],
>                       [1, 0, 0, 0],
>                       [0, 0, 1, 0],
>                       [1, 1, 0, 0]]);;
gap> Successors(mat);
[ [ 1, 3, 4 ], [ 1 ], [ 3 ], [ 1, 2 ] ]
```

5.3.6 BooleanMatNumber

▷ `BooleanMatNumber(m, n)` (operation)

▷ `NumberBooleanMat(mat)` (operation)

Returns: A boolean matrix, or a positive integer.

These functions implement a bijection from the set of all boolean matrices of dimension *n* and the numbers `[1 .. 2 ^ (n ^ 2)]`.

More precisely, if *m* and *n* are positive integers such that *m* is at most `2 ^ (n ^ 2)`, then `BooleanMatNumber` returns the *m*th *n* by *n* boolean matrix.

If *mat* is an *n* by *n* boolean matrix, then `NumberBooleanMat` returns the number in `[1 .. 2 ^ (n ^ 2)]` that corresponds to *mat*.

Example
```
gap> mat := BooleanMat([[0, 1, 1, 0],
>                       [1, 0, 1, 1],
>                       [1, 1, 0, 1],
>                       [0, 1, 0, 1]]);;
gap> NumberBooleanMat(mat);
27606
gap> Display(BooleanMatNumber(27606, 4));
0 1 1 0
1 0 1 1
1 1 0 1
0 1 0 1
```

5.3.7 BlistNumber

▷ `BlistNumber(m, n)` (function)

▷ `NumberBlist(blist)` (function)

Returns: A boolean list, or a positive integer.

These functions implement a bijection from the set of all boolean lists of length *n* and the numbers `[1 .. 2 ^ n]`.

More precisely, if *m* and *n* are positive integers such that *m* is at most `2 ^ n`, then `BlistNumber` returns the *m*th boolean list of length *n*.

If *blist* is a boolean list of length *n*, then `NumberBlist` returns the number in `[1 .. 2 ^ n]` that corresponds to *blist*.

Example
```
gap> blist := BlistList([1 .. 10], []);
[ false, false, false, false, false, false, false, false, false,
  false ]
gap> NumberBlist(blist);
1
```

```
gap> blist := BlistList([1 .. 10], [10]);
[ false, false, false, false, false, false, false, false, false, true
 ]
gap> NumberBlist(blist);
2
gap> BlistNumber(1, 10);
[ false, false, false, false, false, false, false, false, false,
  false ]
gap> BlistNumber(2, 10);
[ false, false, false, false, false, false, false, false, false, true
 ]
```

5.3.8 CanonicalBooleanMat (for a perm group, perm group and boolean matrix)

▷ `CanonicalBooleanMat(G, H, mat)` (operation)
▷ `CanonicalBooleanMat(G, mat)` (operation)
▷ `CanonicalBooleanMat(mat)` (attribute)

Returns: A boolean matrix.

This operation returns a fixed representative of the orbit of the boolean matrix *mat* under the action of the permutation group *G* on its rows and the permutation group *H* on its columns.

In its second form, when only a single permutation group *G* is specified, *G* acts on the rows and columns of *mat* independently.

In its third form, when only a boolean matrix is specified, `CanonicalBooleanMat` returns a fixed representative of the orbit of *mat* under the action of the symmetric group on its rows, and, independently, on its columns. In other words, `CanonicalBooleanMat` returns a canonical boolean matrix equivalent to *mat* up to rearranging rows and columns. This version of `CanonicalBooleanMat` uses Digraphs and its interface with the bliss library for computing automorphism groups and canonical forms of graphs [JK07]. As a consequence, `CanonicalBooleanMat` with a single argument is significantly faster than the versions with 2 or 3 arguments.

Example
```
gap> mat := BooleanMat([[1, 1, 1, 0, 0, 0],
>                        [0, 0, 0, 1, 0, 1],
>                        [1, 0, 0, 1, 0, 1],
>                        [0, 0, 0, 0, 0, 0],
>                        [0, 1, 1, 1, 1, 1],
>                        [0, 1, 1, 0, 1, 0]]);
Matrix(IsBooleanMat, [[1, 1, 1, 0, 0, 0], [0, 0, 0, 1, 0, 1],
  [1, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 1, 1],
  [0, 1, 1, 0, 1, 0]])
gap> CanonicalBooleanMat(mat);
Matrix(IsBooleanMat, [[0, 0, 1, 1, 1, 0], [1, 1, 0, 0, 1, 0],
  [0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0], [1, 1, 0, 0, 0, 1],
  [1, 1, 1, 1, 0, 1]])
gap> Display(CanonicalBooleanMat(mat));
0 0 1 1 1 0
1 1 0 0 1 0
0 0 0 0 0 0
0 0 1 1 0 0
1 1 0 0 0 1
1 1 1 1 0 1
```

```
gap> Display(CanonicalBooleanMat(Group((1, 3)), mat));
0 1 1 0 0 1
0 0 1 0 0 1
1 1 0 1 0 0
0 0 0 0 0 0
1 0 1 1 1 1
1 0 0 1 1 0
gap> Display(CanonicalBooleanMat(Group((1, 3)), Group(()), mat));
1 1 1 0 0 0
0 0 0 1 0 1
0 1 0 1 0 1
0 0 0 0 0 0
1 0 1 1 1 1
1 0 1 0 1 0
```

5.3.9 IsRowTrimBooleanMat

▷ `IsRowTrimBooleanMat(`*mat*`)` (property)

▷ `IsColTrimBooleanMat(`*mat*`)` (property)

▷ `IsTrimBooleanMat(`*mat*`)` (property)

Returns: `true` or `false`.

A row or column of a boolean matrix of dimension `n` can be thought of of as the characteristic function of a subset `S` of `[1 .. n]`, i.e. `i in S` if and only if the `i`th component of the row or column equals `1`.

A boolean matrix is ROW TRIM if no subset induced by a row of *mat* is contained in the subset induced by any other row of *mat*. COLUMN TRIM is defined analogously. A boolean matrix is TRIM if it is both row and column trim.

Example

```
gap> mat := BooleanMat([[0, 1, 1, 0],
>                       [1, 0, 1, 1],
>                       [1, 1, 0, 1],
>                       [0, 1, 1, 1]]);;
gap> IsTrimBooleanMat(mat);
true
gap> mat := BooleanMat([[0, 1, 1, 0],
>                       [0, 0, 1, 0],
>                       [1, 0, 0, 1],
>                       [1, 0, 1, 0]]);;
gap> IsRowTrimBooleanMat(mat);
false
gap> IsColTrimBooleanMat(mat);
false
```

5.3.10 IsSymmetricBooleanMat

▷ `IsSymmetricBooleanMat(`*mat*`)` (property)

Returns: `true` or `false`.

A boolean matrix is SYMMETRIC if it is symmetric about the main diagonal, i.e. *mat*`[i][j] =` *mat*`[j][i]` for all `i, j` in the range `[1 .. n]` where `n` is the dimension of *mat*.

Example
```
gap> mat := BooleanMat([[0, 1, 1, 0],
>                       [1, 0, 1, 1],
>                       [1, 1, 0, 1],
>                       [0, 1, 0, 1]]);
Matrix(IsBooleanMat, [[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 1],
  [0, 1, 0, 1]])
gap> IsSymmetricBooleanMat(mat);
false
gap> mat := BooleanMat([[0, 1, 1, 0],
>                       [1, 0, 1, 1],
>                       [1, 1, 0, 1],
>                       [0, 1, 1, 1]]);
Matrix(IsBooleanMat, [[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 1],
  [0, 1, 1, 1]])
gap> IsSymmetricBooleanMat(mat);
true
```

5.3.11 IsReflexiveBooleanMat

▷ IsReflexiveBooleanMat(*mat*) (property)

Returns: `true` or `false`.

A boolean matrix is REFLEXIVE if every entry in the main diagonal is `true`, i.e. *mat*`[i][i] = true` for all `i` in the range `[1 .. n]` where `n` is the dimension of *mat*.

Example
```
gap> mat := BooleanMat([[1, 0, 0, 0],
>                       [1, 1, 0, 0],
>                       [0, 1, 0, 1],
>                       [1, 1, 1, 1]]);
Matrix(IsBooleanMat, [[1, 0, 0, 0], [1, 1, 0, 0], [0, 1, 0, 1],
  [1, 1, 1, 1]])
gap> IsReflexiveBooleanMat(mat);
false
gap> mat := BooleanMat([[1, 1, 1, 0],
>                       [1, 1, 1, 1],
>                       [1, 1, 1, 1],
>                       [0, 1, 1, 1]]);
Matrix(IsBooleanMat, [[1, 1, 1, 0], [1, 1, 1, 1], [1, 1, 1, 1],
  [0, 1, 1, 1]])
gap> IsReflexiveBooleanMat(mat);
true
```

5.3.12 IsTransitiveBooleanMat

▷ IsTransitiveBooleanMat(*mat*) (property)

Returns: `true` or `false`.

A boolean matrix is TRANSITIVE if whenever *mat*`[i][j] = true` and *mat*`[j][k] = true` then *mat*`[i][k] = true`.

Example
```
gap> x := BooleanMat([[1, 0, 0, 1],
>                     [1, 0, 1, 1],
```

```
>                    [1, 1, 1, 0],
>                    [0, 1, 1, 0]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [1, 0, 1, 1], [1, 1, 1, 0],
  [0, 1, 1, 0]])
gap> IsTransitiveBooleanMat(x);
false
gap> x := BooleanMat([[1, 1, 1, 1],
>                    [1, 1, 1, 1],
>                    [1, 1, 1, 1],
>                    [1, 1, 1, 1]]);
Matrix(IsBooleanMat, [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1],
  [1, 1, 1, 1]])
gap> IsTransitiveBooleanMat(x);
true
```

5.3.13 IsAntiSymmetricBooleanMat

▷ `IsAntiSymmetricBooleanMat(`*mat*`)` (property)

Returns: `true` or `false`.

A boolean matrix is ANTI-SYMMETRIC if whenever *mat*`[i][j] = true` and *mat*`[j][i] = true` then `i = j`.

Example
```
gap> x := BooleanMat([[1, 0, 0, 1],
>                    [1, 0, 1, 1],
>                    [1, 1, 1, 0],
>                    [0, 1, 1, 0]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [1, 0, 1, 1], [1, 1, 1, 0],
  [0, 1, 1, 0]])
gap> IsAntiSymmetricBooleanMat(x);
false
gap> x := BooleanMat([[1, 0, 0, 1],
>                    [1, 0, 1, 0],
>                    [1, 0, 1, 0],
>                    [0, 1, 1, 0]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [1, 0, 1, 0], [1, 0, 1, 0],
  [0, 1, 1, 0]])
gap> IsAntiSymmetricBooleanMat(x);
true
```

5.3.14 IsTotalBooleanMat

▷ `IsTotalBooleanMat(`*mat*`)` (property)

▷ `IsOntoBooleanMat(`*mat*`)` (property)

Returns: `true` or `false`.

A boolean matrix is TOTAL if there is at least one entry in every row is `true`. Similarly, a boolean matrix is ONTO if at least one entry in every column is `true`.

Example
```
gap> x := BooleanMat([[1, 0, 0, 1],
>                    [1, 0, 1, 1],
>                    [1, 1, 1, 0],
```

```
>                      [0, 1, 1, 0]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [1, 0, 1, 1], [1, 1, 1, 0],
  [0, 1, 1, 0]])
gap> IsTotalBooleanMat(x);
true
gap> IsOntoBooleanMat(x);
true
gap> x := BooleanMat([[1, 0, 0, 1],
>                      [1, 0, 1, 0],
>                      [0, 0, 0, 0],
>                      [0, 1, 1, 0]]);
Matrix(IsBooleanMat, [[1, 0, 0, 1], [1, 0, 1, 0], [0, 0, 0, 0],
  [0, 1, 1, 0]])
gap> IsTotalBooleanMat(x);
false
gap> IsOntoBooleanMat(x);
true
```

5.3.15 IsPartialOrderBooleanMat

▷ `IsPartialOrderBooleanMat(mat)` (property)

Returns: `true` or `false`.

A boolean matrix is a PARTIAL ORDER if it is reflexive, transitive, and anti-symmetric.

Example
```
gap> Number(FullBooleanMatMonoid(3), IsPartialOrderBooleanMat);
19
```

5.3.16 IsEquivalenceBooleanMat

▷ `IsEquivalenceBooleanMat(mat)` (property)

Returns: `true` or `false`.

A boolean matrix is an EQUIVALENCE if it is reflexive, transitive, and symmetric.

Example
```
gap> Number(FullBooleanMatMonoid(3), IsEquivalenceBooleanMat);
5
gap> Bell(3);
5
```

5.4 Matrices over finite fields

In this section we describe the operations, properties, and attributes in Semigroups specifically for matrices over finite fields. These are in addition to those given elsewhere in this chapter for arbitrary matrices over semirings.

5.4.1 NewMatrixOverFiniteField (for a filter, a field, an integer, and a list)

▷ `NewMatrixOverFiniteField(filt, F, rows)` (operation)

Returns: a new matrix object.

Creates a new *n*-by-*n* matrix over the finite field *F* with constructing filter *filt*. The matrix itself is given by a list *rows* of rows. Currently the only possible value for *filt* is IsPlistMatrixOverFiniteFieldRep.

Example
```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,
>                                  GF(4),
>                                  Z(4) * [[1, 0], [0, 1]]);
Matrix(GF(2^2), [[Z(2^2), 0*Z(2)], [0*Z(2), Z(2^2)]])
gap> y := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,
>                                  GF(4),
>                                  []);
Matrix(GF(2^2), [])
```

5.4.2 IdentityMatrixOverFiniteField (for a finite field and a pos int)

▷ IdentityMatrixOverFiniteField(*F*, *n*) (operation)
▷ IdentityMatrixOverFiniteField(*mat*, *n*) (operation)

Given a finite field *F* and a positive integer *n*, this operation returns an *n*-by-*n* identity matrix with entries in the finite field *F*. If instead the first argument is an *n*-by-*n* matrix *mat* whose BaseDomain (5.4.7) is a finite field *F*, then IdentityMatrixOverFiniteField(*mat*, *n*) returns the same as IdentityMatrixOverFiniteField(*F*, *n*).

Example
```
gap> x := NewIdentityMatrixOverFiniteField(
>                   IsPlistMatrixOverFiniteFieldRep, GF(4), 2);
Matrix(GF(2^2), [[Z(2)^0, 0*Z(2)], [0*Z(2), Z(2)^0]])
gap> y := NewZeroMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,
>                                      GF(4), 2);
Matrix(GF(2^2), [[0*Z(2), 0*Z(2)], [0*Z(2), 0*Z(2)]])
```

5.4.3 NewIdentityMatrixOverFiniteField

▷ NewIdentityMatrixOverFiniteField(*filt*, *F*, *n*) (operation)
▷ NewZeroMatrixOverFiniteField(*filt*, *F*, *n*) (operation)

Creates a new *n*-by-*n* zero or identity matrix with entries in the finite field *F*.

Example
```
gap> x := NewIdentityMatrixOverFiniteField(
>                   IsPlistMatrixOverFiniteFieldRep, GF(4), 2);
Matrix(GF(2^2), [[Z(2)^0, 0*Z(2)], [0*Z(2), Z(2)^0]])
gap> y := NewZeroMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,
>                                      GF(4), 2);
Matrix(GF(2^2), [[0*Z(2), 0*Z(2)], [0*Z(2), 0*Z(2)]])
```

5.4.4 RowSpaceBasis (for a matrix over finite field)

▷ RowSpaceBasis(*m*) (attribute)
▷ RowSpaceTransformation(*m*) (attribute)

▷ `RowSpaceTransformationInv(m)` (attribute)

To compute the value of any of the above attributes, a canonical basis for the row space of `m` is computed along with an invertible matrix `RowSpaceTransformation` such that `m * RowSpaceTransformation(m) = RowSpaceBasis(m)`. `RowSpaceTransformationInv(m)` is the inverse of `RowSpaceTransformation(m)`.

Example
```
gap> x := Matrix(GF(4), Z(4) ^ 0 * [[1, 1, 0], [0, 1, 1], [1, 1, 1]]);
Matrix(GF(2^2), [[Z(2)^0, Z(2)^0, 0*Z(2)], [0*Z(2), Z(2)^0, Z(2)^0],
  [Z(2)^0, Z(2)^0, Z(2)^0]])
gap> RowSpaceBasis(x);
<rowbasis of rank 3 over GF(2^2)>
gap> RowSpaceTransformation(x);
[ [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ]
```

5.4.5 RowRank (for a matrix over finite field)

▷ `RowRank(m)` (attribute)

Returns: Length of a basis of the row space of `m`.

Example
```
gap> x := Matrix(GF(5), Z(5) ^ 0 * [[1, 1, 0], [0, 0, 0], [1, 1, 1]]);
Matrix(GF(5), [[Z(5)^0, Z(5)^0, 0*Z(5)], [0*Z(5), 0*Z(5), 0*Z(5)],
  [Z(5)^0, Z(5)^0, Z(5)^0]])
gap> RowRank(x);
2
```

5.4.6 RightInverse (for a matrix over finite field)

▷ `RightInverse(m)` (attribute)

▷ `LeftInverse(m)` (attribute)

Returns: A matrix over a finite field.

These attributes contain a semigroup left-inverse, and a semigroup right-inverse of the matrix `m` respectively.

Example
```
gap> x := Matrix(GF(4), Z(4) ^ 0 * [[1, 1, 0], [0, 0, 0], [1, 1, 1]]);
Matrix(GF(2^2), [[Z(2)^0, Z(2)^0, 0*Z(2)], [0*Z(2), 0*Z(2), 0*Z(2)],
  [Z(2)^0, Z(2)^0, Z(2)^0]])
gap> LeftInverse(x);
Matrix(GF(2^2), [[Z(2)^0, Z(2)^0, 0*Z(2)], [0*Z(2), 0*Z(2), 0*Z(2)],
  [Z(2)^0, 0*Z(2), Z(2)^0]])
gap> Display(LeftInverse(x) * x);
Z(2)^0 Z(2)^0 0*Z(2)
0*Z(2) 0*Z(2) 0*Z(2)
0*Z(2) 0*Z(2) Z(2)^0
```

5.4.7 BaseDomain (for a matrix over finite field)

▷ `BaseDomain(mat)` (attribute)

Returns: If `mat` is a matrix over a finite field (in the category `IsMatrixOverSemiring` (5.1.1)), then BaseDomain returns the finite field specified at the point that *mat* was created. Every entry in the matrix *mat* belongs to BaseDomain(*mat*).

Example
```
gap> x := Matrix(GF(5), Z(5) ^ 0 * [[1, 1, 0], [0, 0, 0], [1, 1, 1]]);
Matrix(GF(5), [[Z(5)^0, Z(5)^0, 0*Z(5)], [0*Z(5), 0*Z(5), 0*Z(5)],
  [Z(5)^0, Z(5)^0, Z(5)^0]])
gap> BaseDomain(x);
GF(5)
```

5.4.8 TransposedMatImmutable (for a matrix over finite field)

▷ TransposedMatImmutable(*m*) (attribute)

Returns: An immutable matrix.

This attribute contains an immutable copy of *m*. Note that matrices are immutable by default.

Example
```
gap> x := Matrix(GF(5), Z(5) ^ 0 * [[1, 1, 0], [0, 0, 0], [1, 1, 1]]);
Matrix(GF(5), [[Z(5)^0, Z(5)^0, 0*Z(5)], [0*Z(5), 0*Z(5), 0*Z(5)],
  [Z(5)^0, Z(5)^0, Z(5)^0]])
gap> TransposedMatImmutable(x);
Matrix(GF(5), [[Z(5)^0, 0*Z(5), Z(5)^0], [Z(5)^0, 0*Z(5), Z(5)^0],
  [0*Z(5), 0*Z(5), Z(5)^0]])
```

5.5 Integer Matrices

In this section we describe operations in Semigroups specifically for integer matrices. These are in addition to those given elsewhere in this chapter for arbitrary matrices over semirings. These include:

- InverseOp (5.5.1)
- IsTorsion (5.5.2)
- Order (5.5.3)

5.5.1 InverseOp (for an integer matrix)

▷ InverseOp(*mat*) (operation)

Returns: An integer matrix.

If *mat* is an integer matrix (i.e. belongs to the category IsIntegerMatrix (5.1.8)) whose inverse (if it exists) is also an integer matrix, then InverseOp returns the inverse of *mat*.

An integer matrix has an integer matrix inverse if and only if it has determinant one.

Example
```
gap> mat := Matrix(IsIntegerMatrix, [[0, 0, -1],
>                                     [0, 1, 0],
>                                     [1, 0, 0]]);;
gap> InverseOp(mat);
Matrix(IsIntegerMatrix, [[0, 0, 1], [0, 1, 0], [-1, 0, 0]])
gap> mat * InverseOp(mat) = One(mat);
true
```

5.5.2 IsTorsion (for an integer matrix)

▷ `IsTorsion(mat)` (attribute)

Returns: `true` or `false`

If *mat* is an integer matrix (i.e. belongs to the category `IsIntegerMatrix` (5.1.8)), then `IsTorsion` returns `true` if *mat* is torsion and `false` otherwise.

An integer matrix *mat* is torsion if and only if there exists an integer `n` such that *mat* to the power of `n` is equal to the identity matrix.

Example
```
gap> mat := Matrix(IsIntegerMatrix, [[0, 0, -1],
>                                    [0, 1, 0],
>                                    [1, 0, 0]]);;
gap> IsTorsion(mat);
true
gap> mat := Matrix(IsIntegerMatrix, [[0, 0, -1, 0],
>                                    [0, -1, 0, 0],
>                                    [4, 4, 2, -1],
>                                    [1, 1, 0, 3]]);;
gap> IsTorsion(mat);
false
```

5.5.3 Order

▷ `Order(mat)` (attribute)

Returns: An integer or `infinity`.

If *mat* is an integer matrix (i.e. belongs to the category `IsIntegerMatrix` (5.1.8)), then `InverseOp` returns the order of *mat*. The order of *mat* is the smallest integer power of *mat* equal to the identity. If no such integer exists, the order is equal to `infinity`.

Example
```
gap> mat := Matrix(IsIntegerMatrix, [[0, 0, -1, 0],
>                                    [0, -1, 0, 0],
>                                    [4, 4, 2, -1],
>                                    [1, 1, 0, 3]]);;
gap> Order(mat);
infinity
gap> mat := Matrix(IsIntegerMatrix, [[0, 0, -1],
>                                    [0, 1, 0],
>                                    [1, 0, 0]]);;
gap> Order(mat);
4
```

5.6 Max-plus and min-plus matrices

In this section we describe operations in Semigroups specifically for max-plus and min-plus matrices. These are in addition to those given elsewhere in this chapter for arbitrary matrices over semirings. These include:

- `InverseOp` (5.6.1)
- `RadialEigenvector` (5.6.2)

- `SpectralRadius` (5.6.3)
- `UnweightedPrecedenceDigraph` (5.6.4)

5.6.1 InverseOp

▷ `InverseOp(mat)` (operation)

Returns: A max-plus matrix.

If *mat* is an invertible max-plus matrix (i.e. belongs to the category `IsMaxPlusMatrix` (5.1.8)) and is a row permutation applied to the identity), then `InverseOp` returns the inverse of *mat*. This method is described in [Far09].

Example
```
gap> InverseOp(Matrix(IsMaxPlusMatrix, [[-infinity, -infinity, 0],
>                                        [0, -infinity, -infinity],
>                                        [-infinity, 0, -infinity]]));
Matrix(IsMaxPlusMatrix, [[-infinity, 0, -infinity],
  [-infinity, -infinity, 0], [0, -infinity, -infinity]])
```

5.6.2 RadialEigenvector

▷ `RadialEigenvector(mat)` (operation)

Returns: A vector.

If *mat* is a `n` by `n` max-plus matrix (i.e. belongs to the category `IsMaxPlusMatrix` (5.1.8)), then `RadialEigenvector` returns an eigenvector corresponding to the eigenvalue equal to the spectral radius of *mat*. This method is described in [Far09].

Example
```
gap> RadialEigenvector(Matrix(IsMaxPlusMatrix, [[0, -3], [-2, -10]]));
[ 0, -2 ]
```

5.6.3 SpectralRadius

▷ `SpectralRadius(mat)` (operation)

Returns: A rational number.

If *mat* is a max-plus matrix (i.e. belongs to the category `IsMaxPlusMatrix` (5.1.8)), then `SpectralRadius` returns the supremum of the set of eigenvalues of *mat*. This method is described in [BF92].

Example
```
gap> SpectralRadius(Matrix(IsMaxPlusMatrix, [[-infinity, 1, -4],
>                                             [2, -infinity, 0],
>                                             [0, 1, 1]]));
3/2
```

5.6.4 UnweightedPrecedenceDigraph

▷ `UnweightedPrecedenceDigraph(mat)` (operation)

Returns: A digraph.

If *mat* is a max-plus matrix (i.e. belongs to the category `IsMaxPlusMatrix` (5.1.8)), then `UnweightedPrecedenceDigraph` returns the unweighted precedence digraph corresponding to *mat*.

For an `n` by `n` matrix *mat*, the unweighted precedence digraph is defined as the digraph with `n` vertices where vertex `i` is adjacent to vertex `j` if and only if *mat*`[i][j]` is not equal to `-infinity`.

Example
```
gap> UnweightedPrecedenceDigraph(Matrix(IsMaxPlusMatrix, [[2, -2, 0],
> [-infinity, 10, -2], [-infinity, 2, 1]]));
<digraph with 3 vertices, 7 edges>
```

5.7 Matrix semigroups

In this section we describe operations and attributes in Semigroups specifically for matrix semigroups. These are in addition to those given elsewhere in this chapter for arbitrary matrices over semirings. These include:

- `IsXMatrixSemigroup` (5.7.1)
- `IsFinite` (5.7.3)
- `IsTorsion` (5.7.4)
- `NormalizeSemigroup` (5.7.5)

Random matrix semigroups can be created by using the function `RandomSemigroup` (6.6.1). We can also create a matrix semigroup isomorphic to a given semigroup by using `IsomorphismSemigroup` (6.5.1) and `AsSemigroup` (6.5.3). These functions require a filter, and accept any of the filters in Section 5.7.1.

There are corresponding functions which can be used for matrix monoids: `RandomMonoid` (6.6.1), `IsomorphismMonoid` (6.5.2), and `AsMonoid` (6.5.4). These can be used with the filters in Section 5.7.2.

5.7.1 Matrix semigroup filters

▷ `IsMatrixOverSemiringSemigroup(obj)` (Category)
▷ `IsBooleanMatSemigroup(obj)` (Category)
▷ `IsMatrixOverFiniteFieldSemigroup(obj)` (Category)
▷ `IsMaxPlusMatrixSemigroup(obj)` (Category)
▷ `IsMinPlusMatrixSemigroup(obj)` (Category)
▷ `IsTropicalMatrixSemigroup(obj)` (Category)
▷ `IsTropicalMaxPlusMatrixSemigroup(obj)` (Category)
▷ `IsTropicalMinPlusMatrixSemigroup(obj)` (Category)
▷ `IsNTPMatrixSemigroup(obj)` (Category)
▷ `IsIntegerMatrixSemigroup(obj)` (Category)

Returns: `true` or `false`.

The above are the currently supported types of matrix semigroups. For monoids see Section 5.7.2.

5.7.2 Matrix monoid filters

▷ `IsMatrixOverSemiringMonoid(obj)` (Category)
▷ `IsBooleanMatMonoid(obj)` (Category)
▷ `IsMatrixOverFiniteFieldMonoid(obj)` (Category)
▷ `IsMaxPlusMatrixMonoid(obj)` (Category)

▷ IsMinPlusMatrixMonoid(*obj*) (Category)
▷ IsTropicalMatrixMonoid(*obj*) (Category)
▷ IsTropicalMaxPlusMatrixMonoid(*obj*) (Category)
▷ IsTropicalMinPlusMatrixMonoid(*obj*) (Category)
▷ IsNTPMatrixMonoid(*obj*) (Category)
▷ IsIntegerMatrixMonoid(*obj*) (Category)

Returns: `true` or `false`.

The above are the currently supported types of matrix monoids. They correspond to the matrix semigroup types in Section 5.7.1. For matrix semigroups over finite fields there is also `IsMatrixOverFiniteFieldGroup` (5.7.7).

5.7.3 IsFinite

▷ IsFinite(*S*) (property)

Returns: `true` or `false`.

If *S* is a max-plus or min-plus matrix semigroup (i.e. belongs to the category `IsMaxPlusMatrixSemigroup` (5.7.1)), then `IsFinite` returns `true` if *S* is finite and `false` otherwise. This method is based on [Gau96] (max-plus) and [Sim78] (min-plus). For min-plus matrix semigroups, this method is only valid if each min-plus matrix in the semigroup contains only non-negative integers. Note, this method is terminating and does not involve enumerating semigroups.

Example
```
gap> IsFinite(Semigroup(Matrix(IsMaxPlusMatrix,
>                              [[0, -3],
>                               [-2, -10]])));
true
gap> IsFinite(Semigroup(Matrix(IsMaxPlusMatrix,
>                              [[1, -infinity, 2],
>                               [-2, 4, -infinity],
>                               [1, 0, 3]])));
false
gap> IsFinite(Semigroup(Matrix(IsMinPlusMatrix,
>                              [[infinity, 0],
>                               [5, 4]])));
false
gap> IsFinite(Semigroup(Matrix(IsMinPlusMatrix,
>                              [[1, 0],
>                               [0, infinity]])));
true
```

5.7.4 IsTorsion

▷ IsTorsion(*S*) (attribute)

Returns: `true` or `false`.

If *S* is a max-plus matrix semigroup (i.e. belongs to the category `IsMaxPlusMatrixSemigroup` (5.7.1)), then `IsTorsion` returns `true` if *S* is torsion and `false` otherwise. This method is based on [Gau96] and draws on [Bur16], [BF92] and [Far09].

Example
```
gap> IsTorsion(Semigroup(Matrix(IsMaxPlusMatrix,
>                               [[0, -3],
>                                [-2, -10]])));
```

```
true
gap> IsTorsion(Semigroup(Matrix(IsMaxPlusMatrix,
>                               [[1, -infinity, 2],
>                                [-2, 4, -infinity],
>                                [1, 0, 3]])));
false
```

5.7.5 NormalizeSemigroup

▷ NormalizeSemigroup(*S*) (operation)

Returns: A semigroup.

This method applies to max-plus matrix semigroups (i.e. those belonging to the category IsMaxPlusMatrixSemigroup (5.7.1)) that are finitely generated, such that the spectral radius of the matrix equal to the sum of the generators (with respect to the max-plus semiring) is zero. NormalizeSemigroup returns a semigroup of matrices all with strictly non-positive entries. Note that the output is isomorphic to a min-plus matrix semigroup. This method is based on [Gau96] and [Bur16].

Example
```
gap> NormalizeSemigroup(Semigroup(Matrix(IsMaxPlusMatrix,
>                                         [[0, -3],
>                                          [-2, -10]])));
<commutative semigroup of 2x2 max-plus matrices with 1 generator>
```

5.7.6 Matrix groups

For interfacing the semigroups code with GAP's library code for matrix groups, the Semigroups package implements matrix groups that delegate to the GAP library. These functions include:

- IsMatrixOverFiniteFieldGroup (5.7.7)
- \^ (5.7.8)
- IsomorphismMatrixGroup (5.7.9)
- AsMatrixGroup (5.7.10)

This type of group only applies to matrices over finite fields (see IsMatrixOverFiniteFieldSemigroup (5.7.1)).

5.7.7 IsMatrixOverFiniteFieldGroup

▷ IsMatrixOverFiniteFieldGroup(*G*) (filter)

Returns: true or false.

A *matrix group* is simply a group of invertible matrices over a finite field. An object in Semigroups is a matrix group if it satisfies IsGroup (**Reference: IsGroup**) and IsMatrixOverFiniteFieldCollection (5.1.9).

Example
```
gap> x := Matrix(GF(4), Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]]);;
gap> G := Group(x);
<group of 3x3 matrices over GF(2^2) with 1 generator>
```

```
gap> IsMatrixOverFiniteFieldGroup(G);
true
gap> G := Group(Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]]);
Group([ <an immutable 3x3 matrix over GF2> ])
gap> IsGroup(G);
true
gap> IsMatrixOverFiniteFieldGroup(G);
false
```

5.7.8 \^ (for a matrix over finite field group and matrix over finite field)

▷ `\^(G, mat)` (operation)

Returns: A matrix group over a finite field.

The arguments of this operation, *G* and *mat*, must be categories `IsMatrixOverFiniteFieldGroup` (5.7.7) and `IsMatrixOverFiniteField` (5.1.8). If *G* consists of `d` by `d` matrices over `GF(q)` and *mat* is a `d` by `d` matrix over `GF(q)`, then *G* `^` *mat* returns the conjugate of *G* by *mat* inside `GL(d, q)`.

Example

```
gap> x := Matrix(GF(4), Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]]);;
gap> y := Matrix(GF(4), Z(4) ^ 0 * [[1, 0, 0], [1, 0, 1], [1, 1, 1]]);;
gap> G := Group(x);
<group of 3x3 matrices over GF(2^2) with 1 generator>
gap> G ^ y;
<group of 3x3 matrices over GF(2^2) with 1 generator>
```

5.7.9 IsomorphismMatrixGroup

▷ `IsomorphismMatrixGroup(G)` (attribute)

Returns: An isomorphism.

If *G* belongs to the category `IsMatrixOverFiniteFieldGroup` (5.7.7), then `IsomorphismMatrixGroup` returns an isomorphism from *G* into a group consisting of GAP library matrices.

Example

```
gap> x := Matrix(GF(4), Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]]);;
gap> G := Group(x);;
gap> iso := IsomorphismMatrixGroup(G);;
gap> Source(iso); Range(iso);
<group of 3x3 matrices over GF(2^2) with 1 generator>
Group(
[
  [ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] ])
```

5.7.10 AsMatrixGroup

▷ `AsMatrixGroup(G)` (attribute)

Returns: A group of GAP library matrices over a finite field.

Returns the image of the isomorphism returned by 5.7.9.

Example

```
gap> x := Matrix(GF(4), Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]]);;
gap> G := Group(x);
<group of 3x3 matrices over GF(2^2) with 1 generator>
gap> AsMatrixGroup(G);
Group(
[
  [ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] ])
```

Chapter 6

Creating semigroups and monoids

In this chapter we describe the various ways that semigroups and monoids can be created in Semigroups, and the options that are available at the time of creation.

6.1 Underlying algorithms and related representations

Computing the Green's structure of a semigroup is fundamental to almost every other algorithm for semigroups. There are two fundamental algorithms in the Semigroups package for computing the Green's structure of a semigroup, which are described in the next two subsections.

6.1.1 Acting semigroups

The first type of fundamental algorithms are those described in [EENMP15], which when applied to a semigroup with relatively large subgroups are the most efficient methods in the Semigroups package. For example, the complexity of computing, say, the size of a transformation semigroup that happens to be a group, is the same as the complexity of the Schreier-Sims Algorithm (polynomial in the number of points acted on by the transformations) for a permutation group.

In theory, these algorithms can be applied to compute any subsemigroup of a regular semigroup; but so far in the Semigroups package they are only implemented for semigroups of: transformations (see (**Reference: Transformations**)), partial permutations (see (**Reference: Partial permutations**)), bipartitions (see Chapter 3), subsemigroups of regular Rees 0-matrix semigroups over permutation groups (see Chapter (**Reference: Rees Matrix Semigroups**)), and matrices over a finite field (see Section 5.4).

We refer to semigroups to which the algorithms in [EENMP15] can be applied as *acting semigroups*, and such semigroups belong to the category `IsActingSemigroup` (6.1.3).

If you know *a priori* that the semigroup you want to compute is large and $\mathscr{J}$-trivial, then you can disable the special methods for acting semigroups when you create the semigroups; see Section 6.3 for more details.

It is harder for the acting semigroup algorithms to compute Green's $\mathscr{L}$- and $\mathscr{H}$-classes of a transformation semigroup and the methods used to compute with Green's $\mathscr{R}$- and $\mathscr{D}$-classes are the most efficient in Semigroups. Thus, if you are computing with a transformation semigroup, wherever possible, it is advisable to use the commands relating to Green's $\mathscr{R}$- or $\mathscr{D}$-classes rather than those relating to Green's $\mathscr{L}$- or $\mathscr{H}$-classes. No such difficulties are present when computing with the other types of acting semigroups in Semigroups.

There are methods in Semigroups for computing individual Green's classes of an acting semigroup without computing the entire data structure of the underlying semigroup; see `GreensRClassOfElementNC` (12.1.3). It is also possible to compute the $\mathscr{R}$-classes, the number of elements and test membership in a semigroup without computing all the elements; see, for example, `GreensRClasses` (12.1.4), `RClassReps` (12.1.5), `IteratorOfRClassReps` (12.2.1), `IteratorOfRClasses` (12.2.2), or `NrRClasses` (12.1.9). This may be useful if you want to study a very large semigroup where computing all the elements of the semigroup is not feasible.

6.1.2 Enumerable semigroups

The second fundamental algorithm is the Froidure-Pin Algorithm [FP97]. The Semigroups package contains two implementations of the Froidure-Pin Algorithm: one in the libsemigroups C++ library and the other within the Semigroups package kernel module.

Both implementations outperform the algorithms for acting semigroups when applied to semigroups with small (trivial) subgroups. This method is also used to determine the structure of a semigroup when the algorithms described in [EENMP15] do not apply. It is possible to specify which methods should be applied to a given semigroup; see Section 6.3.

We refer to semigroups to which the Froidure-Pin Algorithm can be applied in GAP as *enumerable semigroups*, and such semigroups should belong to the representation `IsEnumerableSemigroupRep` (6.1.4). Every acting semigroup in Semigroups is an enumerable semigroup since the acting semigroup data structure does not readily admit computation of certain properties or attributes.

Currently, the libsemigroups implementation of the Froidure-Pin Algorithm can be applied to semigroups consisting of the following types of elements: transformations (see (**Reference: Transformations**)), partial permutations (see (**Reference: Partial permutations**)), bipartitions (see Chapter 3), partitioned binary relations (see Chapter 4) as defined in [MM11]; and matrices over the following semirings:

- the *Boolean semiring* $\{0,1\}$ where $0+0=0$, $0+1=1+1=1+0=1$, and $1\cdot 0=0\cdot 0=0\cdot 1=0$
- finite fields;
- the *max-plus semiring* of natural numbers and negative infinity $\mathbb{N}\cup\{-\infty\}$ with operations max and plus;
- the *min-plus semiring* of natural numbers and infinity $\mathbb{N}\cup\{\infty\}$ with operations min and plus;
- the *tropical max-plus semiring* $\{-\infty,0,1,\ldots,t\}$ for some threshold t with operations max and plus;
- the *tropical min-plus semiring* $\{0,1,\ldots,t,\infty\}$ for some threshold t with operations min and plus;
- the semiring $\mathbb{N}_{t,p}=\{0,1,\ldots,t,t+1,\ldots,t+p-1\}$ for some threshold t and period p under addition and multiplication modulo the congruence $t=t+p$.

(see Chapter 5).

The version of the Froidure-Pin Algorithm [FP97] written in C within the Semigroups package kernel module can be used to compute any other semigroup in GAP with representation `IsEnumerableSemigroupRep` (6.1.4). In theory, any finite semigroup can be computed using this algorithm. However, the condition that the semigroup has representation `IsEnumerableSemigroupRep`

(6.1.4) is imposed to avoid this method being used when it is inappropriate (such as for finitely presented semigroups which happen to be finite). If implementing a new type of semigroup in GAP, then simply do

Example
```
InstallTrueMethod(IsGeneratorsOfEnumerableSemigroup,
                  MyNewSemigroupType);
```

to make your new semigroup type `MyNewSemigroupType` use this version of the Froidure-Pin Algorithm.

Mostly due to the way that GAP handles memory, this implementation is approximately 4 times slower than the implementation in libsemigroups. This version of the Froidure-Pin Algorithm is included because it applies to a wider class of semigroups than those currently implemented in libsemigroups and it is more straightforward to extend the classes of semigroup to which it applies. From a usage perspective there is no difference between those enumerable semigroups that are representable in libsemigroups and those that are not, except that the latter has superior performance.

6.1.3 IsActingSemigroup

▷ `IsActingSemigroup(obj)` (Category)

Returns: `true` or `false`.

Every acting semigroup, as defined in Section 6.1.1, belongs to this category.

Example
```
gap> S := Semigroup(Transformation([1, 3, 2]));;
gap> IsActingSemigroup(S);
true
gap> IsEnumerableSemigroupRep(S);
true
gap> S := FreeSemigroup(3);;
gap> IsActingSemigroup(S);
false
```

6.1.4 IsEnumerableSemigroupRep

▷ `IsEnumerableSemigroupRep(obj)` (Representation)

Returns: `true` or `false`.

Every semigroup with this representation can have the Froidure-Pin algorithm applied to it; see Section 6.1.2 for more details.

Basic operations for enumerable semigroups are `AsListCanonical` (13.1.1), `EnumeratorCanonical` (13.1.1), `IteratorCanonical` (13.1.1), `PositionCanonical` (13.1.2), `Enumerate` (13.1.3), and `IsFullyEnumerated` (13.1.4).

Example
```
gap> S := Semigroup(Transformation([1, 3, 2]));;
gap> IsEnumerableSemigroupRep(S);
true
gap> S := FreeSemigroup(3);;
gap> IsEnumerableSemigroupRep(S);
false
```

6.2 Semigroups represented by generators

6.2.1 InverseMonoidByGenerators

▷ `InverseMonoidByGenerators(coll[, opts])` (operation)
▷ `InverseSemigroupByGenerators(coll[, opts])` (operation)

Returns: An inverse monoid or semigroup.

If *coll* is a collection satisfying `IsGeneratorsOfInverseSemigroup`, then `InverseMonoidByGenerators` and `InverseSemigroupByGenerators` return the inverse monoid and semigroup generated by *coll*, respectively.

If present, the optional second argument *opts* should be a record containing the values of the options for the semigroup being created, as described in Section 6.3.

6.3 Options when creating semigroups

When using any of the functions:

- `InverseSemigroup` (**Reference: InverseSemigroup**),
- `InverseMonoid` (**Reference: InverseMonoid**),
- `Semigroup` (**Reference: Semigroup**),
- `Monoid` (**Reference: Monoid**),
- `SemigroupByGenerators` (**Reference: SemigroupByGenerators**),
- `MonoidByGenerators` (**Reference: MonoidByGenerators**),
- `ClosureSemigroup` (6.4.1),
- `ClosureMonoid` (6.4.1),
- `ClosureInverseSemigroup` (6.4.1),
- `ClosureInverseMonoid` (6.4.1),
- `SemigroupIdeal` (7.1.1)

a record can be given as an optional final argument. The components of this record specify the values of certain options for the semigroup being created. A list of these options and their default values is given below.

Assume that *S* is the semigroup created by one of the functions given above and that either: *S* is generated by a collection *gens*; or *S* is an ideal of such a semigroup.

`acting`
: this component should be `true` or `false`. Roughly speaking, there are two types of methods in the Semigroups package: those for semigroups which have to be fully enumerated, and those for semigroups that do not; see Section 1.1. In order for a semigroup to use the latter methods in Semigroups it must satisfy `IsActingSemigroup` (6.1.3). By default any semigroup or monoid of transformations, partial permutations, Rees 0-matrix elements, or bipartitions satisfies `IsActingSemigroup`.

There are cases (such as when it is known *a priori* that the semigroup is $\mathscr{D}$-trivial), when it might be preferable to use the methods that involve fully enumerating a semigroup. In other words, it might be desirable to disable the more sophisticated methods for acting semigroups. If this is the case, then the value of this component can be set `false` when the semigroup is created. Following this none of the special methods for acting semigroup will be used to compute anything about the semigroup.

`regular`

this component should be `true` or `false`. If it is known *a priori* that the semigroup `S` being created is a regular semigroup, then this component can be set to `true`. In this case, `S` knows it is a regular semigroup and can take advantage of the methods for regular semigroups in Semigroups. It is usually much more efficient to compute with a regular semigroup that to compute with a non-regular semigroup.

If this option is set to `true` when the semigroup being defined is NOT regular, then the results might be unpredictable.

The default value for this option is `false`.

`hashlen`

this component should be a positive integer, which roughly specifies the lengths of the hash tables used internally by Semigroups. Semigroups uses hash tables in several fundamental methods. The lengths of these tables are a compromise between performance and memory usage; larger tables provide better performance for large computations but use more memory. Note that it is unlikely that you will need to specify this option unless you find that GAP runs out of memory unexpectedly or that the performance of Semigroups is poorer than expected. If you find that GAP runs out of memory unexpectedly, or you plan to do a large number of computations with relatively small semigroups (say with tens of thousands of elements), then you might consider setting `hashlen` to be less than the default value of `25013` for each of these semigroups. If you find that the performance of Semigroups is unexpectedly poor, or you plan to do a computation with a very large semigroup (say, more than 10 million elements), then you might consider setting `hashlen` to be greater than the default value of `25013`.

You might find it useful to set the info level of the info class `InfoOrb` to 2 or higher since this will indicate when hash tables used by Semigroups are being grown; see `SetInfoLevel` (**Reference: InfoLevel**).

`small`

if this component is set to `true`, then Semigroups will compute a small subset of *`gens`* that generates *`S`* at the time that *`S`* is created. This will increase the amount of time required to create *`S`* substantially, but may decrease the amount of time required for subsequent calculations with *`S`*. If this component is set to `false`, then Semigroups will return the semigroup generated by *`gens`* without modifying *`gens`*. The default value for this component is `false`.

This option is ignored when passed to `ClosureSemigroup` (6.4.1) or `ClosureInverseSemigroup` (6.4.1).

`cong_by_ker_trace_threshold`

this should be a positive integer, which specifies a semigroup size. If *`S`* is a semigroup with inverse op, and *`S`* has a size greater than or equal to this threshold, then any congruence defined on it may use the "kernel and trace" method to perform calculations. If its size is less than the threshold, then other methods will be used instead. The "kernel and trace" method has better complexity than the generic method, but has large overheads which make it a poor choice for small semigroups. The default value for this component is `10 ^ 5`. See Section 16.7 for more information about the "kernel and trace" method.

`report`
: this component should be either `true` or `false`. If this component is set to `true`, then some additional information will be provided during computations performed by the libsemigroups C++ library.

`batch_size`
: this component should be a positive integer. If *S* is a semigroup with representation `IsEnumerableSemigroupRep` (6.1.4), then when certain computations are performed with *S* using the libsemigroups C++ library, then the computations will be executed in batches of size at least `batch_size`. This value of this component changes the performance of the libsemigroups C++ library — you may wish to tweak this parameter if you experience sub-optimal performance.

`nr_threads`
: this component should be a positive integer. This number sets the maximum number of threads that can be used by computations in the libsemigroups C++ library.

Example

```
gap> S := Semigroup(Transformation([1, 2, 3, 3]),
>                   rec(hashlen := 100003, small := false));
<commutative transformation semigroup of degree 4 with 1 generator>
```

The default values of the options described above are stored in a global variable named `SEMIGROUPS.DefaultOptionsRec` (6.3.1). If you want to change the default values of these options for a single GAP session, then you can simply redefine the value in GAP. For example, to change the option `small` from the default value of *false* use:

Example

```
gap> SEMIGROUPS.DefaultOptionsRec.small := true;
true
```

If you want to change the default values of the options stored in `SEMIGROUPS.DefaultOptionsRec` (6.3.1) for all GAP sessions, then you can edit these values in the file `semigroups/gap/options.g`.

6.3.1 SEMIGROUPS.DefaultOptionsRec

▷ `SEMIGROUPS.DefaultOptionsRec` (global variable)

This global variable is a record whose components contain the default values of certain options for semigroups. A description of these options is given above in Section 6.3.

The value of `SEMIGROUPS.DefaultOptionsRec` is defined in the file `semigroups/gap/options.g` as:

Example

```
rec(acting := true, regular := false, small := false,
    hashlen := rec(L := 25013, M := 6257, S := 251),
    report := false, batch_size := 8192, nr_threads := 4,
    cong_by_ker_trace_threshold := 10 ^ 5)
```

6.4 New semigroups from old

6.4.1 ClosureSemigroup

▷ `ClosureSemigroup(S, coll[, opts])` (operation)
▷ `ClosureMonoid(S, coll[, opts])` (operation)
▷ `ClosureInverseSemigroup(S, coll[, opts])` (operation)
▷ `ClosureInverseMonoid(S, coll[, opts])` (operation)

Returns: A semigroup, monoid, inverse semigroup, or inverse monoid.

These operations return the semigroup, monoid, inverse semigroup or inverse monoid generated by the argument *S* and the collection of elements *coll* after removing duplicates and elements from *coll* that are already in *S*. In most cases, the new semigroup knows at least as much information about its structure as was already known about that of *S*.

When `X` is any of `Semigroup` (**Reference: Semigroup**), `Monoid` (**Reference: Monoid**), `InverseSemigroup` (**Reference: InverseSemigroup**), or `InverseMonoid` (**Reference: InverseMonoid**), the argument *S* of the operation `ClosureX` must belong to the category `IsX`, and `ClosureX(`*S*`, `*coll*`)` returns an object in the category `IsX` such that

Example
```
ClosureX(S, coll) = X(S, coll);
```

but may have fewer generators, if for example, *coll* contains a duplicates or elements already known to belong to *S*.

For example, the argument *S* of `ClosureInverseSemigroup` must be an inverse semigroup in the category `IsInverseSemigroup` (**Reference: IsInverseSemigroup**). `ClosureInverseSemigroup(`*S*`, `*coll*`)` returns an inverse semigroup which is equal to `InverseSemigroup(`*S*`, `*coll*`)`.

If present, the optional third argument *opts* should be a record containing the values of the options for the semigroup being created as described in Section 6.3.

Example
```
gap> gens := [Transformation([2, 6, 7, 2, 6, 1, 1, 5]),
>             Transformation([3, 8, 1, 4, 5, 6, 7, 1]),
>             Transformation([4, 3, 2, 7, 7, 6, 6, 5]),
>             Transformation([7, 1, 7, 4, 2, 5, 6, 3])];;
gap> S := Monoid(gens[1]);;
gap> for x in gens do
>      S := ClosureSemigroup(S, x);
>    od;
gap> S;
<transformation monoid of degree 8 with 4 generators>
gap> Size(S);
233606
gap> S := Monoid(PartialPerm([1]));
<trivial partial perm group of rank 1 with 1 generator>
gap> T := ClosureMonoid(S, [PartialPerm([2 .. 5])]);
<partial perm monoid of rank 5 with 2 generators>
gap> One(T);
<identity partial perm on [ 1, 2, 3, 4, 5 ]>
gap> T := ClosureSemigroup(S, [PartialPerm([2 .. 5])]);
<partial perm semigroup of rank 4 with 2 generators>
gap> One(T);
```

```
fail
gap> ClosureInverseMonoid(DualSymmetricInverseMonoid(3),
>                          DClass(DualSymmetricInverseMonoid(3),
>                                 IdentityBipartition(3)));
<inverse block bijection monoid of degree 3 with 3 generators>
gap> S := InverseSemigroup(Bipartition([[1, -1, -3], [2, 3, -2]]),
>                          Bipartition([[1, -3], [2, -2], [3, -1]]));;
gap> T := ClosureInverseSemigroup(S, DClass(PartitionMonoid(3),
> IdentityBipartition(3)));
<inverse block bijection semigroup of degree 3 with 3 generators>
gap> T := ClosureInverseSemigroup(T, [T.1, T.1, T.1]);
<inverse block bijection semigroup of degree 3 with 3 generators>
gap> S := InverseMonoid([
> PartialPerm([5, 9, 10, 0, 6, 3, 8, 4, 0]),
> PartialPerm([10, 7, 0, 8, 0, 0, 5, 9, 1])]);;
gap> x := PartialPerm([
> 5, 1, 7, 3, 10, 0, 2, 12, 0, 14, 11, 0, 16, 0, 0, 0, 0, 6, 9, 15]);
[4,3,7,2,1,5,10,14][8,12][13,16][18,6][19,9][20,15](11)
gap> S := ClosureInverseSemigroup(S, x);
<inverse partial perm semigroup of rank 19 with 4 generators>
gap> Size(S);
9744
gap> T := Idempotents(SymmetricInverseSemigroup(10));;
gap> S := ClosureInverseSemigroup(S, T);
<inverse partial perm semigroup of rank 19 with 14 generators>
```

6.4.2 SubsemigroupByProperty (for a semigroup and function)

▷ SubsemigroupByProperty(*S*, *func*) (operation)

▷ SubsemigroupByProperty(*S*, *func*, *limit*) (operation)

Returns: A semigroup.

SubsemigroupByProperty returns the subsemigroup of the semigroup *S* generated by those elements of *S* fulfilling *func* (which should be a function returning true or false).

If no elements of *S* fulfil *func*, then fail is returned.

If the optional third argument *limit* is present and a positive integer, then once the subsemigroup has at least *limit* elements the computation stops.

Example

```
gap> func := function(x)
>      local n;
>      n := DegreeOfTransformation(x);
>      return 1 ^ x <> 1 and ForAll([1 .. n], y -> y = 1 or y ^ x = y);
>    end;
function( x ) ... end
gap> T := SubsemigroupByProperty(FullTransformationSemigroup(3), func);
<transformation semigroup of size 2, degree 3 with 2 generators>
gap> T := SubsemigroupByProperty(FullTransformationSemigroup(4), func);
<transformation semigroup of size 3, degree 4 with 3 generators>
gap> T := SubsemigroupByProperty(FullTransformationSemigroup(5), func);
<transformation semigroup of size 4, degree 5 with 4 generators>
```

6.4.3 InverseSubsemigroupByProperty

▷ `InverseSubsemigroupByProperty(S, func)` (operation)

Returns: An inverse semigroup.

`InverseSubsemigroupByProperty` returns the inverse subsemigroup of the inverse semigroup *S* generated by those elements of *S* fulfilling *func* (which should be a function returning `true` or `false`).

If no elements of *S* fulfil *func*, then `fail` is returned.

If the optional third argument *limit* is present and a positive integer, then once the subsemigroup has at least *limit* elements the computation stops.

Example

```
gap> IsIsometry := function(f)
> local n, i, j, k, l;
>  n := RankOfPartialPerm(f);
>  for i in [1 .. n - 1] do
>    k := DomainOfPartialPerm(f)[i];
>    for j in [i + 1 .. n] do
>      l := DomainOfPartialPerm(f)[j];
>      if not AbsInt(k ^ f - l ^ f) = AbsInt(k - l) then
>        return false;
>      fi;
>    od;
>  od;
>  return true;
> end;;
gap> S := InverseSubsemigroupByProperty(SymmetricInverseSemigroup(5),
> IsIsometry);;
gap> Size(S);
142
```

6.4.4 DirectProduct

▷ `DirectProduct(S[, T, ...])` (function)

▷ `DirectProductOp(list, S)` (operation)

Returns: A transformation semigroup.

The function `DirectProduct` takes an arbitrary positive number of transformation semigroups and returns another transformation semigroup isomorphic to their direct product. The operation `DirectProductOp` is included for consistency with the GAP library (see `DirectProductOp` (**Reference: DirectProductOp**)). It takes exactly two arguments, namely a non-empty list `list` of transformation semigroups and one of these semigroups, `S`.

Example

```
gap> S := Semigroup(Transformation([2, 1]));;
gap> T := Semigroup(Transformation([1, 2, 3, 3, 3]));;
gap> DP := DirectProduct(S, T);
<commutative transformation semigroup of degree 7 with 2 generators>
gap> Elements(DP);
[ Transformation( [ 1, 2, 3, 4, 5, 5, 5 ] ),
  Transformation( [ 2, 1, 3, 4, 5, 5, 5 ] ) ]
gap> S := Monoid([Transformation([2, 4, 3, 4]),
>                 Transformation([3, 3, 2, 3, 3])]);;
```

```
gap> T := Semigroup([Transformation([3, 5, 4, 2, 6, 3])]);;
gap> DP := DirectProduct(S, T);
<transformation semigroup of degree 11 with 4 generators>
gap> Size(DP);
35
```

6.5 Changing the representation of a semigroup

The Semigroups package provides two convenient constructors `IsomorphismSemigroup` (6.5.1) and `IsomorphismMonoid` (6.5.2) for changing the representation of a given semigroup or monoid. These methods can be used to find an isomorphism from any semigroup to a semigroup of any other type, provided such an isomorphism exists.

Note that at present neither `IsomorphismSemigroup` (6.5.1) nor `IsomorphismMonoid` (6.5.2) can be used to determine whether two given semigroups, or monoids, are isomorphic.

Some methods for `IsomorphismSemigroup` (6.5.1) and `IsomorphismMonoid` (6.5.2) are based on methods for the GAP library operations:

- `IsomorphismReesMatrixSemigroup` (**Reference: IsomorphismReesMatrixSemigroup**),
- `AntiIsomorphismTransformationSemigroup` (**Reference: AntiIsomorphismTransformationSemigroup**),
- `IsomorphismTransformationSemigroup` (**Reference: IsomorphismTransformationSemigroup**) and `IsomorphismTransformationMonoid` (**Reference: IsomorphismTransformationMonoid**),
- `IsomorphismPartialPermSemigroup` (**Reference: IsomorphismPartialPermSemigroup**) and `IsomorphismPartialPermMonoid` (**Reference: IsomorphismPartialPermMonoid**),
- `IsomorphismFpSemigroup` (**Reference: IsomorphismFpSemigroup**) and `IsomorphismFpMonoid`.

The operation `IsomorphismMonoid` (6.5.2) can be used to return an isomorphism from a semigroup which is mathematically a monoid (but does not below to the category of monoids in GAP `IsMonoid` (**Reference: IsMonoid**)) into a monoid. This is the primary purpose of the operation `IsomorphismMonoid` (6.5.2). Either `IsomorphismSemigroup` (6.5.1) or `IsomorphismMonoid` (6.5.2) can be used to change the representation of a monoid, but only the latter is guaranteed to return an object in the category of monoids.

Example

```
gap> S := Monoid(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>               Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> AsSemigroup(IsBooleanMatSemigroup, S);
<monoid of 10x10 boolean matrices with 2 generators>
gap> AsMonoid(IsBooleanMatMonoid, S);
<monoid of 10x10 boolean matrices with 2 generators>
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> AsSemigroup(IsBooleanMatSemigroup, S);
<semigroup of 10x10 boolean matrices with 2 generators>
gap> AsMonoid(IsBooleanMatMonoid, S);
```

```
<monoid of 8x8 boolean matrices with 2 generators>
gap> M := Monoid([
> Bipartition([[1, -3], [2, 3, 6], [4, 7, -6], [5, -8], [8, -4, -5],
>              [-1], [-2], [-7]]),
> Bipartition([[1, 3, -6], [2, -8], [4, 8, -1], [5], [6, -3, -4],
>              [7], [-2], [-5], [-7]]),
> Bipartition([[1, 2, 4, -3, -7, -8], [3, 5, 6, 8, -4, -6],
>              [7, -1, -2, -5]])]);;
gap> AsMonoid(IsPBRMonoid, M);
<pbr monoid of size 163, degree 163 with 3 generators>
gap> AsSemigroup(IsPBRSemigroup, M);
<pbr semigroup of size 163, degree 8 with 4 generators>
```

There are some further methods in Semigroups for obtaining an isomorphism from a Rees matrix, or 0-matrix, semigroup to another such semigroup with particular properties; `RMSNormalization` (6.5.7) and `RZMSNormalization` (6.5.6).

6.5.1 IsomorphismSemigroup

▷ `IsomorphismSemigroup(filt, S)` (operation)

Returns: An isomorphism of semigroups.

`IsomorphismSemigroup` can be used to find an isomorphism from a given semigroup to a semigroup of another type, provided such an isomorphism exists.

The first argument *filt* must be of the form `IsXSemigroup`, for example, `IsTransformationSemigroup` (**Reference: IsTransformationSemigroup**), `IsFpSemigroup` (**Reference: IsFpSemigroup**), and `IsPBRSemigroup` (4.6.1) are some possible values for *filt*. Note that *filt* should not be of the form `IsXMonoid`; see `IsomorphismMonoid` (6.5.2). The second argument *S* should be a semigroup.

`IsomorphismSemigroup` returns an isomorphism from *S* to a semigroup *T* of the type described by *filt*, if such an isomorphism exists. More precisely, if `T` is the range of the returned isomorphism, then *filt*`(T)` will return `true`. For example, if *filt* is `IsTransformationSemigroup`, then the range of the returned isomorphism will be a transformation semigroup.

An error is returned if there is no isomorphism from *S* to a semigroup satisfying *filt*. For example, there is no method for `IsomorphismSemigroup` when *filt* is, say, `IsReesMatrixSemigroup` (**Reference: IsReesMatrixSemigroup**) and when *S* is a non-simple semigroup. Similarly, there is no method when *filt* is `IsPartialPermSemigroup` (**Reference: IsPartialPermSemigroup**) and when *S* is a non-inverse semigroup.

In some cases, if no better method is installed, `IsomorphismSemigroup` returns an isomorphism found by composing an isomorphism from *S* to a transformation semigroup `T`, and an isomorphism from `T` to a semigroup of type *filt*.

Note that if the argument *S* belongs to the category of monoids `IsMonoid` (**Reference: IsMonoid**), then `IsomorphismSemigroup` will often, but not always, return a monoid isomorphism.

Example

```
gap> S := Semigroup([
> Bipartition([
>   [1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
> Bipartition([
>   [1, -4], [2, 3, 4, 5], [6], [-1, -6], [-2, -3], [-5]])]);
<bipartition semigroup of degree 6 with 2 generators>
```

```
gap> IsomorphismSemigroup(IsTransformationSemigroup, S);
MappingByFunction( <bipartition semigroup of size 11, degree 6 with 2
 generators>, <transformation semigroup of size 11, degree 12 with 2
 generators>, function( x ) ... end, function( x ) ... end )
gap> IsomorphismSemigroup(IsBooleanMatSemigroup, S);
MappingByFunction( <bipartition semigroup of size 11, degree 6 with 2
 generators>, <semigroup of size 11, 12x12 boolean matrices with 2
 generators>, function( x ) ... end, function( x ) ... end )
gap> IsomorphismSemigroup(IsFpSemigroup, S);
MappingByFunction( <bipartition semigroup of size 11, degree 6 with 2
 generators>, <fp semigroup on the generators
[ s1, s2 ]>, function( x ) ... end, function( x ) ... end )
gap> S := InverseSemigroup([
> PartialPerm([1, 2, 3, 6, 8, 10],
>             [2, 6, 7, 9, 1, 5]),
> PartialPerm([1, 2, 3, 4, 6, 7, 8, 10],
>             [3, 8, 1, 9, 4, 10, 5, 6])]);;
gap> IsomorphismSemigroup(IsBipartitionSemigroup, S);
MappingByFunction( <inverse partial perm semigroup of rank 10 with 2
 generators>, <inverse bipartition semigroup of degree 10 with 2
 generators>, function( x ) ... end, <Operation "AsPartialPerm"> )
gap> S := SymmetricInverseMonoid(4);
<symmetric inverse monoid of degree 4>
gap> IsomorphismSemigroup(IsBlockBijectionSemigroup, S);
MappingByFunction( <symmetric inverse monoid of degree 4>,
<inverse block bijection monoid of degree 5 with 3 generators>
 , function( x ) ... end, function( x ) ... end )
gap> Size(Range(last));
209
gap> S := Semigroup([
> PartialPerm([3, 1]), PartialPerm([1, 3, 4])]);;
gap> IsomorphismSemigroup(IsBlockBijectionSemigroup, S);
MappingByFunction( <partial perm semigroup of rank 3 with 2
 generators>, <block bijection semigroup of degree 5 with 2
 generators>, function( x ) ... end, function( x ) ... end )
```

6.5.2 IsomorphismMonoid

▷ `IsomorphismMonoid(`*filt*`, `*S*`)` (operation)

Returns: An isomorphism of monoids.

`IsomorphismMonoid` can be used to find an isomorphism from a given semigroup which is mathematically a monoid (but might not belong to the category of monoids in GAP) to a monoid, provided such an isomorphism exists.

The first argument *filt* must be of the form `IsXMonoid`, for example, `IsTransformationMonoid` (**Reference: IsTransformationMonoid**), `IsFpMonoid` (**Reference: IsFpMonoid**), and `IsBipartitionMonoid` (3.8.1) are some possible values for *filt*. Note that *filt* should not be of the form `IsXSemigroup`; see `IsomorphismSemigroup` (6.5.1). The second argument *S* should be a semigroup which is mathematically a monoid but which may or may not belong to the category `IsMonoid` (**Reference: IsMonoid**) of monoids in GAP, i.e. *S* must satisfy `IsMonoidAsSemigroup` (14.1.12).

`IsomorphismMonoid` returns a monoid isomorphism from *S* to a semigroup *T* of the type described by *filt*, if such an isomorphism exists. In this context, a *monoid isomorphism* is a semigroup isomorphism that maps the `MultiplicativeNeutralElement` (**Reference: MultiplicativeNeutralElement**) of *S* to the `One` (**Reference: One**) of *T*. If `T` is the range of the returned isomorphism, then *filt*`(T)` will return `true`. For example, if *filt* is `IsTransformationMonoid`, then the range of the returned isomorphism will be a transformation monoid.

An error is returned if there is no isomorphism from *S* to a monoid satisfying *filt*. For example, there is no method for `IsomorphismMonoid` when *filt* is, say, `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**) and when *S* is a not 0-simple. Similarly, there is no method when *filt* is `IsPartialPermMonoid` (**Reference: IsPartialPermMonoid**) and when *S* is a non-inverse monoid.

In some cases, if no better method is installed, `IsomorphismMonoid` returns an isomorphism found by composing an isomorphism from *S* to a transformation monoid `T`, and an isomorphism from `T` to a monoid of type *filt*.

Example
```
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));
<transformation semigroup of degree 10 with 2 generators>
gap> IsomorphismMonoid(IsTransformationMonoid, S);
MappingByFunction( <transformation semigroup of degree 10 with 2
 generators>, <transformation monoid of degree 8 with 2 generators>
 , function( x ) ... end, function( x ) ... end )
gap> IsomorphismMonoid(IsBooleanMatMonoid, S);
MappingByFunction( <transformation semigroup of degree 10 with 2
 generators>, <monoid of 8x8 boolean matrices with 2 generators>
 , function( x ) ... end, function( x ) ... end )
gap> IsomorphismMonoid(IsFpMonoid, S);
MappingByFunction( <transformation semigroup of degree 10 with 2
 generators>, <fp monoid on the generators
[ m1, m2 ]>, function( x ) ... end, function( x ) ... end )
```

6.5.3 AsSemigroup

▷ `AsSemigroup(`*filt*`,` *S*`)` (operation)

Returns: A semigroup.

`AsSemigroup(`*filt*`,` *S*`)` is just shorthand for `Range(IsomorphismSemigroup(`*filt*`,` *S*`))`, when *S* is a semigroup; see `IsomorphismSemigroup` (6.5.1) for more details.

Note that if the argument *S* belongs to the category of monoids `IsMonoid` (**Reference: IsMonoid**), then `AsSemigroup` will often, but not always, return a monoid. A monoid is not returned if there is not a good monoid isomorphism from *S* to a monoid of the required type, but there is a good semigroup isomorphism.

If it is not possible to convert the semigroup *S* to a semigroup of type *filt*, then an error is given.

Example
```
gap> S := Semigroup([
> Bipartition([
>   [1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
> Bipartition([
>   [1, -4], [2, 3, 4, 5], [6], [-1, -6], [-2, -3], [-5]])]);
<bipartition semigroup of degree 6 with 2 generators>
```

```
gap> AsSemigroup(IsTransformationSemigroup, S);
<transformation semigroup of size 11, degree 12 with 2 generators>
gap> S := Semigroup([
> Bipartition([
>   [1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
> Bipartition([
>   [1, -4], [2, 3, 4, 5], [6], [-1, -6], [-2, -3], [-5]])]);
<bipartition semigroup of degree 6 with 2 generators>
gap> AsSemigroup(IsTransformationSemigroup, S);
<transformation semigroup of size 11, degree 12 with 2 generators>
gap> T := Semigroup(Transformation([2, 2, 3]),
>                   Transformation([3, 1, 3]));
<transformation semigroup of degree 3 with 2 generators>
gap> S := AsSemigroup(IsMatrixOverFiniteFieldSemigroup, GF(5), T);
<semigroup of 3x3 matrices over GF(5) with 2 generators>
gap> Size(S);
5
```

6.5.4 AsMonoid

▷ `AsMonoid([filt, ]S)` (operation)

Returns: A monoid or `fail`.

`AsMonoid(filt, S)` is just shorthand for `Range(IsomorphismMonoid(filt, S))`, when *S* is a semigroup or monoid; see `IsomorphismMonoid` (6.5.2) for more details.

If the first argument *filt* is omitted and the semigroup *S* is mathematically a monoid which does not belong to the category of monoids in GAP, then `AsMonoid` returns a monoid (in the category of monoids) isomorphic to *S* and of the same type as *S*. If *S* is already in the category of monoids and the first argument *filt* is omitted, then *S* is returned.

If the first argument *filt* is omitted and the semigroup *S* is not a monoid, i.e. it does not satisfy `IsMonoidAsSemigroup` (14.1.12), then `fail` is returned.

Example
```
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> AsMonoid(S);
<transformation monoid of degree 8 with 2 generators>
gap> AsSemigroup(IsBooleanMatSemigroup, S);
<semigroup of 10x10 boolean matrices with 2 generators>
gap> AsMonoid(IsBooleanMatMonoid, S);
<monoid of 8x8 boolean matrices with 2 generators>
gap> S := Monoid(Bipartition([[1, -1, -3], [2, 3], [-2]]),
>                Bipartition([[1, -1], [2, 3, -3], [-2]]));
<bipartition monoid of degree 3 with 2 generators>
gap> AsMonoid(IsTransformationMonoid, S);
<transformation monoid of size 3, degree 3 with 2 generators>
gap> AsMonoid(S);
<bipartition monoid of size 3, degree 3 with 2 generators>
```

6.5.5 IsomorphismPermGroup

▷ `IsomorphismPermGroup(S)` (attribute)

Returns: An isomorphism.

If the semigroup `S` is mathematically a group, so that it satisfies `IsGroupAsSemigroup` (14.1.6), then `IsomorphismPermGroup` returns an isomorphism to a permutation group.

If `S` is not a group then an error is given.

See also `IsomorphismPermGroup` (**Reference: IsomorphismPermGroup**).

Example
```
gap> S := Semigroup(Transformation([2, 2, 3, 4, 6, 8, 5, 5]),
>                   Transformation([3, 3, 8, 2, 5, 6, 4, 4]));;
gap> IsGroupAsSemigroup(S);
true
gap> Range(IsomorphismPermGroup(S));
Group([ (5,6,8), (2,3,8,4) ])
gap> StructureDescription(Range(IsomorphismPermGroup(S)));
"S6"
gap> S := Range(IsomorphismPartialPermSemigroup(SymmetricGroup(4)));
<partial perm group of size 24, rank 4 with 2 generators>
gap> IsomorphismPermGroup(S);
MappingByFunction( <partial perm group of size 24, rank 4 with
  2 generators>, Group([ (1,2,3,4), (1,
2) ]), <Attribute "AsPermutation">, function( x ) ... end )
gap> G := GroupOfUnits(PartitionMonoid(4));
<block bijection group of degree 4 with 2 generators>
gap> StructureDescription(G);
"S4"
gap> iso := IsomorphismPermGroup(G);;
gap> RespectsMultiplication(iso);
true
gap> inv := InverseGeneralMapping(iso);;
gap> ForAll(G, x -> (x ^ iso) ^ inv = x);
true
gap> ForAll(G, x -> ForAll(G, y -> (x * y) ^ iso = x ^ iso * y ^ iso));
true
```

6.5.6 RZMSNormalization

▷ `RZMSNormalization(R)` (attribute)

Returns: An isomorphism.

If `R` is a Rees 0-matrix semigroup $M^0[I,T,\Lambda;P]$ then `RZMSNormalization` returns an isomorphism from `R` to a *normalized* Rees 0-matrix semigroup $S = M^0[I,T,\Lambda;Q]$. The structure matrix Q is obtained by *normalizing* the matrix P (see `Matrix` (**Reference: Matrix**)) and has the following properties:

- The matrix Q is in block diagonal form, and the blocks are ordered by decreasing size along the leading diagonal (the size of a block is defined to be the number of rows it contains multiplied by the number of columns it contains).

  If the index sets I and Λ are partitioned into k parts according to the `RZMSConnectedComponents` (13.14.2) of S, giving a disjoint union $I = I_1 \cup \ldots \cup I_k$ and

$\Lambda = \Lambda_1 \cup \ldots \cup \Lambda_k$, then the rth block corresponds to the sub-matrix Q_r of Q defined by I_r and Λ_r.

- The first non-zero entry in a row occurs no sooner than the first non-zero entry in any previous row.
- The first non-zero entry in a column occurs no sooner than the first non-zero entry in any previous column.
- The previous two items imply that if the matrix P has any rows/columns consisting entirely of zeroes, then these will become the final rows/columns of Q.

Furthermore, if T is a group (i.e. a semigroup for which `IsGroupAsSemigroup` (14.1.6) returns `true`), then the non-zero entries of the structure matrix Q are chosen such that the following hold:

- The first non-zero entry of every row and every column is equal to the identity of T.
- For each r, let Q_r be the sub-matrix of Q defined by I_r and Λ_r (as above), and let T_r be the subsemigroup of T generated by the non-zero entries of Q_r. Then the idempotent generated subsemigroup of S is equal to:
  - $\bigcup_{r=1}^{k} M^0[I_r, T_r, \Lambda_r, Q_r]$, where the zeroes of these Rees 0-matrix semigroups are all identified with the zero of S.

The normalization given by `RZMSNormalization` is based on Theorem 2 of [Gra68] and is sometimes called *Graham normal form*. Note that isomorphic Rees 0-matrix semigroups can have normalizations which are not equal.

Example

```
gap> R := ReesZeroMatrixSemigroup(Group(()),
> [[0, (), 0],
>  [(), 0, 0],
>  [0, 0, ()]]);
<Rees 0-matrix semigroup 3x3 over Group(())>
gap> iso := RZMSNormalization(R);
MappingByFunction( <Rees 0-matrix semigroup 3x3 over Group(())>,
<Rees 0-matrix semigroup 3x3 over Group(())>
 , function( x ) ... end, function( x ) ... end )
gap> S := Range(iso);
<Rees 0-matrix semigroup 3x3 over Group(())>
gap> Matrix(S);
[ [ (), 0, 0 ], [ 0, (), 0 ], [ 0, 0, () ] ]
gap> R := ReesZeroMatrixSemigroup(SymmetricGroup(4),
> [[0, 0, 0, (1, 3, 2)],
>  [(2, 3), 0, 0, 0],
>  [0, 0, (1, 3), (1, 2)],
>  [0, (4, 1, 2, 3), 0, 0]]);
<Rees 0-matrix semigroup 4x4 over Sym( [ 1 .. 4 ] )>
gap> S := Range(RZMSNormalization(R));
<Rees 0-matrix semigroup 4x4 over Sym( [ 1 .. 4 ] )>
gap> Matrix(S);
[ [ (), (), 0, 0 ], [ 0, (), 0, 0 ], [ 0, 0, (), 0 ], [ 0, 0, 0, () ]
 ]
```

6.5.7 RMSNormalization

▷ RMSNormalization(*R*) (attribute)

Returns: An isomorphism.

If *R* is a Rees matrix semigroup over a group G (i.e. a semigroup for which IsGroupAsSemigroup (14.1.6) returns true), then RMSNormalization returns an isomorphism from *R* to a *normalized* Rees matrix semigroup S over G.

The semigroup S is normalized in the sense that the first entry of each row and column of the Matrix (**Reference: Matrix**) of S is the identity element of G.

Example
```
gap> R := ReesMatrixSemigroup(SymmetricGroup(4),
> [[(1, 2), (2, 4, 3), (2, 1, 4)],
>  [(1, 3, 2), (1, 2)(3, 4), ()],
>  [(2, 3), (1, 3, 2, 4), (2, 3)]]);
<Rees matrix semigroup 3x3 over Sym( [ 1 .. 4 ] )>
gap> iso := RMSNormalization(R);
MappingByFunction( <Rees matrix semigroup 3x3 over Sym( [ 1 .. 4 ] )>
 , <Rees matrix semigroup 3x3 over Sym( [ 1 .. 4 ] )>
 , function( x ) ... end, function( x ) ... end )
gap> S := Range(iso);
<Rees matrix semigroup 3x3 over Sym( [ 1 .. 4 ] )>
gap> Matrix(S);
[ [ (), (), () ], [ (), (1,2), (1,4,2,3) ], [ (), (1,4,2,3), (2,4) ] ]
```

6.6 Random semigroups

6.6.1 RandomSemigroup

▷ RandomSemigroup(*arg...*) (function)

▷ RandomMonoid(*arg...*) (function)

▷ RandomInverseSemigroup(*arg...*) (function)

▷ RandomInverseMonoid(*arg...*) (function)

Returns: A semigroup.

The operations described in this section can be used to generate semigroups, in some sense, at random. There is no guarantee given about the distribution of these semigroups, and this is only intended as a means of generating semigroups for testing and other similar purposes.

Roughly speaking, the arguments of RandomSemigroup are a filter specifying the type of the semigroup to be returned, together with some further parameters that describe some attributes of the semigroup to be returned. For instance, we may want to specify the number of generators, and, say, the degree, or dimension, of the elements, where appropriate. The arguments of RandomMonoid, RandomInverseSemigroup, and RandomInverseMonoid are analogous.

If no arguments are specified, then they are all chosen at random, for a truly random experience.

The first argument, if present, should be a filter *filter*. For RandomSemigroup and RandomInverseSemigroup the filter *filter* must be of the form IsXSemigroup. For example, IsTransformationSemigroup (**Reference: IsTransformationSemigroup**), IsFpSemigroup (**Reference: IsFpSemigroup**), and IsPBRSemigroup (4.6.1) are some possible values for *filter*. For RandomMonoid and RandomInverseMonoid the argument *filter* must be of the form IsXMonoid; such as IsBipartitionMonoid (3.8.1) or IsBooleanMatMonoid (5.7.2).

Suppose that the first argument *filter* is IsFpSemigroup (**Reference: IsFpSemigroup**). Then the only other arguments that can be specified is (and this argument is also optional):

number of generators

The second argument, if present, should be a positive integer *m* indicating the number of generators that the semigroup should have. If the second argument *m* is not specified, then a number is selected at random.

If *filter* is a filter such as IsTransformationSemigroup (**Reference: IsTransformationSemigroup**) or IsIntegerMatrixSemigroup (5.7.1), then a further argument can be specified:

degree / dimension

The third argument, if present, should be a positive integer *n*, which specifies the degree or dimension of the generators. For example, if the first argument *filter* is IsTransformationSemigroup, then the value of this argument is the degree of the transformations in the returned semigroup; or if *filter* is IsMatrixOverFiniteFieldSemigroup, then this argument is the dimension of the matrices in the returned semigroup.

If *filter* is IsTropicalMaxPlusMatrixSemigroup (5.7.1), for example, then a fourth argument can be given (or not!):

threshold

The fourth argument, if present, should be a positive integer *t*, which specifies the threshold of the semiring over which the matrices in the returned semigroup are defined.

You get the idea, the error messages are self-explanatory, and RandomSemigroup works for most of the type of semigroups defined in GAP.

RandomMonoid is similar to RandomSemigroup except it returns a monoid. Likewise, RandomInverseSemigroup and RandomInverseMonoid return inverse semigroups and monoids, respectively.

Example

```
gap> RandomSemigroup();
<semigroup of 10x10 max-plus matrices with 12 generators>
gap> RandomMonoid(IsTransformationMonoid);
<transformation monoid of degree 9 with 7 generators>
gap> RandomMonoid(IsPartialPermMonoid, 2);
<partial perm monoid of rank 17 with 2 generators>
gap> RandomMonoid(IsPartialPermMonoid, 2, 3);
<partial perm monoid of rank 3 with 2 generators>
gap> RandomInverseSemigroup(IsTropicalMinPlusMatrixSemigroup);
<semigroup of 6x6 tropical min-plus matrices with 14 generators>
gap> RandomInverseSemigroup(IsTropicalMinPlusMatrixSemigroup, 1);
<semigroup of 6x6 tropical min-plus matrices with 14 generators>
gap> RandomSemigroup(IsTropicalMinPlusMatrixSemigroup, 2);
<semigroup of 11x11 tropical min-plus matrices with 2 generators>
gap> RandomSemigroup(IsTropicalMinPlusMatrixSemigroup, 2, 1);
<semigroup of 1x1 tropical min-plus matrices with 2 generators>
gap> RandomSemigroup(IsTropicalMinPlusMatrixSemigroup, 2, 1, 3);
gap> last.1;
Matrix(IsTropicalMinPlusMatrix, [[infinity]], 3)
gap> RandomSemigroup(IsNTPMatrixSemigroup, 2, 1, 3, 4);
<semigroup of 1x1 ntp matrices with 2 generators>
```

```
gap> last.1;
Matrix(IsNTPMatrix, [[2]], 3, 4)
gap> RandomSemigroup(IsReesMatrixSemigroup, 2, 2);
<Rees matrix semigroup 2x2 over
  <permutation group of size 659 with 1 generators>>
gap> RandomSemigroup(IsReesZeroMatrixSemigroup, 2, 2, Group((1, 2), (3, 4)));
<Rees 0-matrix semigroup 2x2 over Group([ (1,2), (3,4) ])>
gap> RandomInverseMonoid(IsMatrixOverFiniteFieldMonoid, 2, 2);
<monoid of 3x3 matrices over GF(421^4) with 3 generators>
gap> RandomInverseMonoid(IsMatrixOverFiniteFieldMonoid, 2, 2, GF(7));
<monoid of 3x3 matrices over GF(7) with 2 generators>
gap> RandomSemigroup(IsBipartitionSemigroup, 5, 5);
<bipartition semigroup of degree 5 with 5 generators>
gap> RandomMonoid(IsBipartitionMonoid, 5, 5);
<bipartition monoid of degree 5 with 5 generators>
gap> RandomSemigroup(IsBooleanMatSemigroup);
<semigroup of 3x3 boolean matrices with 18 generators>
gap> RandomMonoid(IsBooleanMatMonoid);
<monoid of 11x11 boolean matrices with 19 generators>
```

6.7 Endomorphism monoid of a digraph

6.7.1 EndomorphismMonoid (for a digraph)

▷ `EndomorphismMonoid(`*`digraph`*`)` (attribute)
▷ `EndomorphismMonoid(`*`digraph, colors`*`)` (operation)

Returns: A monoid.

An endomorphism of *digraph* is a homomorphism `DigraphHomomorphism` (**Digraphs: DigraphHomomorphism**) from *digraph* back to itself.

`EndomorphismMonoid`, called with a single argument, returns the monoid of all endomorphisms of *digraph*.

If the *colors* argument is specified, then it will return the monoid of endomorphisms which respect the given colouring. The colouring *colors* can be in one of two forms:

- A list of positive integers of size the number of vertices of *digraph*, where *colors*`[i]` is the colour of vertex `i`.
- A list of lists, such that *colors*`[i]` is a list of all vertices with colour `i`.

See also `GeneratorsOfEndomorphismMonoid` (**Digraphs: GeneratorsOfEndomorphismMonoid**). Note that the performance of `EndomorphismMonoid` may differ from that of `GeneratorsOfEndomorphismMonoid` (**Digraphs: GeneratorsOfEndomorphismMonoid**) since the former incrementally adds newly discovered endomorphisms to the monoid using `ClosureMonoid` (6.4.1).

Example
```
gap> gr := Digraph(List([1 .. 3], x -> [1 .. 3]));;
gap> EndomorphismMonoid(gr);
<transformation monoid of degree 3 with 3 generators>
gap> gr := CompleteDigraph(3);;
gap> EndomorphismMonoid(gr);
```

```
<transformation group of size 6, degree 3 with 2 generators>
gap> EndomorphismMonoid(gr, [1, 2, 2]);
<transformation group of degree 3 with 1 generator>
gap> EndomorphismMonoid(gr, [[1], [2, 3]]);
<transformation group of degree 3 with 1 generator>
```

Chapter 7

Ideals

In this chapter we describe the various ways that an ideal of a semigroup can be created and manipulated in Semigroups.

We write *ideal* to mean two-sided ideal everywhere in this chapter.

The methods in the Semigroups package apply to any ideal of a semigroup that is created using the function `SemigroupIdeal` (7.1.1) or `SemigroupIdealByGenerators`. Anything that can be calculated for a semigroup defined by a generating set can also be found for an ideal. This works particularly well for regular ideals, since such an ideal can be represented using a similar data structure to that used to represent a semigroup defined by a generating set but without the necessity to find a generating set for the ideal. Many methods for non-regular ideals rely on first finding a generating set for the ideal, which can be costly (but not nearly as costly as an exhaustive enumeration of the elements of the ideal). We plan to improve the functionality of Semigroups for non-regular ideals in the future.

7.1 Creating ideals

7.1.1 SemigroupIdeal

▷ `SemigroupIdeal(S, obj1, obj2, .., .)` (function)

Returns: An ideal of a semigroup.

If *obj1*, *obj2*, .. . are (any combination) of elements of the semigroup *S* or collections of elements of *S* (including subsemigroups and ideals of *S*), then `SemigroupIdeal` returns the 2-sided ideal of the semigroup *S* generated by the union of *obj1*, *obj2*, ...

The `Parent` (**Reference: Parent**) of the ideal returned by this function is *S*.

Example
```
gap> S := SymmetricInverseMonoid(10);
<symmetric inverse monoid of degree 10>
gap> I := SemigroupIdeal(S, PartialPerm([1, 2]));
<inverse partial perm semigroup ideal of rank 10 with 1 generator>
gap> Size(I);
4151
gap> I := SemigroupIdeal(S, I, Idempotents(S));
<inverse partial perm semigroup ideal of rank 10 with 1025 generators>
```

7.2 Attributes of ideals

7.2.1 GeneratorsOfSemigroupIdeal

▷ `GeneratorsOfSemigroupIdeal(I)` (attribute)

Returns: The generators of an ideal of a semigroup.

This function returns the generators of the two-sided ideal `I`, which were used to defined `I` when it was created.

If `I` is an ideal of a semigroup, then `I` is defined to be the least 2-sided ideal of a semigroup `S` containing a set `J` of elements of `S`. The set `J` is said to *generate* `I`.

The notion of the generators of an ideal is distinct from the notion of the generators of a semigroup or monoid. In particular, the semigroup generated by the generators of an ideal is not, in general, equal to that ideal. Use `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**) to obtain a semigroup generating set for an ideal, but beware that this can be very costly.

Example
```
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 4, -1], [-2, -4], [-3]]),
> Bipartition([[1, 2, 3, -3], [4], [-1], [-2, -4]]),
> Bipartition([[1, 3, -2], [2, 4], [-1, -3, -4]]),
> Bipartition([[1], [2, 3, 4], [-1, -3, -4], [-2]]),
> Bipartition([[1], [2, 4, -2], [3, -4], [-1], [-3]]));;
gap> I := SemigroupIdeal(S, S.1 * S.2 * S.5);;
gap> GeneratorsOfSemigroupIdeal(I);
[ <bipartition: [ 1, 2, 3, 4, -4 ], [ -1 ], [ -2 ], [ -3 ]> ]
gap> I = Semigroup(GeneratorsOfSemigroupIdeal(I));
false
```

7.2.2 MinimalIdealGeneratingSet

▷ `MinimalIdealGeneratingSet(I)` (attribute)

Returns: A minimal set ideal generators of an ideal.

This function returns a minimal set of elements of the parent of the semigroup ideal `I` required to generate `I` as an ideal.

The notion of the generators of an ideal is distinct from the notion of the generators of a semigroup or monoid. In particular, the semigroup generated by the generators of an ideal is not, in general, equal to that ideal. Use `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**) to obtain a semigroup generating set for an ideal, but beware that this can be very costly.

Example
```
gap> S := Monoid([
> Bipartition([[1, 2, 3, -2], [4], [-1, -4], [-3]]),
> Bipartition([[1, 4, -2, -4], [2, -1, -3], [3]])]);;
gap> I := SemigroupIdeal(S, S);;
gap> MinimalIdealGeneratingSet(I);
[ <block bijection: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ], [ 4, -4 ]> ]
```

7.2.3 SupersemigroupOfIdeal

▷ `SupersemigroupOfIdeal(I)` (attribute)

Returns: An ideal of a semigroup.

The Parent (**Reference: Parent**) of an ideal is the semigroup in which the ideal was created, i.e. the first argument of SemigroupIdeal (7.1.1) or SemigroupIdealByGenerators. This function returns the semigroup containing the generators of the semigroup (i.e. GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**)) which are used to compute the ideal.

For a regular semigroup ideal, SupersemigroupOfIdeal will always be the top most semigroup used to create any of the predecessors of the current ideal. For example, if S is a semigroup, I is a regular ideal of S, and J is an ideal of I, then Parent(J) is I and SupersemigroupOfIdeal(J) is S. This is to avoid computing a generating set for I, in this example, which is expensive and unnecessary since I is regular (in which case the Green's relations of I are just restrictions of the Green's relations on S).

If S is a semigroup, I is a non-regular ideal of S, J is an ideal of I, then SupersemigroupOfIdeal(J) is I, since we currently have to use GeneratorsOfSemigroup(I) to compute anything about I other than its size and membership.

Example

```
gap> S := FullTransformationSemigroup(8);
<full transformation monoid of degree 8>
gap> x := Transformation([2, 6, 7, 2, 6, 1, 1, 5]);;
gap> D := DClass(S, x);
<Green's D-class: Transformation( [ 6, 3, 4, 6, 3, 5, 5, 1 ] )>
gap> R := PrincipalFactor(D);
<Rees 0-matrix semigroup 1050x56 over Group([ (2,8,7,4,3), (3,4) ])>
gap> S := Semigroup(List([1 .. 10], x -> Random(R)));
<subsemigroup of 1050x56 Rees 0-matrix semigroup with 10 generators>
gap> I := SemigroupIdeal(S, MultiplicativeZero(S));
<regular Rees 0-matrix semigroup ideal with 1 generator>
gap> SupersemigroupOfIdeal(I);
<subsemigroup of 1050x56 Rees 0-matrix semigroup with 10 generators>
gap> J := SemigroupIdeal(I, Representative(MinimalDClass(S)));
<regular Rees 0-matrix semigroup ideal with 1 generator>
gap> Parent(J) = I;
true
gap> SupersemigroupOfIdeal(J) = I;
false
```

Chapter 8

Standard examples

In this chapter we describe some standard examples of semigroups which are available in the Semigroups package.

8.1 Transformation semigroups

In this section, we describe the operations in Semigroups that can be used to create transformation semigroups belonging to several standard classes of example. See (**Reference: Transformations**) for more information about transformations.

8.1.1 CatalanMonoid

▷ `CatalanMonoid(n)` (operation)

Returns: A transformation monoid.

If `n` is a positive integer, then this operation returns the Catalan monoid of degree `n`. The *Catalan monoid* is the semigroup of the order-preserving and order-decreasing transformations of `[1 .. n]` with the usual ordering.

The Catalan monoid is generated by the `n - 1` transformations f_i:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & i & i+1 & i+2 & \cdots & n \\ 1 & 2 & 3 & \cdots & i & i & i+2 & \cdots & n \end{pmatrix},$$

where $i = 1, \ldots, n-1$ and has size equal to the nth Catalan number.

Example

```
gap> S := CatalanMonoid(9);
<transformation monoid of degree 9 with 8 generators>
gap> Size(S);
4862
```

8.1.2 EndomorphismsPartition

▷ `EndomorphismsPartition(list)` (operation)

Returns: A transformation monoid.

If `list` is a list of positive integers, then `EndomorphismsPartition` returns a monoid of endomorphisms preserving a partition of `[1 .. Sum(list)]` with a part of length `list[i]` for every `i`.

For example, if *list* = [1, 2, 3], then EndomorphismsPartition returns the monoid of endomorphisms of the partition [[1], [2, 3], [4, 5, 6]].

If f is a transformation of [1 .. n], then it is an ENDOMORPHISM of a partition P on [1 .. n] if (i, j) in P implies that (i ^ f, j ^ f) is in P.

EndomorphismsPartition returns a monoid with a minimal size generating set, as described in [ABMS14].

Example
```
gap> S := EndomorphismsPartition([3, 3, 3]);
<transformation semigroup of degree 9 with 4 generators>
gap> Size(S);
531441
```

8.1.3 PartialTransformationMonoid

▷ PartialTransformationMonoid(*n*) (operation)

Returns: A transformation monoid.

If *n* is a positive integer, then this function returns a semigroup of transformations on *n* + 1 points which is isomorphic to the semigroup consisting of all partial transformation on *n* points. This monoid has (*n* + 1) ^ *n* elements.

Example
```
gap> PartialTransformationMonoid(8);
<regular transformation monoid of degree 9 with 4 generators>
gap> Size(last);
43046721
```

8.1.4 SingularTransformationSemigroup

▷ SingularTransformationSemigroup(*n*) (operation)

▷ SingularTransformationMonoid(*n*) (operation)

Returns: The semigroup of non-invertible transformations.

If *n* is a integer greater than 1, then this function returns the semigroup of non-invertible transformations, which is generated by the *n*(*n* - 1) idempotents of degree *n* and rank *n* - 1 and has $n^n - n!$ elements.

Example
```
gap> S := SingularTransformationSemigroup(5);
<regular transformation semigroup ideal of degree 5 with 1 generator>
gap> Size(S);
3005
```

8.1.5 Semigroups of order-preserving transformations

▷ OrderEndomorphisms(*n*) (operation)

▷ SingularOrderEndomorphisms(*n*) (operation)

▷ OrderAntiEndomorphisms(*n*) (operation)

▷ PartialOrderEndomorphisms(*n*) (operation)

▷ PartialOrderAntiEndomorphisms(*n*) (operation)

Returns: A semigroup of transformations related to a linear order.

`OrderEndomorphisms(n)`

`OrderEndomorphisms(n)` returns the monoid of transformations that preserve the usual order on $\{1,2,\ldots,n\}$, where `n` is a positive integer. `OrderEndomorphisms(n)` is generated by the `n + 1` transformations:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 1 & 1 & 2 & \cdots & n-2 & n-1 \end{pmatrix}, \qquad \begin{pmatrix} 1 & 2 & \cdots & i-1 & i & i+1 & i+2 & \cdots & n \\ 1 & 2 & \cdots & i-1 & i+1 & i+1 & i+2 & \cdots & n \end{pmatrix}$$

where $i = 0,\ldots,n-1$, and has $\binom{2n-1}{n-1}$ elements.

`SingularOrderEndomorphisms(n)`

`SingularOrderEndomorphisms(n)` returns the ideal of `OrderEndomorphisms(n)` consisting of the non-invertible elements, when `n` is at least 2. The only invertible element in `OrderEndomorphisms(n)` is the identity transformation. Therefore `SingularOrderEndomorphisms(n)` has $\binom{2n-1}{n-1}-1$ elements.

`OrderAntiEndomorphisms(n)`

`OrderAntiEndomorphisms(n)` returns the monoid of transformations that preserve or reverse the usual order on $\{1,2,\ldots,n\}$, where `n` is a positive integer. `OrderAntiEndomorphisms(n)` is generated by the generators of `OrderEndomorphisms(n)` along with the bijective transformation that reverses the order on $\{1,2,\ldots,n\}$. The monoid `OrderAntiEndomorphisms(n)` has $\binom{2n-1}{n-1}-n$ elements.

`PartialOrderEndomorphisms(n)`

`PartialOrderEndomorphisms(n)` returns a monoid of transformations on `n + 1` points that is isomorphic to the monoid consisting of all partial transformations that preserve the usual order on $\{1,2,\ldots,n\}$.

`PartialOrderAntiEndomorphisms(n)`

`PartialAntiOrderEndomorphisms(n)` returns a monoid of transformations on `n + 1` points that is isomorphic to the monoid consisting of all partial transformations that preserve or reverse the usual order on $\{1,2,\ldots,n\}$.

Example

```
gap> S := OrderEndomorphisms(5);
<regular transformation monoid of degree 5 with 5 generators>
gap> IsIdempotentGenerated(S);
true
gap> Size(S) = Binomial(2 * 5 - 1, 5 - 1);
true
gap> Difference(S, SingularOrderEndomorphisms(5));
[ IdentityTransformation ]
gap> SingularOrderEndomorphisms(10);
<regular transformation semigroup ideal of degree 10 with 1 generator>
gap> T := OrderAntiEndomorphisms(4);
<regular transformation monoid of degree 4 with 5 generators>
gap> Transformation([4, 2, 2, 1]) in T;
true
gap> U := PartialOrderEndomorphisms(6);
<regular transformation monoid of degree 7 with 12 generators>
gap> V := PartialOrderAntiEndomorphisms(6);
<regular transformation monoid of degree 7 with 13 generators>
```

```
gap> IsSubsemigroup(V, U);
true
```

8.2 Semigroups of partial permutations

In this section, we describe the operations in Semigroups that can be used to create semigroups of partial permutations belonging to several standard classes of example. See (**Reference: Partial permutations**) for more information about partial permutations.

8.2.1 MunnSemigroup

▷ `MunnSemigroup(S)` (attribute)

Returns: The Munn semigroup of a semilattice.

If S is a semilattice, then `MunnSemigroup` returns the inverse semigroup of partial permutations of isomorphisms of principal ideals of S; called the *Munn semigroup* of S.

This function was written jointly by J. D. Mitchell, Yann Péresse (St Andrews), Yanhui Wang (York).

Example

```
gap> S := InverseSemigroup([
> PartialPerm([1, 2, 3, 4, 5, 6, 7, 10], [4, 6, 7, 3, 8, 2, 9, 5]),
> PartialPerm([1, 2, 7, 9], [5, 6, 4, 3])]);
<inverse partial perm semigroup of rank 10 with 2 generators>
gap> T := IdempotentGeneratedSubsemigroup(S);;
gap> M := MunnSemigroup(T);
<inverse partial perm semigroup of rank 60 with 7 generators>
gap> NrIdempotents(M);
60
gap> NrIdempotents(S);
60
```

8.2.2 RookMonoid

▷ `RookMonoid(n)` (operation)

Returns: An inverse monoid of partial permutations.

`RookMonoid` is a synonym for `SymmetricInverseMonoid` (**Reference: SymmetricInverseMonoid**).

Example

```
gap> S := RookMonoid(4);
<symmetric inverse monoid of degree 4>
gap> S = SymmetricInverseMonoid(4);
true
```

8.2.3 Inverse monoids of order-preserving partial permutations

▷ `POI(n)` (operation)

▷ `PODI(n)` (operation)

▷ `POPI(n)` (operation)

▷ PORI(n) (operation)

Returns: An inverse monoid of partial permutations related to a linear order.

POI(n)

POI(n) returns the inverse monoid of partial permutations that preserve the usual order on $\{1,2,\ldots,n\}$, where n is a positive integer. POI(n) is generated by the n partial permutations:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ - & 1 & 2 & \cdots & n-1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 & \cdots & i-1 & i & i+1 & i+2 & \cdots & n \\ 1 & 2 & \cdots & i-1 & i+1 & - & i+2 & \cdots & n \end{pmatrix}$$

where $i = 1,\ldots,n-1$, and has $\binom{2n}{n}$ elements.

PODI(n)

PODI(n) returns the inverse monoid of partial permutations that preserve or reverse the usual order on $\{1,2,\ldots,n\}$, where n is a positive integer. PODI(n) is generated by the generators of POI(n), along with the permutation that reverses the usual order on $\{1,2,\ldots,n\}$. PODI(n) has $\binom{2n}{n} - n^2 - 1$ elements.

POPI(n)

POPI(n) returns the inverse monoid of partial permutations that preserve the orientation of $\{1,2,\ldots,n\}$, where n is a positive integer. POPI(n) is generated by the partial permutations:

$$\begin{pmatrix} 1 & 2 & \cdots & n-1 & n \\ 2 & 3 & \cdots & n & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 & \cdots & n-2 & n-1 & n \\ 1 & 2 & \cdots & n-2 & n & - \end{pmatrix},$$

and has $1 + \frac{n}{2}\binom{2n}{n}$ elements.

PORI(n)

PORI(n) returns the inverse monoid of partial permutations that preserve or reverse the orientation of $\{1,2,\ldots,n\}$, where n is a positive integer. PORI(n) is generated by the generators of POPI(n), along with the permutation that reverses the usual order on $\{1,2,\ldots,n\}$. PORI(n) has $\frac{n}{2}\binom{2n}{n} - n(n+1)$ elements.

Example

```
gap> S := PORI(10);
<inverse partial perm monoid of rank 10 with 3 generators>
gap> S := POPI(10);
<inverse partial perm monoid of rank 10 with 2 generators>
gap> Size(S) = 1 + 5 * Binomial(20, 10);
true
gap> S := PODI(10);
<inverse partial perm monoid of rank 10 with 11 generators>
gap> S := POI(10);
<inverse partial perm monoid of rank 10 with 10 generators>
gap> Size(S) = Binomial(20, 10);
true
gap> IsSubsemigroup(PORI(10), PODI(10))
> and IsSubsemigroup(PORI(10), POPI(10))
> and IsSubsemigroup(PODI(10), POI(10))
> and IsSubsemigroup(POPI(10), POI(10));
true
```

8.3 Semigroups of bipartitions

In this section, we describe the operations in Semigroups that can be used to create bipartition semigroups belonging to several standard classes of example. See Chapter 3 for more information about bipartitions.

8.3.1 PartitionMonoid

▷ `PartitionMonoid(n)` (operation)
▷ `RookPartitionMonoid(n)` (operation)
▷ `SingularPartitionMonoid(n)` (operation)

Returns: A bipartition monoid.

If `n` is a non-negative integer, then this operation returns the partition monoid of degree `n`. The *partition monoid of degree* `n` is the monoid consisting of all the bipartitions of degree `n`.

`SingularPartitionMonoid` returns the ideal of the partition monoid consisting of the non-invertible elements (i.e. those not in the group of units), when `n` is positive.

If `n` is positive, then `RookPartitionMonoid` returns submonoid of the partition monoid of degree `n + 1` consisting of those bipartitions with `n + 1` and `-n - 1` in the same block; see [HR05], [Gro06], and [Eas16].

Example
```
gap> S := PartitionMonoid(5);
<regular bipartition *-monoid of size 115975, degree 5 with 4
 generators>
gap> Size(S);
115975
gap> T := SingularPartitionMonoid(5);
<regular bipartition *-semigroup ideal of degree 5 with 1 generator>
gap> Size(S) - Size(T) = Factorial(5);
true
gap> S := RookPartitionMonoid(5);
<regular bipartition *-monoid of degree 6 with 5 generators>
gap> Size(S);
678570
```

8.3.2 BrauerMonoid

▷ `BrauerMonoid(n)` (operation)
▷ `PartialBrauerMonoid(n)` (operation)
▷ `SingularBrauerMonoid(n)` (operation)

Returns: A bipartition monoid.

If `n` is a non-negative integer, then this operation returns the Brauer monoid of degree `n`. The *Brauer monoid* is the submonoid of the partition monoid consisiting of those bipartitions where the size of every block is 2.

`PartialBrauerMonoid` returns the partial Brauer monoid, which is the submonoid of the partition monoid consisting of those bipartitions where the size of every block is *at most* 2. The partial Brauer monoid contains the Brauer monoid as a submonoid.

`SingularBrauerMonoid` returns the ideal of the Brauer monoid consisting of the non-invertible elements (i.e. those not in the group of units), when `n` is at least 2.

Example
```
gap> S := BrauerMonoid(4);
<regular bipartition *-monoid of degree 4 with 3 generators>
gap> IsSubsemigroup(S, JonesMonoid(4));
true
gap> Size(S);
105
gap> SingularBrauerMonoid(8);
<regular bipartition *-semigroup ideal of degree 8 with 1 generator>
gap> S := PartialBrauerMonoid(3);
<regular bipartition *-monoid of degree 3 with 8 generators>
gap> IsSubsemigroup(S, BrauerMonoid(3));
true
gap> Size(S);
76
```

8.3.3 JonesMonoid

▷ `JonesMonoid(n)` (operation)
▷ `TemperleyLiebMonoid(n)` (operation)
▷ `SingularJonesMonoid(n)` (operation)

Returns: A bipartition monoid.

If `n` is a non-negative integer, then this operation returns the Jones monoid of degree `n`. The *Jones monoid* is the subsemigroup of the Brauer monoid consisting of those bipartitions that are planar; see `PlanarPartitionMonoid` (8.3.9). The Jones monoid is sometimes referred to as the TEMPERLEY-LIEB MONOID.

`SingularJonesMonoid` returns the ideal of the Jones monoid consisting of the non-invertible elements (i.e. those not in the group of units), when `n` is at least 2.

Example
```
gap> S := JonesMonoid(4);
<regular bipartition *-monoid of degree 4 with 3 generators>
gap> S = TemperleyLiebMonoid(4);
true
gap> SingularJonesMonoid(8);
<regular bipartition *-semigroup ideal of degree 8 with 1 generator>
```

8.3.4 PartialJonesMonoid

▷ `PartialJonesMonoid(n)` (operation)

Returns: A bipartition monoid.

If `n` is a non-negative integer, then `PartialJonesMonoid` returns the partial Jones monoid of degree `n`. The *partial Jones monoid* is a subsemigroup of the partial Brauer monoid. An element of the partial Brauer monoid is contained in the partial Jones monoid if the partition that it defines is finer than the partition defined by some element of the Jones monoid. In other words, an element of the partial Jones monoid can be formed from some element x of the Jones monoid by replacing some blocks `[a, b]` of x by singleton blocks `[a], [b]`.

Note that, in general, the partial Jones monoid of degree `n` is strictly contained in the Motzkin monoid of the same degree.

See `PartialBrauerMonoid` (8.3.2), `JonesMonoid` (8.3.3), and `MotzkinMonoid` (8.3.6).

```
Example
gap> S := PartialJonesMonoid(4);
<regular bipartition *-monoid of degree 4 with 7 generators>
gap> T := JonesMonoid(4);
<regular bipartition *-monoid of degree 4 with 3 generators>
gap> U := MotzkinMonoid(4);
<regular bipartition *-monoid of degree 4 with 8 generators>
gap> IsSubsemigroup(U, S);
true
gap> IsSubsemigroup(S, T);
true
gap> Size(U);
323
gap> Size(S);
143
gap> Size(T);
14
```

8.3.5 AnnularJonesMonoid

▷ `AnnularJonesMonoid(n)` (operation)

Returns: A bipartition monoid.

If `n` is a non-negative integer, then `AnnularJonesMonoid` returns the annular Jones monoid of degree `n`. The *annular Jones monoid* is the subsemigroup of the partition monoid consisting of all annular bipartitions whose blocks have size 2 (annular bipartitions are defined in Chapter 3). See `BrauerMonoid` (8.3.2).

```
Example
gap> S := AnnularJonesMonoid(4);
<regular bipartition *-monoid of degree 4 with 2 generators>
```

8.3.6 MotzkinMonoid

▷ `MotzkinMonoid(n)` (operation)

Returns: A bipartition monoid.

If `n` is a non-negative integer, then this operation returns the Motzkin monoid of degree `n`. The *Motzkin monoid* is the subsemigroup of the partial Brauer monoid consisting of those bipartitions that are planar (planar bipartitions are defined in Chapter 3).

Note that the Motzkin monoid of degree `n` contains the partial Jones monoid of degree `n`, but in general, these monoids are not equal; see `PartialJonesMonoid` (8.3.4).

```
Example
gap> S := MotzkinMonoid(4);
<regular bipartition *-monoid of degree 4 with 8 generators>
gap> T := PartialJonesMonoid(4);
<regular bipartition *-monoid of degree 4 with 7 generators>
gap> IsSubsemigroup(S, T);
true
gap> Size(S);
323
gap> Size(T);
143
```

8.3.7 DualSymmetricInverseSemigroup

▷ `DualSymmetricInverseSemigroup(n)` (operation)
▷ `DualSymmetricInverseMonoid(n)` (operation)
▷ `SingularDualSymmetricInverseMonoid(n)` (operation)
▷ `PartialDualSymmetricInverseMonoid(n)` (operation)

Returns: An inverse bipartition monoid.

If `n` is a positive integer, then the operations `DualSymmetricInverseSemigroup` and `DualSymmetricInverseMonoid` return the dual symmetric inverse monoid of degree `n`, which is the subsemigroup of the partition monoid consisting of the block bijections of degree `n`.

`SingularDualSymmetricInverseMonoid` returns the ideal of the dual symmetric inverse monoid consisting of the non-invertible elements (i.e. those not in the group of units), when `n` is at least 2.

`PartialDualSymmetricInverseMonoid` returns the submonoid of the dual symmetric inverse monoid of degree `n + 1` consisting of those block bijections with `n + 1` and `-n - 1` in the same block; see [KM11] and [KMU15].

See `IsBlockBijection` (3.5.16).

Example
```
gap> Number(PartitionMonoid(3), IsBlockBijection);
25
gap> S := DualSymmetricInverseSemigroup(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> Size(S);
25
gap> S := PartialDualSymmetricInverseMonoid(5);
<inverse block bijection monoid of degree 6 with 4 generators>
gap> Size(S);
29072
```

8.3.8 UniformBlockBijectionMonoid

▷ `UniformBlockBijectionMonoid(n)` (operation)
▷ `FactorisableDualSymmetricInverseMonoid(n)` (operation)
▷ `SingularUniformBlockBijectionMonoid(n)` (operation)
▷ `PartialUniformBlockBijectionMonoid(n)` (operation)
▷ `SingularFactorisableDualSymmetricInverseMonoid(n)` (operation)
▷ `PlanarUniformBlockBijectionMonoid(n)` (operation)
▷ `SingularPlanarUniformBlockBijectionMonoid(n)` (operation)

Returns: An inverse bipartition monoid.

If `n` is a positive integer, then this operation returns the uniform block bijection monoid of degree `n`. The *uniform block bijection monoid* is the submonoid of the partition monoid consisting of the block bijections of degree n where the number of positive integers in a block equals the number of negative integers in that block. The uniform block bijection monoid is also referred to as the *factorisable dual symmetric inverse monoid*.

`SingularUniformBlockBijectionMonoid` returns the ideal of the uniform block bijection monoid consisting of the non-invertible elements (i.e. those not in the group of units), when `n` is at least 2.

PlanarUniformBlockBijectionMonoid returns the submonoid of the uniform block bijection monoid consisting of the planar elements (i.e. those in the planar partition monoid, see PlanarPartitionMonoid (8.3.9)).

SingularPlanarUniformBlockBijectionMonoid returns the ideal of the planar uniform block bijection monoid consisting of the non-invertible elements (i.e. those not in the group of units), when *n* is at least 2.

PartialUniformBlockBijectionMonoid returns the submonoid of the uniform block bijection monoid of degree *n* + 1 consisting of those uniform block bijection with *n* + 1 and -*n* - 1 in the same block.

See IsUniformBlockBijection (3.5.17).

Example
```
gap> S := UniformBlockBijectionMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> Size(PlanarUniformBlockBijectionMonoid(8));
128
gap> S := DualSymmetricInverseMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> IsFactorisableInverseMonoid(S);
false
gap> S := UniformBlockBijectionMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> IsFactorisableInverseMonoid(S);
true
gap> S := AsSemigroup(IsBipartitionSemigroup,
>                     SymmetricInverseMonoid(5));
<inverse bipartition monoid of degree 5 with 3 generators>
gap> IsFactorisableInverseMonoid(S);
true
gap> S := PartialUniformBlockBijectionMonoid(5);
<inverse block bijection monoid of degree 6 with 4 generators>
gap> NrIdempotents(S);
203
gap> IsFactorisableInverseMonoid(S);
true
```

8.3.9 PlanarPartitionMonoid

▷ PlanarPartitionMonoid(*n*) (operation)

▷ SingularPlanarPartitionMonoid(*n*) (operation)

Returns: A bipartition monoid.

If *n* is a positive integer, then this operation returns the planar partition monoid of degree *n* which is the monoid consisting of all the planar bipartitions of degree *n* (planar bipartitions are defined in Chapter 3).

SingularPlanarPartitionMonoid returns the ideal of the planar partition monoid consisting of the non-invertible elements (i.e. those not in the group of units).

Example
```
gap> S := PlanarPartitionMonoid(5);
<regular bipartition *-monoid of degree 5 with 9 generators>
gap> Size(S);
16796
```

```
gap> T := SingularPlanarPartitionMonoid(5);
<regular bipartition *-semigroup ideal of degree 5 with 1 generator>
gap> Size(T);
16795
gap> Difference(S, T);
[ <block bijection: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ], [ 4, -4 ],
     [ 5, -5 ]> ]
```

8.3.10 ModularPartitionMonoid

▷ `ModularPartitionMonoid(m, n)` (operation)
▷ `SingularModularPartitionMonoid(m, n)` (operation)
▷ `PlanarModularPartitionMonoid(m, n)` (operation)
▷ `SingularPlanarModularPartitionMonoid(m, n)` (operation)

Returns: A bipartition monoid.

If `m` and `n` are positive integers, then this operation returns the modular-`m` partition monoid of degree `n`. The *modular-`m` partition monoid* is the submonoid of the partition monoid such that the numbers of positive and negative integers contained in each block are congruent mod `m`.

`SingularModularPartitionMonoid` returns the ideal of the modular partition monoid consisting of the non-invertible elements (i.e. those not in the group of units), when either `m = n = 1` or `m, n > 1`.

`PlanarModularPartitionMonoid` returns the submonoid of the modular-`m` partition monoid consisting of the planar elements (i.e. those in the planar partition monoid, see `PlanarPartitionMonoid` (8.3.9)).

`SingularPlanarModularPartitionMonoid` returns the ideal of the planar modular partition monoid consisting of the non-invertible elements (i.e. those not in the group of units), when either `m = n = 1` or `m, n > 1`.

Example

```
gap> S := ModularPartitionMonoid(3, 7);
<regular bipartition *-monoid of degree 7 with 4 generators>
gap> Size(S);
826897
gap> S := SingularModularPartitionMonoid(1, 1);
<commutative inverse bipartition semigroup ideal of degree 1 with
  1 generator>
gap> Size(SingularModularPartitionMonoid(2, 4));
355
gap> S := PlanarModularPartitionMonoid(4, 9);
<regular bipartition *-monoid of degree 9 with 14 generators>
gap> Size(S);
1795
gap> S := SingularPlanarModularPartitionMonoid(3, 5);
<regular bipartition *-semigroup ideal of degree 5 with 1 generator>
gap> Size(SingularPlanarModularPartitionMonoid(1, 2));
13
```

8.3.11 ApsisMonoid

▷ `ApsisMonoid(m, n)` (operation)
▷ `SingularApsisMonoid(m, n)` (operation)
▷ `CrossedApsisMonoid(m, n)` (operation)
▷ `SingularCrossedApsisMonoid(m, n)` (operation)

Returns: A bipartition monoid.

If `m` and `n` are positive integers, then this operation returns the `m`-apsis monoid of degree `n`. The `m`-*apsis monoid* is the monoid of bipartitions generated when the diapses in generators of the Jones monoid are replaced with `m`-apses. Note that an `m`-*apsis* is a block that contains precisely `m` consecutive integers.

`SingularApsisMonoid` returns the ideal of the apsis monoid consisting of the non-invertible elements (i.e. those not in the group of units), when $m \leq n$.

`CrossedApsisGeneratedMonoid` returns the semigroup generated by the symmetric group of degree `n` and the `m`-apsis monoid of degree `n`.

`SingularCrossedApsisMonoid` returns the ideal of the crossed apsis monoid consisting of the non-invertible elements (i.e. those not in the group of units), when `m <= n`.

Example
```
gap> S := ApsisMonoid(3, 7);
<regular bipartition *-monoid of degree 7 with 5 generators>
gap> Size(S);
320
gap> T := SingularApsisMonoid(3, 7);
<regular bipartition *-semigroup ideal of degree 7 with 1 generator>
gap> Difference(S, T) = [One(S)];
true
gap> Size(CrossedApsisMonoid(4, 9));
24291981
gap> SingularCrossedApsisMonoid(4, 6);
<regular bipartition *-semigroup ideal of degree 6 with 1 generator>
```

8.4 Standard PBR semigroups

In this section, we describe the operations in Semigroups that can be used to create standard examples of semigroups of partitioned binary relations (PBRs). See Chapter 4 for more information about PBRs.

8.4.1 FullPBRMonoid

▷ `FullPBRMonoid(n)` (operation)

Returns: A PBR monoid.

If `n` is a positive integer not greater than 2, then this operation returns the monoid consisting of all of the partitioned binary relations (PBRs) of degree `n`; called the *full PBR monoid*. There are `2 ^ (2 * n) ^ 2` PBRs of degree `n`. The full PBR monoid of degree `n` is currently too large to compute when $n \geq 3$.

The full PBR monoid is not regular in general.

Example
```
gap> S := FullPBRMonoid(1);
<pbr monoid of degree 1 with 4 generators>
```

```
gap> S := FullPBRMonoid(2);
<pbr monoid of degree 2 with 10 generators>
```

8.5 Semigroups of matrices over a finite field

In this section, we describe the operations in Semigroups that can be used to create semigroups of matrices over a finite field that belonging to several standard classes of example. See the section 'Matrices over finite fields' for more information about matrices over a finite field.

8.5.1 FullMatrixMonoid

▷ FullMatrixMonoid(*d*, *q*) (operation)
▷ GeneralLinearMonoid(*d*, *q*) (operation)
▷ GLM(*d*, *q*) (operation)

Returns: A matrix monoid.

These operations return the full matrix monoid of *d* by *d* matrices over the field with *q* elements. The *full matrix monoid*, also known as the *general linear monoid*, with these parameters, is the monoid consisting of all *d* by *d* matrices with entries from the field GF(*q*). This monoid has *q* ^ (*d* ^ 2) elements.

Example
```
gap> S := FullMatrixMonoid(3, 4);
<general linear monoid 3x3 over GF(2^2)>
gap> Size(S);
262144
gap> S = GeneralLinearMonoid(3, 4);
true
gap> GLM(2, 2);
<general linear monoid 2x2 over GF(2)>
```

8.5.2 SpecialLinearMonoid

▷ SpecialLinearMonoid(*d*, *q*) (operation)
▷ SLM(*d*, *q*) (operation)

Returns: A matrix monoid.

These operations return the special linear monoid of *d* by *d* matrices over the field with *q* elements. The *special linear monoid* is the monoid consisting of all *d* by *d* matrices with entries from the field GF(*q*) that have determinant 0 or 1. In other words, the special linear monoid is formed from the general linear monoid of the same parameters by replacing its group of units (the general linear group) by the special linear group.

Example
```
gap> S := SpecialLinearMonoid(3, 4);
<regular monoid of 3x3 matrices over GF(2^2) with 3 generators>
gap> S = SLM(3, 4);
true
gap> Size(S);
141184
```

8.5.3 IsFullMatrixMonoid

▷ `IsFullMatrixMonoid(S)` (property)
▷ `IsGeneralLinearMonoid(S)` (property)

`IsFullMatrixMonoid` and `IsGeneralLinearMonoid` return `true` if the semigroup `S` was created using either of the commands `FullMatrixMonoid` (8.5.1) or `GeneralLinearMonoid` (8.5.1) and `false` otherwise.

Example
```
gap> S := RandomSemigroup(IsTransformationSemigroup, 4, 4);;
gap> IsFullMatrixMonoid(S);
false
gap> S := GeneralLinearMonoid(3, 3);
<general linear monoid 3x3 over GF(3)>
gap> IsFullMatrixMonoid(S);
true
```

8.6 Semigroups of boolean matrices

In this section, we describe the operations in Semigroups that can be used to create semigroups of boolean matrices belonging to several standard classes of example. See the section 'Boolean matrices' for more information about boolean matrices.

8.6.1 FullBooleanMatMonoid

▷ `FullBooleanMatMonoid(d)` (operation)

Returns: The monoid of all boolean matrices of dimension `d`.

If `d` is a positive integer less than or equal to 5, then this operation returns the full boolean matrix monoid of dimension `d`. The *full boolean matrix monoid of dimension* `d` is the monoid consisting of all `d` by `d` boolean matrices, and has `2 ^ (n ^ 2)` matrices.

`FullBooleanMatMonoid` returns a monoid with a generating set that is minimal in size. These generating sets are pre-computed.

Example
```
gap> S := FullBooleanMatMonoid(3);
<monoid of 3x3 boolean matrices with 5 generators>
gap> Size(S);
512
```

8.6.2 RegularBooleanMatMonoid

▷ `RegularBooleanMatMonoid(d)` (operation)

Returns: A monoid of boolean matrices.

If `d` is a positive integer, then `RegularBooleanMatMonoid` returns the monoid generated by the regular `d` by `d` boolean matrices. Note that this monoid is *not* regular in general. `RegularBooleanMatMonoid(d)` is generated by the four boolean matrices `A`, `B`, `C`, `D`, whose `true` entries are:

- `A[i][i + 1]` and `A[n][1]`, for $i \in \{1,\ldots,n-1\}$;

- B[1][2], B[2][1], and B[i][i] for $i \in \{3,\dots,n\}$;
- C[1][2] and C[i][i], for $i \in \{2,\dots,n-1\}$; and
- D[1][2], D[i][i], for $i \in \{2,\dots,n\}$, and D[n][1].

This monoid has nearly 2 ^ (n ^ 2) elements.

8.6.3 ReflexiveBooleanMatMonoid

▷ ReflexiveBooleanMatMonoid(*d*) (operation)

Returns: A monoid of boolean matrices.

If *d* is a positive integer less than or equal to 5, then this operation returns the monoid consisting of all reflexive *d* by *d* boolean matrices. A boolean matrix mat is *reflexive* if each entry of its leading diagonal is true, i.e. if mat[i][i] is true for all $i \in \{1,\dots,d\}$.

The generating sets for the monoids returned by ReflexiveBooleanMatMonoid are pre-computed, and read from a file. Small generating sets are not known for $d \geq 6$.

Example
```
gap> S := ReflexiveBooleanMatMonoid(4);
<monoid of 4x4 boolean matrices with 38 generators>
gap> Size(S);
4096
```

8.6.4 HallMonoid

▷ HallMonoid(*d*) (operation)

Returns: A monoid of boolean matrices.

If *d* is a positive integer less than or equal to 5, then this operation returns the monoid consisting Hall matrices of degree *d*. A *Hall matrix* is a boolean matrix in which every column and every row contains at least one true entry. Equivalently, a Hall matrix is a boolean matrix than contains a permutation.

A Hall matrix of dimension *d* corresponds to a solution to Hall's Marriage Problem, when there are two collection of *d* people. Thus the number of solutions to Hall's Marriage Problem in this instance is the number of elements of HallMonoid(*d*).

The operation HallMonoid returns a monoid with a generating set that is minimal in size. These generating sets are pre-computed, and a minimal generating set is not known for larger dimensions.

Example
```
gap> S := HallMonoid(3);
<monoid of 3x3 boolean matrices with 4 generators>
gap> Size(S);
247
```

8.6.5 GossipMonoid

▷ GossipMonoid(*d*) (operation)

Returns: A monoid of boolean matrices.

If *d* is a positive integer, then this operation returns the *d* by *d* gossip monoid. The *gossip monoid* is defined to be the monoid generated by the collection of all *d* by *d* boolean matrices that define an equivalence relation; see IsEquivalenceBooleanMat (5.3.16).

For $d \geq 2$, `GossipMonoid(d)` returns a monoid with $\binom{d}{2}$ generators. The generating set is the collection of boolean matrices that define an equivalence relation that has one equivalence class of size 2, and no other non-trivial equivalence classes. Note that this generating set is strictly contained within the collection of all equivalence relation boolean matrices.

The number of elements of `GossipMonoid(d)` is known for some small values of d — see [BDF15] for more information about the gossip monoid, and its size for $d \leq 9$.

Example
```
gap> S := GossipMonoid(3);
<monoid of 3x3 boolean matrices with 3 generators>
gap> Size(S);
11
```

8.6.6 TriangularBooleanMatMonoid

▷ `TriangularBooleanMatMonoid(d)` (operation)
▷ `UnitriangularBooleanMatMonoid(d)` (operation)

Returns: A monoid of boolean matrices.

If d is a positive integer, then `TriangularBooleanMatMonoid` returns the monoid consisting of the upper-triangular d by d boolean matrices. A boolean matrix is *upper-triangular* if the entry in row `i`, column `j` is `false` whenever `i > j`.

`UnitriangularBooleanMatMonoid` returns the subsemigroup of the `TriangularBooleanMatMonoid` that consists of reflexive upper-triangular boolean matrices; see `ReflexiveBooleanMatMonoid` (8.6.3).

Example
```
gap> S := TriangularBooleanMatMonoid(3);
<monoid of 3x3 boolean matrices with 6 generators>
gap> Size(S);
64
gap> T := UnitriangularBooleanMatMonoid(4);
<monoid of 4x4 boolean matrices with 6 generators>
gap> Size(T);
64
```

8.7 Semigroups of matrices over a semiring

In this section, we describe the operations in Semigroups that can be used to create semigroups of matices over a semiring that belong to several standard classes of example. See Chapter 5 for more information about matrices over a semiring.

8.7.1 FullTropicalMaxPlusMonoid

▷ `FullTropicalMaxPlusMonoid(d, t)` (operation)

Returns: A monoid of tropical max plus matrices.

If `d = 2` and t is a positive integer, then `FullTropicalMaxPlusMonoid` returns the monoid consisting of all d by d matrices with entries from the tropical max-plus semiring with threshold t. A small generating set for larger values of d is not currently known.

This monoid contains `(t + 2) ^ (d ^ 2)` elements.

Example
```
gap> S := FullTropicalMaxPlusMonoid(2, 5);
<monoid of 2x2 tropical max-plus matrices with 24 generators>
gap> Size(S);
2401
gap> (5 + 2) ^ (2 ^ 2);
2401
```

8.7.2 FullTropicalMinPlusMonoid

▷ `FullTropicalMinPlusMonoid(d, t)` (operation)

Returns: A monoid of tropical min plus matrices.

If `d` is equal to 2 or 3, and `t` is a positive integer, then `FullTropicalMinPlusMonoid` returns the monoid consisting of all `d` by `d` matrices with entries from the tropical min-plus semiring with threshold `t`. A small generating set for larger values of `d` is not currently known.

This monoid contains `(t + 2) ^ (d ^ 2)` elements.

Example
```
gap> S := FullTropicalMinPlusMonoid(3, 2);
<monoid of 3x3 tropical min-plus matrices with 21 generators>
gap> Size(S);
262144
gap> (2 + 2) ^ (3 ^ 2);
262144
```

Chapter 9

Standard constructions

In this chapter we describe some standard semigroup constructions which are available in the Semigroups package.

9.1 Standard constructions

In this section, we describe the functions in Semigroups that can be used to create standard semigroup constructions in various representations. For all of the constructions, the default representation is as a semigroup of transformations. In general, these functions do not return a representation of minimal degree.

9.1.1 TrivialSemigroup

▷ `TrivialSemigroup([filt, ][deg])` (function)

Returns: A trivial semigroup.

A TRIVIAL semigroup is a semigroup with precisely one element. This function returns a trivial semigroup in the representation given by the filter *filter*, and (if possible) with the degree of the representation given by the non-negative integer *deg*.

The optional argument *filt* may be one of the following:

- `IsTransformationSemigroup` (the default, if *filt* is not specified),
- `IsPartialPermSemigroup`,
- `IsBipartitionSemigroup`,
- `IsBlockBijectionSemigroup`,
- `IsPBRSemigroup`,
- `IsBooleanMatSemigroup`.

If the optional argument *deg* is not specified, then the smallest possible degree will be used.

Example
```
gap> S := TrivialSemigroup();
<trivial transformation group of degree 0 with 1 generator>
gap> Size(S);
```

```
1
gap> S := TrivialSemigroup(3);
<trivial transformation group of degree 3 with 1 generator>
gap> S := TrivialSemigroup(IsBipartitionSemigroup, 2);
<trivial block bijection group of degree 2 with 1 generator>
gap> Elements(S);
[ <block bijection: [ 1, 2, -1, -2 ]> ]
```

9.1.2 MonogenicSemigroup

▷ `MonogenicSemigroup([filt, ]m, r)` (function)

Returns: A monogenic semigroup with index *m* and period *r*.

If *m* and *r* are positive integers, then this function returns a monogenic semigroup S with index *m* and period *r* in the representation given by the filter *filt*.

The optional argument *filt* may be one of the following:

- IsTransformationSemigroup (the default, if *filt* is not specified),
- IsPartialPermSemigroup,
- IsBipartitionSemigroup,
- IsBlockBijectionSemigroup,
- IsPBRSemigroup,
- IsBooleanMatSemigroup.

The semigroup S is generated by a single element, f. S consists of the elements $f, f^2, \ldots, f^m, \ldots, f^{m+r-1}$. The minimal ideal of S consists of the elements $f^m, \ldots, f^{m+r-1}$ and is isomorphic to the cyclic group of order r.

See IsMonogenicSemigroup (14.1.10) for more information about monogenic semigroups.

Example

```
gap> S := MonogenicSemigroup(5, 3);
<commutative non-regular transformation semigroup of size 7, degree 8
 with 1 generator>
gap> IsMonogenicSemigroup(S);
true
gap> I := MinimalIdeal(S);;
gap> IsGroupAsSemigroup(I);
true
gap> StructureDescription(I);
"C3"
gap> S := MonogenicSemigroup(IsBlockBijectionSemigroup, 9, 1);
<commutative non-regular block bijection semigroup of size 9,
 degree 10 with 1 generator>
```

9.1.3 RectangularBand

▷ `RectangularBand([filt, ]m, n)` (function)

Returns: An *m* by *n* rectangular band.

If *m* and *n* are positive integers, then this function returns a semigroup isomorphic to an *m* by *n* rectangular band, in the representation given by the filter *filt*.

The optional argument *filt* may be one of the following:

- `IsTransformationSemigroup` (the default, if *filt* is not specified),
- `IsBipartitionSemigroup`,
- `IsPBRSemigroup`,
- `IsBooleanMatSemigroup`,
- `IsReesMatrixSemigroup`.

See `IsRectangularBand` (14.1.14) for more information about rectangular bands.

Example
```
gap> T := RectangularBand(5, 6);
<regular transformation semigroup of size 30, degree 10 with 6
 generators>
gap> IsRectangularBand(T);
true
gap> S := RectangularBand(IsReesMatrixSemigroup, 4, 8);
<Rees matrix semigroup 4x8 over Group(())>
gap> IsRectangularBand(S);
true
gap> IsCompletelySimpleSemigroup(S) and IsHTrivial(S);
true
```

9.1.4 ZeroSemigroup

▷ `ZeroSemigroup([filt, ]n)` (function)

Returns: A zero semigroup of order *n*.

If *n* is a positive integer, then this function returns a zero semigroup of order *n* in the representation given by the filter *filt*.

The optional argument *filt* may be one of the following:

- `IsTransformationSemigroup` (the default, if *filt* is not specified),
- `IsPartialPermSemigroup`,
- `IsBipartitionSemigroup`,
- `IsBlockBijectionSemigroup`,
- `IsPBRSemigroup`,
- `IsBooleanMatSemigroup`,
- `IsReesZeroMatrixSemigroup` (provided that *n* `> 1`).

See `IsZeroSemigroup` (14.1.26) for more information about zero semigroups.

Example
```
gap> S := ZeroSemigroup(5);
<commutative non-regular transformation semigroup of size 5, degree 5
 with 4 generators>
gap> IsZeroSemigroup(S);
true
gap> S := ZeroSemigroup(IsPartialPermSemigroup, 15);
<commutative non-regular partial perm semigroup of size 15, rank 14
 with 14 generators>
gap> Size(S);
15
gap> z := MultiplicativeZero(S);
<empty partial perm>
gap> IsZeroSemigroup(S);
true
gap> ForAll(S, x -> ForAll(S, y -> x * y = z));
true
```

9.1.5 LeftZeroSemigroup

▷ LeftZeroSemigroup(*[filt,]n*) (function)
▷ RightZeroSemigroup(*[filt,]n*) (function)

Returns: A left zero (or right zero) semigroup of order *n*.

If *n* is a positive integer, then this function returns a left zero (or right zero, as appropriate) semigroup of order *n* in the representation given by the filter *filt*. If *filt* is not specified then the default representation is IsTransformationSemigroup.

The function LeftZeroSemigroup([*filt*,] *n*) simply calls RectangularBand([*filt*,] *n*, 1) and the function RightZeroSemigroup([*filt*,] *n*) simply calls RectangularBand([*filt*,] 1, *n*).

For more information about RectangularBand, including its permitted values of *filt*, see RectangularBand (9.1.3). See IsLeftZeroSemigroup (14.1.9) and IsRightZeroSemigroup (14.1.17) for more information about left zero and right zero semigroups.

Example
```
gap> S := LeftZeroSemigroup(20);
<transformation semigroup of degree 6 with 20 generators>
gap> IsLeftZeroSemigroup(S);
true
gap> ForAll(Tuples(S, 2), p -> p[1] * p[2] = p[1]);
true
gap> S := RightZeroSemigroup(IsBipartitionSemigroup, 5);
<regular bipartition semigroup of size 5, degree 3 with 5 generators>
gap> IsRightZeroSemigroup(S);
true
```

Chapter 10

Free objects

This chapter describes the functions in Semigroups for dealing with free inverse semigroups and free bands. This part of the manual and the functions described herein were written by Julius Jonušas.

10.1 Free inverse semigroups

An inverse semigroup F is said to be *free* on a non-empty set X if there is a map f from F to X such that for every inverse semigroup S and a map g from X to S there exists a unique homomorphism g' from F to S such that $fg' = g$. Moreover, by this universal property, every inverse semigroup can be expressed as a quotient of a free inverse semigroup.

The internal representation of an element of a free inverse semigroup uses a Munn tree. A *Munn tree* is a directed tree with distinguished start and terminal vertices and where the edges are labeled by generators so that two edges labeled by the same generator are only incident to the same vertex if one of the edges is coming in and the other is leaving the vertex. For more information regarding free inverse semigroups and the Munn representations see Section 5.10 of [How95].

See also (**Reference: Inverse semigroups and monoids**), (**Reference: Partial permutations**) and (**Reference: Free Groups, Monoids and Semigroups**).

An element of a free inverse semigroup in Semigroups is displayed, by default, as a shortest word corresponding to the element. However, there might be more than one word of the minimum length. For example, if x and y are generators of a free inverse semigroups, then

$$xyy^{-1}xx^{-1}x^{-1} = xxx^{-1}yy^{-1}x^{-1}.$$

See `MinimalWord` (10.3.2). Therefore we provide a another method for printing elements of a free inverse semigroup: a unique canonical form. Suppose an element of a free inverse semigroup is given as a Munn tree. Let L be the set of words corresponding to the shortest paths from the start vertex to the leaves of the tree. Also let w be a word corresponding to the shortest path from start to terminal vertices. The word vv^{-1} is an idempotent for every v in L. The canonical form is given by multiplying these idempotents, in shortlex order, and then postmultiplying by w. For example, consider the word $xyy^{-1}xx^{-1}x^{-1}$ again. The words corresponding to the paths to the leaves are in this case xx and xy. And w is an empty word since start and terminal vertices are the same. Therefore, the canonical form is

$$xxx^{-1}x^{-1}xyy^{-1}x^{-1}.$$

See `CanonicalForm` (10.3.1).

10.1.1 FreeInverseSemigroup (for a given rank)

▷ `FreeInverseSemigroup(rank[, name])` (function)
▷ `FreeInverseSemigroup(name1, name2, ...)` (function)
▷ `FreeInverseSemigroup(names)` (function)

Returns: A free inverse semigroup.

Returns a free inverse semigroup on *rank* generators, where *rank* is a positive integer. If *rank* is not specified, the number of *names* is used. If `S` is a free inverse semigroup, then the generators can be accessed by `S.1`, `S.2` and so on.

Example
```
gap> S := FreeInverseSemigroup(7);
<free inverse semigroup on the generators
[ x1, x2, x3, x4, x5, x6, x7 ]>
gap> S := FreeInverseSemigroup(7, "s");
<free inverse semigroup on the generators
[ s1, s2, s3, s4, s5, s6, s7 ]>
gap> S := FreeInverseSemigroup("a", "b", "c");
<free inverse semigroup on the generators [ a, b, c ]>
gap> S := FreeInverseSemigroup(["a", "b", "c"]);
<free inverse semigroup on the generators [ a, b, c ]>
gap> S.1;
a
gap> S.2;
b
```

10.1.2 IsFreeInverseSemigroupCategory

▷ `IsFreeInverseSemigroupCategory(obj)` (Category)

Every free inverse semigroup in GAP created by `FreeInverseSemigroup` (10.1.1) belongs to the category `IsFreeInverseSemigroup`. Basic operations for a free inverse semigroup are: `GeneratorsOfInverseSemigroup` (**Reference: GeneratorsOfInverseSemigroup**) and `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**). Elements of a free inverse semigroup belong to the category `IsFreeInverseSemigroupElement` (10.1.4).

10.1.3 IsFreeInverseSemigroup

▷ `IsFreeInverseSemigroup(S)` (property)

Returns: `true` or `false`

Attempts to determine whether the given semigroup *S* is a free inverse semigroup.

10.1.4 IsFreeInverseSemigroupElement

▷ `IsFreeInverseSemigroupElement` (Category)

Every element of a free inverse semigroup belongs to the category `IsFreeInverseSemigroupElement`.

10.1.5 IsFreeInverseSemigroupElementCollection

▷ `IsFreeInverseSemigroupElementCollection` (Category)

Every collection of elements of a free inverse semigroup belongs to the category `IsFreeInverseSemigroupElementCollection`. For example, every free inverse semigroup belongs to `IsFreeInverseSemigroupElementCollection`.

10.2 Displaying free inverse semigroup elements

There is a way to change how GAP displays free inverse semigroup elements using the user preference `FreeInverseSemigroupElementDisplay`. See `UserPreference` (**Reference: UserPreference**) for more information about user preferences.

There are two possible values for `FreeInverseSemigroupElementDisplay`:

minimal
: With this option selected, GAP will display a shortest word corresponding to the free inverse semigroup element. However, this shortest word is not unique. This is a default setting.

canonical
: With this option selected, GAP will display a free inverse semigroup element in the canonical form.

Example

```
gap> SetUserPreference("semigroups",
>                      "FreeInverseSemigroupElementDisplay",
>                      "minimal");
gap> S := FreeInverseSemigroup(2);
<free inverse semigroup on the generators [ x1, x2 ]>
gap> S.1 * S.2;
x1*x2
gap> SetUserPreference("semigroups",
>                      "FreeInverseSemigroupElementDisplay",
>                      "canonical");
gap> S.1 * S.2;
x1x2x2^-1x1^-1x1x2
```

10.3 Operators and operations for free inverse semigroup elements

`w ^ -1`
: returns the semigroup inverse of the free inverse semigroup element `w`.

`u * v`
: returns the product of two free inverse semigroup elements `u` and `v`.

`u = v`
: checks if two free inverse semigroup elements are equal, by comparing their canonical forms.

10.3.1 CanonicalForm (for a free inverse semigroup element)

▷ CanonicalForm(*w*) (attribute)

Returns: A string.

Every element of a free inverse semigroup has a unique canonical form. If *w* is such an element, then CanonicalForm returns the canonical form of *w* as a string.

Example
```
gap> S := FreeInverseSemigroup(3);
<free inverse semigroup on the generators [ x1, x2, x3 ]>
gap> x := S.1; y := S.2;
x1
x2
gap> CanonicalForm(x ^ 3 * y ^ 3);
"x1x1x1x2x2x2x2^-1x2^-1x2^-1x1^-1x1^-1x1^-1x1x1x1x2x2x2"
```

10.3.2 MinimalWord (for free inverse semigroup element)

▷ MinimalWord(*w*) (attribute)

Returns: A string.

For an element *w* of a free inverse semigroup S, MinimalWord returns a word of minimal length equal to *w* in S as a string.

Note that there maybe more than one word of minimal length which is equal to *w* in S.

Example
```
gap> S := FreeInverseSemigroup(3);
<free inverse semigroup on the generators [ x1, x2, x3 ]>
gap> x := S.1;
x1
gap> y := S.2;
x2
gap> MinimalWord(x ^ 3 * y ^ 3);
"x1*x1*x1*x2*x2*x2"
```

10.4 Free bands

A semigroup B is a *free band* on a non-empty set X if B is a band with a map f from B to X such that for every band S and every map g from X to B there exists a unique homomorphism g' from B to S such that $fg' = g$. The free band on a set X is unique up to isomorphism. Moreover, by the universal property, every band can be expressed as a quotient of a free band.

For an alternative description of a free band. Suppose that X is a non-empty set and X^+ a free semigroup on X. Also suppose that b is the smallest congurance on X^+ containing the set

$$\{(w^2, w) : w \in X^+\}.$$

Then the free band on X is isomorphic to the quotient of X^+ by b. See Section 4.5 of [How95] for more information on free bands.

10.4.1 FreeBand (for a given rank)

▷ FreeBand(*rank*[, *name*]) (function)
▷ FreeBand(*name1*, *name2*, ..., .) (function)
▷ FreeBand(*names*) (function)

Returns: A free band.

Returns a free band on *rank* generators, for a positive integer *rank*. If *rank* is not specified, the number of *names* is used. The resulting semigroup is always finite.

Example
```
gap> FreeBand(6);
<free band on the generators [ x1, x2, x3, x4, x5, x6 ]>
gap> FreeBand(6, "b");
<free band on the generators [ b1, b2, b3, b4, b5, b6 ]>
gap> FreeBand("a", "b", "c");
<free band on the generators [ a, b, c ]>
gap> FreeBand("a", "b", "c");
<free band on the generators [ a, b, c ]>
gap> S := FreeBand(["a", "b", "c"]);
<free band on the generators [ a, b, c ]>
gap> Size(S);
159
gap> gens := Generators(S);
[ a, b, c ]
gap> S.1 * S.2;
ab
```

10.4.2 IsFreeBandCategory

▷ IsFreeBandCategory (Category)

IsFreeBandCategory is the category of semigroups created using FreeBand (10.4.1).

Example
```
gap> IsFreeBandCategory(FreeBand(3));
true
gap> IsFreeBand(SymmetricGroup(6));
false
```

10.4.3 IsFreeBand (for a given semigroup)

▷ IsFreeBand(*S*) (property)

Returns: true or false.

IsFreeBand returns true if the given semigroup *S* is a free band.

Example
```
gap> IsFreeBand(FreeBand(3));
true
gap> IsFreeBand(SymmetricGroup(6));
false
gap> IsFreeBand(FullTransformationMonoid(7));
false
```

10.4.4 IsFreeBandElement

▷ `IsFreeBandElement` (Category)

`IsFreeBandElement` is a `Category` containing the elements of a free band.

Example

```
gap> IsFreeBandElement(Generators(FreeBand(4))[1]);
true
gap> IsFreeBandElement(Transformation([1, 3, 4, 1]));
false
gap> IsFreeBandElement((1, 2, 3, 4));
false
```

10.4.5 IsFreeBandElementCollection

▷ `IsFreeBandElementCollection` (Category)

Every collection of elements of a free band belongs to the category `IsFreeBandElementCollection`. For example, every free band belongs to `IsFreeBandElementCollection`.

10.4.6 IsFreeBandSubsemigroup

▷ `IsFreeBandSubsemigroup` (filter)

`IsFreeBandSubsemigroup` is a synonym for `IsSemigroup` and `IsFreeBandElementCollection`.

Example

```
gap> S := FreeBand(2);
<free band on the generators [ x1, x2 ]>
gap> x := S.1;
x1
gap> y := S.2;
x2
gap> new := Semigroup([x * y, x]);
<semigroup with 2 generators>
gap> IsFreeBand(new);
false
gap> IsFreeBandSubsemigroup(new);
true
```

10.4.7 ContentOfFreeBandElement

▷ `ContentOfFreeBandElement(x)` (attribute)
▷ `ContentOfFreeBandElementCollection(coll)` (attribute)

Returns: A list of integers

The content of a free band element `x` is the set of generators appearing in the word representing the element `x` of the free band.

The function `ContentOfFreeBandElement` returns the content of free band element `x` represented as a list of integers, where `1` represents the first generator, `2` the second generator, and so on.

The function `ContentOfFreeBandElementCollection` returns the the least list `C` for the collection of free band elements *coll* such that the content of every element in *coll* is contained in `C`.

Example
```
gap> S := FreeBand(2);
<free band on the generators [ x1, x2 ]>
gap> x := S.1;
x1
gap> y := S.2;
x2
gap> ContentOfFreeBandElement(x);
[ 1 ]
gap> ContentOfFreeBandElement(x * y);
[ 1, 2 ]
gap> ContentOfFreeBandElement(x * y * x);
[ 1, 2 ]
gap> ContentOfFreeBandElementCollection([x, y]);
[ 1, 2 ]
```

10.5 Operators and operations for free band elements

u * *v*
: returns the product of two free band elements *u* and *v*.

u = *v*
: checks if two free band elements are equal.

u < *v*
: compares the sizes of the internal representations of two free band elements.

10.5.1 GreensDClassOfElement (for a free band and element)

▷ `GreensDClassOfElement(S, x)` (operation)

Returns: A Green's $\mathscr{D}$-class

Let *S* be a free band. Two elements of *S* are $\mathscr{D}$-related if and only if they have the same content i.e. the set of generators appearing in any factorization of the elements. Therefore, a $\mathscr{D}$-class of a free band element *x* is the set of elements of *S* which have the same content as *x* .

Example
```
gap> S := FreeBand(3, "b");
<free band on the generators [ b1, b2, b3 ]>
gap> x := S.1 * S.2;
b1b2
gap> D := GreensDClassOfElement(S, x);
<Green's D-class: b1b2>
gap> IsGreensDClass(D);
true
```

Chapter 11

Graph inverse semigroups

In this chapter we describe a class of semigroups arising from directed graphs.

11.1 Creating graph inverse semigroups

11.1.1 GraphInverseSemigroup

▷ `GraphInverseSemigroup(E)` (operation)

Returns: A graph inverse semigroup.

If `E` is a digraph (i.e. it satisfies `IsDigraph` (**Digraphs: IsDigraph**)), then `GraphInverseSemigroup` returns the graph inverse semigroup $G(E)$ where, roughly speaking, elements correspond to paths in the graph `E`.

Let us describe `E` as a digraph `E` $= (E^0, E^1, r, s)$, where E^0 is the set of vertices, E^1 is the set of edges, and r and s are functions $E^1 \to E^0$ giving the *range* and *source* of an edge, respectively. The *graph inverse semigroup* $G(E)$ *of* E is the semigroup-with-zero generated by the sets E^0 and E^1, together with a set of variables $\{e^{-1} \mid e \in E^1\}$, satisfying the following relations for all $v, w \in E^0$ and $e, f \in E^1$:

(V) $vw = \delta_{v,w} \cdot v$,

(E1) $s(e) \cdot e = e \cdot r(e) = e$,

(E2) $r(e) \cdot e^{-1} = e^{-1} \cdot s(e) = e^{-1}$,

(CK1)
$e^{-1} \cdot f = \delta_{e,f} \cdot r(e)$.

(Here δ is the Kronecker delta.) We define $v^{-1} = v$ for each $v \in E^0$, and for any path $y = e_1 \dots e_n$ ($e_1 \dots e_n \in E^1$) we let $y^{-1} = e_n^{-1} \dots e_1^{-1}$. With this notation, every nonzero element of $G(E)$ can be written uniquely as xy^{-1} for some paths x, y in E, by the CK1 relation.

For a more complete description, see [MM16].

Example

```
gap> gr := Digraph([[2, 5, 8, 10], [2, 3, 4, 5, 6, 8, 9, 10], [1],
>                   [3, 5, 7, 8, 10], [2, 5, 7], [3, 6, 7, 9, 10],
>                   [1, 4], [1, 5, 9], [1, 2, 7, 8], [3, 5]]);
<digraph with 10 vertices, 37 edges>
gap> S := GraphInverseSemigroup(gr);
```

```
<infinite graph inverse semigroup with 10 vertices, 37 edges>
gap> GeneratorsOfInverseSemigroup(S);
[ e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_10, e_11, e_12,
  e_13, e_14, e_15, e_16, e_17, e_18, e_19, e_20, e_21, e_22, e_23,
  e_24, e_25, e_26, e_27, e_28, e_29, e_30, e_31, e_32, e_33, e_34,
  e_35, e_36, e_37, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_10
 ]
gap> AssignGeneratorVariables(S);
gap> e_1 * e_1 ^ -1;
e_1e_1^-1
gap> e_1 ^ -1 * e_1 ^ -1;
0
gap> e_1 ^ -1 * e_1;
v_2
```

11.1.2 Range (for a graph inverse semigroup element)

▷ `Range(x)` (attribute)
▷ `Source(x)` (attribute)

Returns: A graph inverse semigroup element.

If *x* is an element of a graph inverse semigroup (i.e. it satisfies `IsGraphInverseSemigroupElement` (11.1.4)), then `Range` and `Source` give, respectively, the start and end vertices of *x* when viewed as a path in the digraph over which the semigroup is defined.

For a fuller description, see `GraphInverseSemigroup` (11.1.1).

Example
```
gap> gr := Digraph([[], [1], [3]]);;
gap> S := GraphInverseSemigroup(gr);;
gap> e := S.1;
e_1
gap> Source(e);
v_2
gap> Range(e);
v_1
```

11.1.3 IsVertex (for a graph inverse semigroup element)

▷ `IsVertex(x)` (operation)

Returns: `true` or `false`.

If *x* is an element of a graph inverse semigroup (i.e. it satisfies `IsGraphInverseSemigroupElement` (11.1.4)), then this attribute returns `true` if *x* corresponds to a vertex in the digraph over which the semigroup is defined, and `false` otherwise.

For a fuller description, see `GraphInverseSemigroup` (11.1.1).

Example
```
gap> gr := Digraph([[], [1], [3]]);;
gap> S := GraphInverseSemigroup(gr);;
gap> e := S.1;
e_1
gap> IsVertex(e);
false
```

```
gap> v := S.3;
v_1
gap> IsVertex(v);
true
gap> z := v * e;
0
gap> IsVertex(z);
false
```

11.1.4 IsGraphInverseSemigroup

▷ `IsGraphInverseSemigroup(x)` (filter)
▷ `IsGraphInverseSemigroupElement(x)` (filter)

Returns: `true` or `false`.

The category `IsGraphInverseSemigroup` contains any semigroup defined over a digraph using the `GraphInverseSemigroup` (11.1.1) operation. The category `IsGraphInverseSemigroupElement` contains any element contained in such a semigroup.

Example
```
gap> gr := Digraph([[], [1], [3]]);;
gap> S := GraphInverseSemigroup(gr);
<infinite graph inverse semigroup with 3 vertices, 2 edges>
gap> IsGraphInverseSemigroup(S);
true
gap> x := GeneratorsOfSemigroup(S)[1];
e_1
gap> IsGraphInverseSemigroupElement(x);
true
```

11.1.5 GraphOfGraphInverseSemigroup

▷ `GraphOfGraphInverseSemigroup(S)` (attribute)

Returns: A digraph.

If S is a graph inverse semigroup (i.e. it satisfies `IsGraphInverseSemigroup` (11.1.4)), then this attribute returns the original digraph over which S was defined (most likely the argument given to `GraphInverseSemigroup` (11.1.1) to create S).

Example
```
gap> gr := Digraph([[], [1], [3]]);
<digraph with 3 vertices, 2 edges>
gap> S := GraphInverseSemigroup(gr);;
gap> GraphOfGraphInverseSemigroup(S);
<digraph with 3 vertices, 2 edges>
```

11.1.6 IsGraphInverseSemigroupElementCollection

▷ `IsGraphInverseSemigroupElementCollection` (Category)

Every collection of elements of a graph inverse semigroup belongs to the category `IsGraphInverseSemigroupElementCollection`. For example, every graph inverse semigroup belongs to `IsGraphInverseSemigroupElementCollection`.

11.1.7 IsGraphInverseSubsemigroup

▷ IsGraphInverseSubsemigroup (filter)

IsGraphInverseSubsemigroup is a synonym for IsSemigroup and IsInverseSemigroup and IsGraphInverseSemigroupElementCollection.

See IsGraphInverseSemigroupElementCollection (11.1.6) and IsInverseSemigroup (**Reference: IsInverseSemigroup**).

Example

```
gap> gr := Digraph([[], [1], [2]]);
<digraph with 3 vertices, 2 edges>
gap> S := GraphInverseSemigroup(gr);
<finite graph inverse semigroup with 3 vertices, 2 edges>
gap> Elements(S);
[ e_2^-1, e_1^-1, e_1^-1e_2^-1, 0, e_1, e_1e_1^-1, e_1e_1^-1e_2^-1,
  e_2, e_2e_2^-1, e_2e_1, e_2e_1e_1^-1, e_2e_1e_1^-1e_2^-1, v_1, v_2,
  v_3 ]
gap> T := InverseSemigroup(Elements(S){[3, 5]});;
gap> IsGraphInverseSubsemigroup(T);
true
```

Chapter 12

Green's relations

In this chapter we describe the functions in Semigroups for computing Green's classes and related properties of semigroups.

12.1 Creating Green's classes and representatives

In this section, we describe the methods in the Semigroups package for creating Green's classes.

12.1.1 XClassOfYClass

▷ `DClassOfHClass(class)` (method)
▷ `DClassOfLClass(class)` (method)
▷ `DClassOfRClass(class)` (method)
▷ `LClassOfHClass(class)` (method)
▷ `RClassOfHClass(class)` (method)

Returns: A Green's class.

`XClassOfYClass` returns the `X`-class containing the `Y`-class *class* where `X` and `Y` should be replaced by an appropriate choice of `D`, `H`, `L`, and `R`.

Note that if it is not known to GAP whether or not the representative of *class* is an element of the semigroup containing *class*, then no attempt is made to check this.

The same result can be produced using:

Example
```
First(GreensXClasses(S), x -> Representative(x) in class);
```

but this might be substantially slower. Note that `XClassOfYClass` is also likely to be faster than

Example
```
GreensXClassOfElement(S, Representative(class));
```

`DClass` can also be used as a synonym for `DClassOfHClass`, `DClassOfLClass`, and `DClassOfRClass`; `LClass` as a synonym for `LClassOfHClass`; and `RClass` as a synonym for `RClassOfHClass`. See also `GreensDClassOfElement` (**Reference: GreensDClassOfElement**) and `GreensDClassOfElementNC` (12.1.3).

Example
```
gap> S := Semigroup(Transformation([1, 3, 2]),
>                   Transformation([2, 1, 3]),
```

```
>                     Transformation([3, 2, 1]),
>                     Transformation([1, 3, 1]));;
gap> R := GreensRClassOfElement(S, Transformation([3, 2, 1]));
<Green's R-class: Transformation( [ 3, 2, 1 ] )>
gap> DClassOfRClass(R);
<Green's D-class: Transformation( [ 3, 2, 1 ] )>
gap> IsGreensDClass(DClassOfRClass(R));
true
gap> S := InverseSemigroup(
> PartialPerm([2, 6, 7, 0, 0, 9, 0, 1, 0, 5]),
> PartialPerm([3, 8, 1, 9, 0, 4, 10, 5, 0, 6]));
<inverse partial perm semigroup of rank 10 with 2 generators>
gap> x := S.1;
[3,7][8,1,2,6,9][10,5]
gap> H := HClass(S, x);
<Green's H-class: [3,7][8,1,2,6,9][10,5]>
gap> R := RClassOfHClass(H);
<Green's R-class: [3,7][8,1,2,6,9][10,5]>
gap> L := LClass(H);;
gap> L = LClass(S, PartialPerm([1, 2, 0, 0, 5, 6, 7, 0, 9]));
true
gap> DClass(R) = DClass(L);
true
gap> DClass(H) = DClass(L);
true
```

12.1.2 GreensXClassOfElement

▷ `GreensDClassOfElement(X, f)` (operation)
▷ `DClass(X, f)` (operation)
▷ `GreensHClassOfElement(X, f)` (operation)
▷ `GreensHClassOfElement(R, i, j)` (operation)
▷ `HClass(X, f)` (operation)
▷ `HClass(R, i, j)` (operation)
▷ `GreensLClassOfElement(X, f)` (operation)
▷ `LClass(X, f)` (operation)
▷ `GreensRClassOfElement(X, f)` (operation)
▷ `RClass(X, f)` (operation)

Returns: A Green's class.

These functions produce essentially the same output as the GAP library functions with the same names; see `GreensDClassOfElement` (**Reference: GreensDClassOfElement**). The main difference is that these functions can be applied to a wider class of objects:

`GreensDClassOfElement` **and** `DClass`
: `X` must be a semigroup.

`GreensHClassOfElement` **and** `HClass`
: `X` can be a semigroup, $\mathscr{R}$-class, $\mathscr{L}$-class, or $\mathscr{D}$-class. If `R` is a `IxJ` Rees matrix semigroup or a Rees 0-matrix semigroup, and `i` and `j` are integers of the corresponding index sets, then `GreensHClassOfElement` returns the $\mathscr{H}$-class in row `i` and column `j`.

`GreensLClassOfElement` **and** `LClass`
  X can be a semigroup or $\mathscr{D}$-class.

`GreensRClassOfElement` **and** `RClass`
  X can be a semigroup or $\mathscr{D}$-class.

Note that `GreensXClassOfElement` and `XClass` are synonyms and have identical output. The shorter command is provided for the sake of convenience.

12.1.3 GreensXClassOfElementNC

▷ `GreensDClassOfElementNC(X, f)` (operation)
▷ `DClassNC(X, f)` (operation)
▷ `GreensHClassOfElementNC(X, f)` (operation)
▷ `HClassNC(X, f)` (operation)
▷ `GreensLClassOfElementNC(X, f)` (operation)
▷ `LClassNC(X, f)` (operation)
▷ `GreensRClassOfElementNC(X, f)` (operation)
▷ `RClassNC(X, f)` (operation)

Returns: A Green's class.

These functions are essentially the same as `GreensDClassOfElement` (12.1.2) except that no effort is made to verify if f is an element of X. More precisely, `GreensXClassOfElementNC` and `XClassNC` first check if f has already been shown to be an element of X. If it is not known to GAP if f is an element of X, then no further attempt to verify this is made.

Note that `GreensXClassOfElementNC` and `XClassNC` are synonyms and have identical output. The shorter command is provided for the sake of convenience.

It can be quicker to compute the class of an element using `GreensRClassOfElementNC`, say, than using `GreensRClassOfElement` if it is known *a priori* that f is an element of X. On the other hand, if f is not an element of X, then the results of this computation are unpredictable.

For example, if

Example
```
x := Transformation([15, 18, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
```

in the semigroup X of order-preserving mappings on 20 points, then

Example
```
GreensRClassOfElementNC(X, x);
```

returns an answer relatively quickly, whereas

Example
```
GreensRClassOfElement(X, x)
```

can take a signficant amount of time to return a value.

See also `GreensRClassOfElement` (**Reference: GreensRClassOfElement**) and `RClassOfHClass` (12.1.1).

Example
```
gap> S := RandomSemigroup(IsTransformationSemigroup, 2, 1000);;
gap> x := [1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 1, 2, 2, 1];;
gap> x := EvaluateWord(Generators(S), x);;
gap> R := GreensRClassOfElementNC(S, x);;
```

```
gap> Size(R);
1
gap> L := GreensLClassOfElementNC(S, x);;
gap> Size(L);
1
gap> x := PartialPerm([2, 3, 4, 5, 0, 0, 6, 8, 10, 11]);;
gap> L := LClass(POI(13), x);
<Green's L-class: [1,2,3,4,5,6,8][7,10,11]>
gap> Size(L);
1287
```

12.1.4 GreensXClasses

▷ `GreensDClasses(obj)` (method)
▷ `DClasses(obj)` (method)
▷ `GreensHClasses(obj)` (method)
▷ `HClasses(obj)` (method)
▷ `GreensJClasses(obj)` (method)
▷ `JClasses(obj)` (method)
▷ `GreensLClasses(obj)` (method)
▷ `LClasses(obj)` (method)
▷ `GreensRClasses(obj)` (method)
▷ `RClasses(obj)` (method)

Returns: A list of Green's classes.

These functions produce essentially the same output as the GAP library functions with the same names; see `GreensDClasses` (**Reference: GreensDClasses**). The main difference is that these functions can be applied to a wider class of objects:

`GreensDClasses` **and** `DClasses`
: `X` should be a semigroup.

`GreensHClasses` **and** `HClasses`
: `X` can be a semigroup, $\mathscr{R}$-class, $\mathscr{L}$-class, or $\mathscr{D}$-class.

`GreensLClasses` **and** `LClasses`
: `X` can be a semigroup or $\mathscr{D}$-class.

`GreensRClasses` **and** `RClasses`
: `X` can be a semigroup or $\mathscr{D}$-class.

Note that `GreensXClasses` and `XClasses` are synonyms and have identical output. The shorter command is provided for the sake of convenience.

See also `DClassReps` (12.1.5), `IteratorOfDClassReps` (12.2.1), `IteratorOfDClasses` (12.2.2), and `NrDClasses` (12.1.9).

Example
```
gap> S := Semigroup(Transformation([3, 4, 4, 4]),
>                   Transformation([4, 3, 1, 2]));;
gap> GreensDClasses(S);
[ <Green's D-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's D-class: Transformation( [ 4, 3, 1, 2 ] )>,
```

```
  <Green's D-class: Transformation( [ 4, 4, 4, 4 ] )> ]
gap> GreensRClasses(S);
[ <Green's R-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 3, 1, 2 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 3, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 3, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 4, 3 ] )> ]
gap> D := GreensDClasses(S)[1];
<Green's D-class: Transformation( [ 3, 4, 4, 4 ] )>
gap> GreensLClasses(D);
[ <Green's L-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's L-class: Transformation( [ 1, 2, 2, 2 ] )> ]
gap> GreensRClasses(D);
[ <Green's R-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 3, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 3, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 4, 3 ] )> ]
gap> R := GreensRClasses(D)[1];
<Green's R-class: Transformation( [ 3, 4, 4, 4 ] )>
gap> GreensHClasses(R);
[ <Green's H-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's H-class: Transformation( [ 1, 2, 2, 2 ] )> ]
gap> S := InverseSemigroup([
> PartialPerm([2, 4, 1]), PartialPerm([3, 0, 4, 1])]);;
gap> GreensDClasses(S);
[ <Green's D-class: <identity partial perm on [ 1, 2, 4 ]>>,
  <Green's D-class: <identity partial perm on [ 1, 3, 4 ]>>,
  <Green's D-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's D-class: <identity partial perm on [ 4 ]>>,
  <Green's D-class: <empty partial perm>> ]
gap> GreensLClasses(S);
[ <Green's L-class: <identity partial perm on [ 1, 2, 4 ]>>,
  <Green's L-class: [4,2,1,3]>,
  <Green's L-class: <identity partial perm on [ 1, 3, 4 ]>>,
  <Green's L-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's L-class: [3,1,2]>, <Green's L-class: [1,4][3,2]>,
  <Green's L-class: [1,3,4]>, <Green's L-class: [3,1,4]>,
  <Green's L-class: [1,2](3)>,
  <Green's L-class: <identity partial perm on [ 4 ]>>,
  <Green's L-class: [4,1]>, <Green's L-class: [4,3]>,
  <Green's L-class: [4,2]>, <Green's L-class: <empty partial perm>> ]
gap> D := GreensDClasses(S)[3];
<Green's D-class: <identity partial perm on [ 1, 3 ]>>
gap> GreensLClasses(D);
[ <Green's L-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's L-class: [3,1,2]>, <Green's L-class: [1,4][3,2]>,
  <Green's L-class: [1,3,4]>, <Green's L-class: [3,1,4]>,
  <Green's L-class: [1,2](3)> ]
gap> GreensRClasses(D);
[ <Green's R-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's R-class: [2,1,3]>, <Green's R-class: [2,3][4,1]>,
  <Green's R-class: [4,3,1]>, <Green's R-class: [4,1,3]>,
```

```
<Green's R-class: [2,1](3)> ]
```

12.1.5 XClassReps

▷ `DClassReps(obj)` (attribute)
▷ `HClassReps(obj)` (attribute)
▷ `LClassReps(obj)` (attribute)
▷ `RClassReps(obj)` (attribute)

Returns: A list of representatives.

`XClassReps` returns a list of the representatives of the Green's classes of *obj*, which can be a semigroup, $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{R}$-class where appropriate.

The same output can be obtained by calling, for example:

Example
```
List(GreensXClasses(obj), Representative);
```

Note that if the Green's classes themselves are not required, then `XClassReps` will return an answer more quickly than the above, since the Green's class objects are not created.

See also `GreensDClasses` (12.1.4), `IteratorOfDClassReps` (12.2.1), `IteratorOfDClasses` (12.2.2), and `NrDClasses` (12.1.9).

Example
```
gap> S := Semigroup(Transformation([3, 4, 4, 4]),
>                   Transformation([4, 3, 1, 2]));;
gap> DClassReps(S);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 4, 3, 1, 2 ] ),
  Transformation( [ 4, 4, 4, 4 ] ) ]
gap> LClassReps(S);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 1, 2, 2, 2 ] ),
  Transformation( [ 4, 3, 1, 2 ] ), Transformation( [ 4, 4, 4, 4 ] ),
  Transformation( [ 2, 2, 2, 2 ] ), Transformation( [ 3, 3, 3, 3 ] ),
  Transformation( [ 1, 1, 1, 1 ] ) ]
gap> D := GreensDClasses(S)[1];
<Green's D-class: Transformation( [ 3, 4, 4, 4 ] )>
gap> LClassReps(D);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 1, 2, 2, 2 ] ) ]
gap> RClassReps(D);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 4, 4, 3, 4 ] ),
  Transformation( [ 4, 3, 4, 4 ] ), Transformation( [ 4, 4, 4, 3 ] ) ]
gap> R := GreensRClasses(D)[1];;
gap> HClassReps(R);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 1, 2, 2, 2 ] ) ]
gap> S := SymmetricInverseSemigroup(6);;
gap> e := InverseSemigroup(Idempotents(S));;
gap> M := MunnSemigroup(e);;
gap> L := LClassNC(M, PartialPerm([51, 63], [51, 47]));;
gap> HClassReps(L);
[ <identity partial perm on [ 47, 51 ]>, [27,47](51), [50,47](51),
  [64,47](51), [63,47](51), [59,47](51) ]
```

12.1.6 MinimalDClass

▷ `MinimalDClass(S)` (attribute)

Returns: The minimal $\mathscr{D}$-class of a semigroup.

The minimal ideal of a semigroup is the least ideal with respect to containment. `MinimalDClass` returns the $\mathscr{D}$-class corresponding to the minimal ideal of the semigroup `S`. Equivalently, `MinimalDClass` returns the minimal $\mathscr{D}$-class with respect to the partial order of $\mathscr{D}$-classes.

It is significantly easier to find the minimal $\mathscr{D}$-class of a semigroup, than to find its $\mathscr{D}$-classes.

See also `PartialOrderOfDClasses` (12.1.10), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), `MinimalIdeal` (13.7.1) and `RepresentativeOfMinimalIdeal` (13.7.2).

Example
```
gap> D := MinimalDClass(JonesMonoid(8));
<Green's D-class: <bipartition: [ 1, 2 ], [ 3, 4 ], [ 5, 6 ],
  [ 7, 8 ], [ -1, -2 ], [ -3, -4 ], [ -5, -6 ], [ -7, -8 ]>>
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 5, 7, 8, 9], [2, 6, 9, 1, 5, 3, 8]),
> PartialPerm([1, 3, 4, 5, 7, 8, 9], [9, 4, 10, 5, 6, 7, 1]));;
gap> MinimalDClass(S);
<Green's D-class: <empty partial perm>>
```

12.1.7 MaximalDClasses

▷ `MaximalDClasses(S)` (attribute)

Returns: The maximal $\mathscr{D}$-classes of a semigroup.

`MaximalDClasses` returns the maximal $\mathscr{D}$-classes with respect to the partial order of $\mathscr{D}$-classes.

See also `PartialOrderOfDClasses` (12.1.10), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), and `MinimalDClass` (12.1.6).

Example
```
gap> MaximalDClasses(BrauerMonoid(8));
[ <Green's D-class: <block bijection: [ 1, -1 ], [ 2, -2 ],
      [ 3, -3 ], [ 4, -4 ], [ 5, -5 ], [ 6, -6 ], [ 7, -7 ],
      [ 8, -8 ]>> ]
gap> MaximalDClasses(FullTransformationMonoid(5));
[ <Green's D-class: IdentityTransformation> ]
gap> S := Semigroup(
> PartialPerm([1, 2, 3, 4, 5, 6, 7], [3, 8, 1, 4, 5, 6, 7]),
> PartialPerm([1, 2, 3, 6, 8], [2, 6, 7, 1, 5]),
> PartialPerm([1, 2, 3, 4, 6, 8], [4, 3, 2, 7, 6, 5]),
> PartialPerm([1, 2, 4, 5, 6, 7, 8], [7, 1, 4, 2, 5, 6, 3]));;
gap> MaximalDClasses(S);
[ <Green's D-class: [2,8](1,3)(4)(5)(6)(7)>,
  <Green's D-class: [8,3](1,7,6,5,2)(4)> ]
```

12.1.8 NrRegularDClasses

▷ `NrRegularDClasses(S)` (attribute)

▷ `RegularDClasses(S)` (attribute)

Returns: A positive integer, or a list.

`NrRegularDClasses` returns the number of regular $\mathscr{D}$-classes of the semigroup `S`.

RegularDClasses returns a list of the regular $\mathscr{D}$-classes of the semigroup *S*.

See also IsRegularGreensClass (12.3.2) and IsRegularDClass (**Reference: IsRegularDClass**).

Example
```
gap> S := Semigroup(Transformation([1, 3, 4, 1, 3, 5]),
>                   Transformation([5, 1, 6, 1, 6, 3]));;
gap> NrRegularDClasses(S);
3
gap> NrDClasses(S);
7
gap> AsSet(RegularDClasses(S));
[ <Green's D-class: Transformation( [ 1, 3, 4, 1, 3, 3 ] )>,
  <Green's D-class: Transformation( [ 1, 1, 1, 1, 1 ] )>,
  <Green's D-class: Transformation( [ 1, 1, 1, 1, 1, 1 ] )> ]
```

12.1.9 NrXClasses

▷ NrDClasses(*obj*) (attribute)
▷ NrHClasses(*obj*) (attribute)
▷ NrLClasses(*obj*) (attribute)
▷ NrRClasses(*obj*) (attribute)

Returns: A positive integer.

NrXClasses returns the number of Green's classes in *obj* where *obj* can be a semigroup, $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{R}$-class where appropriate. If the actual Green's classes are not required, then it is more efficient to use

Example
```
NrHClasses(obj)
```

than

Example
```
Length(HClasses(obj))
```

since the Green's classes themselves are not created when NrXClasses is called.

See also GreensRClasses (12.1.4), GreensRClasses (**Reference: GreensRClasses**), IteratorOfRClasses (12.2.2), and IteratorOfRClassReps (12.2.1).

Example
```
gap> S := Semigroup(
>  Transformation([1, 2, 5, 4, 3, 8, 7, 6]),
>  Transformation([1, 6, 3, 4, 7, 2, 5, 8]),
>  Transformation([2, 1, 6, 7, 8, 3, 4, 5]),
>  Transformation([3, 2, 3, 6, 1, 6, 1, 2]),
>  Transformation([5, 2, 3, 6, 3, 4, 7, 4]));;
gap> x := Transformation([2, 5, 4, 7, 4, 3, 6, 3]);;
gap> R := RClass(S, x);
<Green's R-class: Transformation( [ 2, 5, 4, 7, 4, 3, 6, 3 ] )>
gap> NrHClasses(R);
12
gap> D := DClass(R);
<Green's D-class: Transformation( [ 2, 5, 4, 7, 4, 3, 6, 3 ] )>
gap> NrHClasses(D);
72
```

```
gap> L := LClass(S, x);
<Green's L-class: Transformation( [ 2, 5, 4, 7, 4, 3, 6, 3 ] )>
gap> NrHClasses(L);
6
gap> NrHClasses(S);
1555
gap> S := Semigroup(Transformation([4, 6, 5, 2, 1, 3]),
>                   Transformation([6, 3, 2, 5, 4, 1]),
>                   Transformation([1, 2, 4, 3, 5, 6]),
>                   Transformation([3, 5, 6, 1, 2, 3]),
>                   Transformation([5, 3, 6, 6, 6, 2]),
>                   Transformation([2, 3, 2, 6, 4, 6]),
>                   Transformation([2, 1, 2, 2, 2, 4]),
>                   Transformation([4, 4, 1, 2, 1, 2]));;
gap> NrRClasses(S);
150
gap> Size(S);
6342
gap> x := Transformation([1, 3, 3, 1, 3, 5]);;
gap> D := DClass(S, x);
<Green's D-class: Transformation( [ 2, 4, 2, 2, 2, 1 ] )>
gap> NrRClasses(D);
87
gap> S := SymmetricInverseSemigroup(10);;
gap> NrDClasses(S); NrRClasses(S); NrHClasses(S); NrLClasses(S);
11
1024
184756
1024
gap> S := POPI(10);;
gap> NrDClasses(S);
11
gap> NrRClasses(S);
1024
```

12.1.10 PartialOrderOfDClasses

▷ `PartialOrderOfDClasses(S)` (attribute)

Returns: The partial order of the $\mathscr{D}$-classes of S.

Returns a list `list` where `list[i]` contains every `j` such that `GreensDClasses(S)[j]` is immediately less than `GreensDClasses(S)[i]` in the partial order of $\mathscr{D}$- classes of S. There might be other indices in `list`, and it may or may not include `i`. The reflexive transitive closure of the relation defined by `list` is the partial order of $\mathscr{D}$-classes of S.

The partial order on the $\mathscr{D}$-classes is defined by $x \leq y$ if and only if S^1xS^1 is a subset of S^1yS^1.

See also `GreensDClasses` (12.1.4), `GreensDClasses` (**Reference: GreensDClasses**), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), and \< (12.3.1).

Example
```
gap> S := Semigroup(Transformation([2, 4, 1, 2]),
>                   Transformation([3, 3, 4, 1]));;
gap> PartialOrderOfDClasses(S);
[ [ 3 ], [ 2, 3 ], [ 3, 4 ], [ 4 ] ]
```

```
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[1],
>                            GreensDClasses(S)[2]);
false
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[2],
>                            GreensDClasses(S)[1]);
false
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[3],
>                            GreensDClasses(S)[1]);
true
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3], [1, 3, 4]),
> PartialPerm([1, 3, 5], [5, 1, 3]));;
gap> Size(S);
58
gap> PartialOrderOfDClasses(S);
[ [ 1, 3 ], [ 2, 3 ], [ 3, 4 ], [ 4, 5 ], [ 5 ] ]
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[1],
>                            GreensDClasses(S)[2]);
false
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[5],
>                            GreensDClasses(S)[2]);
true
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[3],
>                            GreensDClasses(S)[4]);
false
gap> IsGreensLessThanOrEqual(GreensDClasses(S)[4],
>                            GreensDClasses(S)[3]);
true
```

12.1.11 LengthOfLongestDClassChain

▷ `LengthOfLongestDClassChain(`*S*`)` (attribute)

Returns: A non-negative integer.

If *S* is a semigroup, then `LengthOfLongestDClassChain` returns the length of the longest chain in the partial order defined by `PartialOrderOfDClasses(`*S*`)`. See `PartialOrderOfDClasses` (12.1.10).

The partial order on the $\mathscr{D}$-classes is defined by $x \leq y$ if and only if S^1xS^1 is a subset of S^1yS^1. A *chain* of $\mathscr{D}$-classes is a collection of `n` $\mathscr{D}$-classes $D_1, D_2, \ldots D_n$ such that $D_1 < D_2 < \cdots < D_n$. The *length* of such a chain is `n - 1`.

Example

```
gap> S := TrivialSemigroup();;
gap> LengthOfLongestDClassChain(S);
0
gap> T := ZeroSemigroup(5);;
gap> LengthOfLongestDClassChain(T);
1
gap> U := MonogenicSemigroup(14, 7);;
gap> LengthOfLongestDClassChain(U);
13
gap> V := FullTransformationMonoid(6);
<full transformation monoid of degree 6>
```

```
gap> LengthOfLongestDClassChain(V);
5
```

12.1.12 IsGreensDGreaterThanFunc

▷ IsGreensDGreaterThanFunc(*S*) (attribute)

Returns: A function.

IsGreensDGreaterThanFunc(*S*) returns a function func such that for any two elements x and y of *S*, func(x, y) return true if the $\mathscr{D}$-class of x in *S* is greater than or equal to the $\mathscr{D}$-class of y in *S* under the usual ordering of Green's $\mathscr{D}$-classes of a semigroup.

Example

```
gap> S := Semigroup(Transformation([1, 3, 4, 1, 3]),
>                   Transformation([2, 4, 1, 5, 5]),
>                   Transformation([2, 5, 3, 5, 3]),
>                   Transformation([5, 5, 1, 1, 3]));;
gap> reps := ShallowCopy(AsSet(DClassReps(S)));
[ Transformation( [ 1, 1, 1, 1, 1 ] ),
  Transformation( [ 1, 3, 1, 3, 3 ] ),
  Transformation( [ 1, 3, 4, 1, 3 ] ),
  Transformation( [ 2, 4, 1, 5, 5 ] ) ]
gap> Sort(reps, IsGreensDGreaterThanFunc(S));
gap> reps;
[ Transformation( [ 2, 4, 1, 5, 5 ] ),
  Transformation( [ 1, 3, 4, 1, 3 ] ),
  Transformation( [ 1, 3, 1, 3, 3 ] ),
  Transformation( [ 1, 1, 1, 1, 1 ] ) ]
gap> IsGreensLessThanOrEqual(DClass(S, reps[2]),
>                            DClass(S, reps[1]));
true
gap> S := DualSymmetricInverseMonoid(4);;
gap> IsGreensDGreaterThanFunc(S)(S.1, S.3);
true
gap> IsGreensDGreaterThanFunc(S)(S.3, S.1);
false
gap> IsGreensLessThanOrEqual(DClass(S, S.3),
>                            DClass(S, S.1));
true
gap> IsGreensLessThanOrEqual(DClass(S, S.1),
>                            DClass(S, S.3));
false
```

12.2 Iterators and enumerators of classes and representatives

In this section, we describe the methods in the Semigroups package for incrementally determining Green's classes or their representatives.

12.2.1 IteratorOfXClassReps

▷ IteratorOfDClassReps(*S*) (operation)

▷ IteratorOfHClassReps(*S*) (operation)

▷ `IteratorOfLClassReps(S)` (operation)
▷ `IteratorOfRClassReps(S)` (operation)

Returns: An iterator.

Returns an iterator of the representatives of the Green's classes contained in the semigroup `S`. See (**Reference: Iterators**) for more information on iterators.

See also `GreensRClasses` (**Reference: GreensRClasses**), `GreensRClasses` (12.1.4), and `IteratorOfRClasses` (12.2.2).

Example

```
gap> S := Semigroup(Transformation([3, 2, 1, 5, 4]),
>                   Transformation([5, 4, 3, 2, 1]),
>                   Transformation([5, 4, 3, 2, 1]),
>                   Transformation([5, 5, 4, 5, 1]),
>                   Transformation([4, 5, 4, 3, 3]));;
gap> iter := IteratorOfRClassReps(S);
<iterator of R-class reps>
gap> NextIterator(iter);
Transformation( [ 3, 2, 1, 5, 4 ] )
gap> NextIterator(iter);
Transformation( [ 5, 5, 4, 5, 1 ] )
gap> iter;
<iterator of R-class reps>
gap> file := PackageInfo("semigroups")[1]!.InstallationPath;;
gap> file := Concatenation(file, "/data/doc/greens.pickle");;
gap> S := InverseSemigroup(ReadGenerators(file, 1));
<inverse partial perm semigroup of rank 983 with 2 generators>
gap> NrMovedPoints(S);
983
gap> iter := IteratorOfLClassReps(S);
<iterator of L-class reps>
gap> NextIterator(iter);
<partial perm on 634 pts with degree 1000, codegree 1000>
```

12.2.2 IteratorOfXClasses

▷ `IteratorOfDClasses(S)` (operation)
▷ `IteratorOfHClasses(S)` (operation)
▷ `IteratorOfLClasses(S)` (operation)
▷ `IteratorOfRClasses(S)` (operation)

Returns: An iterator.

Returns an iterator of the Green's classes in the semigroup `S`. See (**Reference: Iterators**) for more information on iterators.

This function is useful if you are, for example, looking for an $\mathscr{R}$-class of a semigroup with a particular property but do not necessarily want to compute all of the $\mathscr{R}$-classes.

See also `GreensRClasses` (12.1.4), `GreensRClasses` (**Reference: GreensRClasses**), `NrRClasses` (12.1.9), and `IteratorOfRClassReps` (12.2.1).

The transformation semigroup in the example below has 25147892 elements but it only takes a fraction of a second to find a non-trivial $\mathscr{R}$-class. The inverse semigroup of partial permutations in the example below has size 158122047816 but it only takes a fraction of a second to find an $\mathscr{R}$-class with more than 1000 elements.

Example

```
gap> gens := [Transformation([2, 4, 1, 5, 4, 4, 7, 3, 8, 1]),
>             Transformation([3, 2, 8, 8, 4, 4, 8, 6, 5, 7]),
>             Transformation([4, 10, 6, 6, 1, 2, 4, 10, 9, 7]),
>             Transformation([6, 2, 2, 4, 9, 9, 5, 10, 1, 8]),
>             Transformation([6, 4, 1, 6, 6, 8, 9, 6, 2, 2]),
>             Transformation([6, 8, 1, 10, 6, 4, 9, 1, 9, 4]),
>             Transformation([8, 6, 2, 3, 3, 4, 8, 6, 2, 9]),
>             Transformation([9, 1, 2, 8, 1, 5, 9, 9, 9, 5]),
>             Transformation([9, 3, 1, 5, 10, 3, 4, 6, 10, 2]),
>             Transformation([10, 7, 3, 7, 1, 9, 8, 8, 4, 10])];;
gap> S := Semigroup(gens);;
gap> iter := IteratorOfRClasses(S);
<iterator of R-classes>
gap> for R in iter do
>   if Size(R) > 1 then
>     break;
>   fi;
> od;
gap> R;
<Green's R-class: Transformation( [ 6, 4, 1, 6, 6, 8, 9, 6, 2, 2 ] )>
gap> Size(R);
21600
gap> S := InverseSemigroup(
>  PartialPerm([1, 2, 3, 4, 5, 6, 7, 10, 11, 19, 20],
>              [19, 4, 11, 15, 3, 20, 1, 14, 8, 13, 17]),
>  PartialPerm([1, 2, 3, 4, 6, 7, 8, 14, 15, 16, 17],
>              [15, 14, 20, 19, 4, 5, 1, 13, 11, 10, 3]),
>  PartialPerm([1, 2, 4, 6, 7, 8, 9, 10, 14, 15, 18],
>              [7, 2, 17, 10, 1, 19, 9, 3, 11, 16, 18]),
>  PartialPerm([1, 2, 3, 4, 5, 7, 8, 9, 11, 12, 13, 16],
>              [8, 3, 18, 1, 4, 13, 12, 7, 19, 20, 2, 11]),
>  PartialPerm([1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 16, 17, 20],
>              [7, 17, 13, 4, 6, 9, 18, 10, 11, 19, 5, 2, 8]),
>  PartialPerm([1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 18],
>              [10, 20, 11, 7, 13, 8, 4, 9, 2, 18, 17, 6, 15]),
>  PartialPerm([1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 14, 17, 18],
>              [10, 20, 18, 1, 14, 16, 9, 5, 15, 4, 8, 12, 19, 11]),
>  PartialPerm([1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 15, 16, 19, 20],
>              [13, 6, 1, 2, 11, 7, 16, 18, 9, 10, 4, 14, 15, 5, 17]),
>  PartialPerm([1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 20],
>              [5, 3, 12, 9, 20, 15, 8, 16, 13, 1, 17, 11, 14, 10, 2]),
>  PartialPerm([1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 17, 18, 19, 20],
>              [8, 3, 9, 20, 2, 12, 14, 15, 4, 18, 13, 1, 17, 19, 5]));;
gap> iter := IteratorOfRClasses(S);
<iterator of R-classes>
gap> repeat
>   R := NextIterator(iter);
> until Size(R) > 1000;
gap> R;
<Green's R-class: [8,3][11,5][13,1][15,2][17,6][19,7]>
gap> Size(R);
```

```
10020240
```

12.3 Properties of Green's classes

In this section, we describe the properties and operators of Green's classes that are available in the Semigroups package

12.3.1 Less than for Green's classes

▷ `\<(left-expr, right-expr)` (method)

Returns: `true` or `false`.

The Green's class *left-expr* is less than or equal to *right-expr* if they belong to the same semigroup and the representative of *left-expr* is less than the representative of *right-expr* under <; see also `Representative` (**Reference: Representative**).

Please note that this is not the usual order on the Green's classes of a semigroup as defined in (**Reference: Green's Relations**). See also `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**).

Example
```
gap> S := FullTransformationSemigroup(4);;
gap> A := GreensRClassOfElement(S, Transformation([2, 1, 3, 1]));
<Green's R-class: Transformation( [ 2, 1, 3, 1 ] )>
gap> B := GreensRClassOfElement(S, Transformation([1, 2, 3, 4]));
<Green's R-class: IdentityTransformation>
gap> A < B;
false
gap> B < A;
true
gap> IsGreensLessThanOrEqual(A, B);
true
gap> IsGreensLessThanOrEqual(B, A);
false
gap> S := SymmetricInverseSemigroup(4);;
gap> A := GreensJClassOfElement(S, PartialPerm([1, 3, 4]));;
gap> B := GreensJClassOfElement(S, PartialPerm([3, 1]));;
gap> A < B;
true
gap> B < A;
false
gap> IsGreensLessThanOrEqual(A, B);
false
gap> IsGreensLessThanOrEqual(B, A);
true
```

12.3.2 IsRegularGreensClass

▷ `IsRegularGreensClass(class)` (property)

Returns: `true` or `false`.

This function returns `true` if *class* is a regular Green's class and `false` if it is not. See also `IsRegularDClass` (**Reference: IsRegularDClass**), `IsGroupHClass`

(**Reference: IsGroupHClass**), `GroupHClassOfGreensDClass` (**Reference: GroupHClassOfGreensDClass**), `GroupHClass` (12.4.1), `NrIdempotents` (13.9.2), `Idempotents` (13.9.1), and `IsRegularSemigroupElement` (**Reference: IsRegularSemigroupElement**).

The function `IsRegularDClass` produces the same output as the GAP library functions with the same name; see `IsRegularDClass` (**Reference: IsRegularDClass**).

Example

```
gap> S := Monoid(Transformation([10, 8, 7, 4, 1, 4, 10, 10, 7, 2]),
>                Transformation([5, 2, 5, 5, 9, 10, 8, 3, 8, 10]));;
gap> f := Transformation([1, 1, 10, 8, 8, 8, 1, 1, 10, 8]);;
gap> R := RClass(S, f);;
gap> IsRegularGreensClass(R);
true
gap> S := Monoid(Transformation([2, 3, 4, 5, 1, 8, 7, 6, 2, 7]),
>                Transformation([3, 8, 7, 4, 1, 4, 3, 3, 7, 2]));;
gap> f := Transformation([3, 8, 7, 4, 1, 4, 3, 3, 7, 2]);;
gap> R := RClass(S, f);;
gap> IsRegularGreensClass(R);
false
gap> NrIdempotents(R);
0
gap> S := Semigroup(Transformation([2, 1, 3, 1]),
>                   Transformation([3, 1, 2, 1]),
>                   Transformation([4, 2, 3, 3]));;
gap> f := Transformation([4, 2, 3, 3]);;
gap> L := GreensLClassOfElement(S, f);;
gap> IsRegularGreensClass(L);
false
gap> R := GreensRClassOfElement(S, f);;
gap> IsRegularGreensClass(R);
false
gap> g := Transformation([4, 4, 4, 4]);;
gap> IsRegularSemigroupElement(S, g);
true
gap> IsRegularGreensClass(LClass(S, g));
true
gap> IsRegularGreensClass(RClass(S, g));
true
gap> IsRegularDClass(DClass(S, g));
true
gap> DClass(S, g) = RClass(S, g);
false
```

12.3.3 IsGreensClassNC

▷ `IsGreensClassNC(class)` (property)

Returns: `true` or `false`.

A Green's class *class* of a semigroup S satisfies `IsGreensClassNC` if it was not known to GAP that the representative of *class* was an element of S at the point that *class* was created.

12.4 Attributes of Green's classes

In this section, we describe the attributes of Green's classes that are available in the Semigroups package

12.4.1 GroupHClass

▷ `GroupHClass(class)` (attribute)

Returns: A group $\mathscr{H}$-class of the $\mathscr{D}$-class *class* if it is regular and `fail` if it is not.

`GroupHClass` is a synonym for `GroupHClassOfGreensDClass` (**Reference: GroupHClassOfGreensDClass**).

See also `IsGroupHClass` (**Reference: IsGroupHClass**), `IsRegularDClass` (**Reference: IsRegularDClass**), `IsRegularGreensClass` (12.3.2), and `IsRegularSemigroup` (14.1.16).

Example

```
gap> S := Semigroup(Transformation([2, 6, 7, 2, 6, 1, 1, 5]),
>                   Transformation([3, 8, 1, 4, 5, 6, 7, 1]));;
gap> IsRegularSemigroup(S);
false
gap> iter := IteratorOfDClasses(S);;
gap> repeat D := NextIterator(iter); until IsRegularDClass(D);
gap> D;
<Green's D-class: Transformation( [ 6, 1, 1, 6, 1, 2, 2, 6 ] )>
gap> NrIdempotents(D);
12
gap> NrRClasses(D);
8
gap> NrLClasses(D);
4
gap> GroupHClass(D);
<Green's H-class: Transformation( [ 1, 2, 2, 1, 2, 6, 6, 1 ] )>
gap> GroupHClassOfGreensDClass(D);
<Green's H-class: Transformation( [ 1, 2, 2, 1, 2, 6, 6, 1 ] )>
gap> StructureDescription(GroupHClass(D));
"S3"
gap> repeat D := NextIterator(iter); until not IsRegularDClass(D);
gap> D;
<Green's D-class: Transformation( [ 7, 5, 2, 2, 6, 1, 1, 2 ] )>
gap> IsRegularDClass(D);
false
gap> GroupHClass(D);
fail
gap> S := InverseSemigroup(
> PartialPerm([2, 1, 6, 0, 3]), PartialPerm([3, 5, 2, 0, 0, 6]));;
gap> x := PartialPerm([1 .. 3], [6, 3, 1]);;
gap> First(DClasses(S), x -> not IsTrivial(GroupHClass(x)));
<Green's D-class: <identity partial perm on [ 1, 2 ]>>
gap> StructureDescription(GroupHClass(last));
"C2"
```

12.4.2 SchutzenbergerGroup

▷ `SchutzenbergerGroup(class)` (attribute)

Returns: A group.

`SchutzenbergerGroup` returns the generalized Schutzenberger group (defined below) of the $\mathscr{R}$-, $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{H}$-class `class`.

If `f` is an element of a semigroup of transformations or partial permutations and `im(f)` denotes the image of `f`, then the *generalized Schutzenberger group* of `im(f)` is the permutation group

$$\{ g|_{\mathrm{im}(f)} : \mathrm{im}(f * g) = \mathrm{im}(f) \}.$$

The generalized Schutzenberger group of the kernel `ker(f)` of a transformation `f` or the domain `dom(f)` of a partial permutation `f` is defined analogously.

The generalized Schutzenberger group of a Green's class is then defined as follows.

$\mathscr{R}$-class

The generalized Schutzenberger group of the image or range of the representative of the $\mathscr{R}$-class.

$\mathscr{L}$-class

The generalized Schutzenberger group of the kernel or domain of the representative of the $\mathscr{L}$-class.

$\mathscr{H}$-class

The intersection of the generalized Schutzenberger groups of the $\mathscr{R}$- and $\mathscr{L}$-class containing the $\mathscr{H}$-class.

$\mathscr{D}$-class

The intersection of the generalized Schutzenberger groups of the $\mathscr{R}$- and $\mathscr{L}$-class containing the representative of the $\mathscr{D}$-class.

The output of this attribute is difficult to describe for other types of semigroup. However, a general description is given in [EENMP15].

Example

```
gap> S := Semigroup(Transformation([4, 4, 3, 5, 3]),
>                   Transformation([5, 1, 1, 4, 1]),
>                   Transformation([5, 5, 4, 4, 5]));;
gap> f := Transformation([5, 5, 4, 4, 5]);;
gap> SchutzenbergerGroup(RClass(S, f));
Group([ (4,5) ])
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 7],
>             [9, 2, 4, 8]),
> PartialPerm([1, 2, 6, 7, 8, 9, 10],
>             [6, 8, 4, 5, 9, 1, 3]),
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 9],
>             [7, 4, 1, 6, 9, 5, 2, 3]));;
gap> List(DClasses(S), SchutzenbergerGroup);
[ Group(()), Group(()), Group(()), Group(()), Group([ (4,9) ]),
  Group(()), Group(()), Group([ (5,8,6), (5,8) ]), Group(()),
  Group(()), Group(()), Group(()), Group(()), Group(()),
  Group([ (1,7,5,6,9,3) ]), Group([ (1,6)(3,5) ]), Group(()),
  Group(()), Group(()), Group(()), Group(()), Group(()), Group(()) ]
```

12.4.3 StructureDescriptionSchutzenbergerGroups

▷ `StructureDescriptionSchutzenbergerGroups(S)` (attribute)

Returns: Distinct structure descriptions of the Schutzenberger groups of a semigroup.

`StructureDescriptionSchutzenbergerGroups` returns the distinct values of `StructureDescription` (**Reference: StructureDescription**) when it is applied to the Schutzenberger groups of the $\mathscr{R}$-classes of the semigroup S.

Example
```
gap> S := Semigroup([
>  PartialPerm([1, 2, 3], [2, 5, 4]),
>  PartialPerm([1, 2, 3], [4, 1, 2]),
>  PartialPerm([1, 2, 3], [5, 2, 3]),
>  PartialPerm([1, 2, 4, 5], [2, 1, 4, 3]),
>  PartialPerm([1, 2, 5], [2, 3, 5]),
>  PartialPerm([1, 2, 3, 5], [2, 3, 5, 4]),
>  PartialPerm([1, 2, 3, 5], [4, 2, 5, 1]),
>  PartialPerm([1, 2, 3, 5], [5, 2, 4, 3]),
>  PartialPerm([1, 2, 5], [5, 4, 3])]);;
gap> StructureDescriptionSchutzenbergerGroups(S);
[ "1", "C2", "S3" ]
gap> S := Monoid(
> Bipartition([[1, 2, 5, -1, -2], [3, 4, -3, -5], [-4]]),
> Bipartition([[1, 2, -2], [3, -1], [4], [5], [-3, -4], [-5]]),
> Bipartition([[1], [2, 3, -5], [4, -3], [5, -2], [-1, -4]]));
<bipartition monoid of degree 5 with 3 generators>
gap> StructureDescriptionSchutzenbergerGroups(S);
[ "1", "C2" ]
```

12.4.4 StructureDescriptionMaximalSubgroups

▷ `StructureDescriptionMaximalSubgroups(S)` (attribute)

Returns: Distinct structure descriptions of the maximal subgroups of a semigroup.

`StructureDescriptionMaximalSubgroups` returns the distinct values of `StructureDescription` (**Reference: StructureDescription**) when it is applied to the maximal subgroups of the semigroup S.

Example
```
gap> S := DualSymmetricInverseSemigroup(6);
<inverse block bijection monoid of degree 6 with 3 generators>
gap> StructureDescriptionMaximalSubgroups(S);
[ "1", "C2", "S3", "S4", "S5", "S6" ]
gap> S := Semigroup(
>  PartialPerm([1, 3, 4, 5, 8],
>              [8, 3, 9, 4, 5]),
>  PartialPerm([1, 2, 3, 4, 8],
>              [10, 4, 1, 9, 6]),
>  PartialPerm([1, 2, 3, 4, 5, 6, 7, 10],
>              [4, 1, 6, 7, 5, 3, 2, 10]),
>  PartialPerm([1, 2, 3, 4, 6, 8, 10],
>              [4, 9, 10, 3, 1, 5, 2]));;
gap> StructureDescriptionMaximalSubgroups(S);
[ "1", "C2", "C3", "C4" ]
```

12.4.5 MultiplicativeNeutralElement (for an H-class)

▷ `MultiplicativeNeutralElement(H)` (method)

Returns: A semigroup element or `fail`.

If the $\mathscr{H}$-class `H` of a semigroup `S` is a subgroup of `S`, then `MultiplicativeNeutralElement` returns the identity of `H`. If `H` is not a subgroup of `S`, then `fail` is returned.

Example
```
gap> S := Semigroup([PartialPerm([1, 5, 2]),
>  PartialPerm([2, 0, 4]), PartialPerm([4, 1, 5]),
>  PartialPerm([1, 0, 3, 0, 4]), PartialPerm([1, 2, 0, 3, 5]),
>  PartialPerm([1, 3, 2, 0, 5]), PartialPerm([5, 0, 0, 4, 3])]);;
gap> H := HClass(S, PartialPerm([1, 2]));;
gap> MultiplicativeNeutralElement(H);
<identity partial perm on [ 1, 2 ]>
gap> H := HClass(S, PartialPerm([1, 4]));;
gap> MultiplicativeNeutralElement(H);
fail
```

12.4.6 StructureDescription (for an H-class)

▷ `StructureDescription(class)` (attribute)

Returns: A string or `fail`.

`StructureDescription` returns the value of `StructureDescription` (**Reference: StructureDescription**) when it is applied to a group isomorphic to the group $\mathscr{H}$-class *`class`*. If *`class`* is not a group $\mathscr{H}$-class, then `fail` is returned.

Example
```
gap> S := Semigroup(
> PartialPerm([1, 2, 3, 4, 6, 7, 8, 9],
>             [1, 9, 4, 3, 5, 2, 10, 7]),
> PartialPerm([1, 2, 4, 7, 8, 9],
>             [6, 2, 4, 9, 1, 3]));;
gap> H := HClass(S, PartialPerm([1, 2, 3, 4, 7, 9],
>                               [1, 7, 3, 4, 9, 2]));;
gap> StructureDescription(H);
"C6"
```

12.4.7 InjectionPrincipalFactor

▷ `InjectionPrincipalFactor(D)` (attribute)
▷ `InjectionNormalizedPrincipalFactor(D)` (attribute)
▷ `IsomorphismReesMatrixSemigroup(D)` (attribute)

Returns: A injective mapping.

If the $\mathscr{D}$-class `D` is a subsemigroup of a semigroup `S`, then the *principal factor* of `D` is just `D` itself. If `D` is not a subsemigroup of `S`, then the principal factor of `D` is the semigroup with elements `D` and a new element `0` with multiplication of $x, y \in D$ defined by:

$$xy = \begin{cases} x * y \text{ (in } S) & \text{if } x * y \in D \\ 0 & \text{if } xy \notin D. \end{cases}$$

`InjectionPrincipalFactor` returns an injective function from the $\mathscr{D}$-class `D` to a Rees (0-)matrix semigroup, which contains the principal factor of `D` as a subsemigroup.

If *D* is a subsemigroup of its parent semigroup, then the function returned by `InjectionPrincipalFactor` or `IsomorphismReesMatrixSemigroup` is an isomorphism from *D* to a Rees matrix semigroup; see `ReesMatrixSemigroup` (**Reference: ReesMatrixSemigroup**).

If *D* is not a semigroup, then the function returned by `InjectionPrincipalFactor` is an injective function from *D* to a Rees 0-matrix semigroup isomorphic to the principal factor of *D*; see `ReesZeroMatrixSemigroup` (**Reference: ReesZeroMatrixSemigroup**). In this case, `IsomorphismReesMatrixSemigroup` and `IsomorphismReesZeroMatrixSemigroup` returns an error.

`InjectionNormalizedPrincipalFactor` returns the composition of `InjectionPrincipalFactor` with `RZMSNormalization` (6.5.6) or `RMSNormalization` (6.5.7) as appropriate.

See also `PrincipalFactor` (12.4.8).

Example

```
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 6, 8, 10],
>             [2, 6, 7, 9, 1, 5]),
> PartialPerm([1, 2, 3, 4, 6, 7, 8, 10],
>             [3, 8, 1, 9, 4, 10, 5, 6]));;
gap> x := PartialPerm([1, 2, 5, 6, 7, 9],
>                     [1, 2, 5, 6, 7, 9]);;
gap> D := GreensDClassOfElement(S, x);
<Green's D-class: <identity partial perm on [ 1, 2, 5, 6, 7, 9 ]>>
gap> R := Range(InjectionPrincipalFactor(D));
<Rees 0-matrix semigroup 3x3 over Group(())>
gap> MatrixOfReesZeroMatrixSemigroup(R);
[ [ (), 0, 0 ], [ 0, (), 0 ], [ 0, 0, () ] ]
gap> Size(R);
10
gap> Size(D);
9
gap> S := Semigroup(
> Bipartition([[1, 2, 3, -3, -5], [4], [5, -2], [-1, -4]]),
> Bipartition([[1, 3, 5], [2, 4, -3], [-1, -2, -4, -5]]),
> Bipartition([[1, 5, -2, -4], [2, 3, 4, -1, -5], [-3]]),
> Bipartition([[1, 5, -1, -2, -3], [2, 4, -4], [3, -5]]));;
gap> D := GreensDClassOfElement(S,
> Bipartition([[1, 5, -2, -4], [2, 3, 4, -1, -5], [-3]]));
<Green's D-class: <bipartition: [ 1, 5, -2, -4 ], [ 2, 3, 4, -1, -5 ]
   , [ -3 ]>>
gap> InjectionNormalizedPrincipalFactor(D);
MappingByFunction( <Green's D-class: <bipartition: [ 1, 5, -2, -4 ],
  [ 2, 3, 4, -1, -5 ], [ -3 ]>>, <Rees matrix semigroup 1x1 over
  Group([ (1,2) ])>, function( x ) ... end, function( x ) ... end )
```

12.4.8 PrincipalFactor

▷ `PrincipalFactor(D)` (attribute)

▷ `NormalizedPrincipalFactor(D)` (attribute)

Returns: A Rees (0-)matrix semigroup.

If *D* is a $\mathscr{D}$-class of semigroup, then `PrincipalFactor(D)` is just shorthand for

Range(InjectionPrincipalFactor(*D*)), and NormalizedPrincipalFactor(*D*) is shorthand for Range(InjectionNormalizedPrincipalFactor(*D*)).

See InjectionPrincipalFactor (12.4.7) and InjectionNormalizedPrincipalFactor (12.4.7) for more details.

Example

```
gap> S := Semigroup([PartialPerm([1, 2, 3], [1, 3, 4]),
>  PartialPerm([1, 2, 3], [2, 5, 3]),
>  PartialPerm([1, 2, 3, 4], [2, 4, 1, 5]),
>  PartialPerm([1, 3, 5], [5, 1, 3])]);;
gap> PrincipalFactor(MinimalDClass(S));
<Rees matrix semigroup 1x1 over Group(())>
gap> MultiplicativeZero(S);
<empty partial perm>
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 4, 5, -1, -3], [-2, -5], [-4]]),
> Bipartition([[1, -5], [2, 3, 4, 5, -1, -3], [-2, -4]]),
> Bipartition([[1, 5, -4], [2, 4, -1, -5], [3, -2, -3]]));;
gap> D := MinimalDClass(S);
<Green's D-class: <bipartition: [ 1, 2, 3, 4, 5, -1, -3 ],
  [ -2, -5 ], [ -4 ]>>
gap> NormalizedPrincipalFactor(D);
<Rees matrix semigroup 1x5 over Group(())>
```

Chapter 13

Attributes and operations for semigroups

In this chapter we decribe the methods that are available in Semigroups for determining the attributes of a semigroup, and the operations which can be applied to a semigroup.

13.1 Accessing the elements of a semigroup

13.1.1 AsListCanonical

▷ `AsListCanonical(S)` (attribute)
▷ `EnumeratorCanonical(S)` (attribute)
▷ `IteratorCanonical(S)` (operation)

Returns: A list, enumerator, or iterator.

When the argument S is a semigroup in the representation `IsEnumerableSemigroupRep` (6.1.4), `AsListCanonical` returns a list of the elements of S in the order they are enumerated by the Froidure-Pin Algorithm. This is the same as the order used to index the elements of S in `RightCayleyGraphSemigroup` (13.2.1) and `LeftCayleyGraphSemigroup` (13.2.1).

`EnumeratorCanonical` and `IteratorCanonical` return an enumerator and an iterator where the elements are ordered in the same way as `AsListCanonical`. Using `EnumeratorCanonical` or `IteratorCanonical` will often use less memory than `AsListCanonical`, but may have slightly worse performance if the elements of the semigroup are looped over repeatedly. `EnumeratorCanonical` returns the same list as `AsListCanonical` if `AsListCanonical` has ever been called for S.

If S is an acting semigroup, then the value returned by `AsList` may not equal the value returned by `AsListCanonical`. `AsListCanonical` exists so that there is a method for obtaining the elements of S in the particular order used by `RightCayleyGraphSemigroup` (13.2.1) and `LeftCayleyGraphSemigroup` (13.2.1).

See also `PositionCanonical` (13.1.2).

Example

```
gap> S := Semigroup(Transformation([1, 3, 2]));;
gap> AsListCanonical(S);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> IteratorCanonical(S);
<iterator>
gap> EnumeratorCanonical(S);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> S := Monoid([Matrix(IsBooleanMat, [[1, 0, 0],
```

```
>                                       [0, 1, 0],
>                                       [0, 1, 0]])]);
<commutative monoid of 3x3 boolean matrices with 1 generator>
gap> it := IteratorCanonical(S);
<iterator>
gap> NextIterator(it);
Matrix(IsBooleanMat, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
gap> en := EnumeratorCanonical(S);
<enumerator of <commutative monoid of 3x3 boolean matrices with 1
 generator>>
gap> en[1];
Matrix(IsBooleanMat, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
gap> Position(en, en[1]);
1
gap> Length(en);
2
```

13.1.2 PositionCanonical

▷ `PositionCanonical(S, x)` (operation)

Returns: `true` or `false`.

When the argument *S* is a semigroup in the representation `IsEnumerableSemigroupRep` (6.1.4) and *x* is an element of *S*, `PositionCanonical` returns the position of *x* in `AsListCanonical(S)` or equivalently the position of *x* in `EnumeratorCanonical(S)`.

See also `AsListCanonical` (13.1.1) and `EnumeratorCanonical` (13.1.1).

Example
```
gap> S := FullTropicalMaxPlusMonoid(2, 3);
<monoid of 2x2 tropical max-plus matrices with 13 generators>
gap> x := Matrix(IsTropicalMaxPlusMatrix, [[1, 3], [2, 1]], 3);
Matrix(IsTropicalMaxPlusMatrix, [[1, 3], [2, 1]], 3)
gap> PositionCanonical(S, x);
234
gap> EnumeratorCanonical(S)[234] = x;
true
```

13.1.3 Enumerate

▷ `Enumerate(S[, limit])` (operation)

Returns: A semigroup (the argument).

If *S* is a semigroup with representation `IsEnumerableSemigroupRep` (6.1.4) and *limit* is a positive integer, then this operation can be used to enumerate at least *limit* elements of *S*, or `Size(S)` elements if this is less than *limit*, using the Froidure-Pin Algorithm.

If the optional second argument *limit* is not given, then the semigroup is enumerated until all of its elements have been found.

For reasons of performance, *S* is enumerated in batches according to the option `batch_size`, which can be specified when *S* is created; see Section 6.3.

Example
```
gap> S := FullTransformationMonoid(7);
<full transformation monoid of degree 7>
gap> Enumerate(S, 1000);
```

```
<full transformation monoid of degree 7>
gap> Display(S);
<partially enumerated semigroup with 8197 elements,
224 rules, max word length 11>
```

13.1.4 IsFullyEnumerated

▷ IsFullyEnumerated(*S*) (operation)

Returns: true or false.

If *S* is a semigroup with representation IsEnumerableSemigroupRep (6.1.4), then this operation returns true if the Froidure-Pin Algorithm has been run to completion (i.e. all of the elements of *S* have been found) and false if *S* has not been fully enumerated.

Example

```
gap> S := FullBooleanMatMonoid(4);
<monoid of 4x4 boolean matrices with 7 generators>
gap> Enumerate(S, 1000);
<monoid of 4x4 boolean matrices with 7 generators>
gap> IsFullyEnumerated(S);
false
gap> Size(S);
65536
gap> IsFullyEnumerated(S);
true
```

13.2 Cayley graphs

13.2.1 RightCayleyGraphSemigroup

▷ RightCayleyGraphSemigroup(*S*) (attribute)

▷ LeftCayleyGraphSemigroup(*S*) (attribute)

Returns: A list of lists of positive integers.

When the argument *S* is a semigroup in the representation IsEnumerableSemigroupRep (6.1.4), RightCayleyGraphSemigroup returns the right Cayley graphs of *S*, as a list graph where graph[i][j] is equal to PositionCanonical(*S*, AsListCanonical(*S*)[i] * GeneratorsOfSemigroup(*S*)[j]). The list returned by LeftCayleyGraphSemigroup is defined analogously.

Example

```
gap> S := FullTransformationMonoid(2);
<full transformation monoid of degree 2>
gap> RightCayleyGraphSemigroup(S);
[ [ 1, 2, 3 ], [ 2, 1, 3 ], [ 3, 4, 3 ], [ 4, 3, 3 ] ]
gap> LeftCayleyGraphSemigroup(S);
[ [ 1, 2, 3 ], [ 2, 1, 4 ], [ 3, 3, 3 ], [ 4, 4, 4 ] ]
```

13.3 Random elements of a semigroup

13.3.1 Random (for a semigroup)

▷ `Random(S)` (method)

Returns: A random element.

This function returns a random element of the semigroup *S*. If the elements of *S* have been calculated, then one of these is chosen randomly. Otherwise, if the data structure for *S* is known, then a random element of a randomly chosen $\mathscr{R}$-class is returned. If the data structure for *S* has not been calculated, then a short product (at most `2 * Length(GeneratorsOfSemigroup(S))`) of generators is returned.

13.4 Properties of elements in a semigroup

13.4.1 IndexPeriodOfSemigroupElement

▷ `IndexPeriodOfSemigroupElement(x)` (operation)

Returns: A list of two positive integers.

If *x* is a semigroup element, then `IndexPeriodOfSemigroupElement(x)` returns the pair `[m, r]`, where `m` and `r` are the least positive integers such that `x^(m + r) = x ^ m`. The number `m` is known as the *index* of *x*, and the numberr is known as the *period* of *x*.

Example
```
gap> x := Transformation([2, 6, 3, 5, 6, 1]);;
gap> IndexPeriodOfSemigroupElement(x);
[ 2, 3 ]
gap> m := IndexPeriodOfSemigroupElement(x)[1];;
gap> r := IndexPeriodOfSemigroupElement(x)[2];;
gap> x ^ (m + r) = x ^ m;
true
gap> x := PartialPerm([0, 2, 3, 0, 5]);
<identity partial perm on [ 2, 3, 5 ]>
gap> IsIdempotent(x);
true
gap> IndexPeriodOfSemigroupElement(x);
[ 1, 1 ]
```

13.4.2 SmallestIdempotentPower

▷ `SmallestIdempotentPower(x)` (attribute)

Returns: A positive integer.

If *x* is a semigroup element, then `SmallestIdempotentPower(x)` returns the least positive integer `n` such that `x^n` is an idempotent. The smallest idempotent power of *x* is the least multiple of the period of *x* that is greater than or equal to the index of *x*; see `IndexPeriodOfSemigroupElement` (13.4.1).

Example
```
gap> x := Transformation([4, 1, 4, 5, 1]);
Transformation( [ 4, 1, 4, 5, 1 ] )
gap> SmallestIdempotentPower(x);
3
gap> ForAll([1 .. 2], i -> not IsIdempotent(x ^ i));
```

```
true
gap> IsIdempotent(x ^ 3);
true
gap> x := Bipartition([[1, 2, -3, -4], [3, -5], [4, -1], [5, -2]]);
<block bijection: [ 1, 2, -3, -4 ], [ 3, -5 ], [ 4, -1 ], [ 5, -2 ]>
gap> SmallestIdempotentPower(x);
4
gap> ForAll([1 .. 3], i -> not IsIdempotent(x ^ i));
true
gap> x := PartialPerm([]);
<empty partial perm>
gap> SmallestIdempotentPower(x);
1
gap> IsIdempotent(x);
true
```

13.5 Expressing semigroup elements as words in generators

It is possible to express an element of a semigroup as a word in the generators of that semigroup. This section describes how to accomplish this in Semigroups.

13.5.1 EvaluateWord

▷ `EvaluateWord(gens, w)` (operation)

Returns: A semigroup element.

The argument *gens* should be a collection of generators of a semigroup and the argument *w* should be a list of positive integers less than or equal to the length of *gens*. This operation evaluates the word *w* in the generators *gens*. More precisely, `EvaluateWord(gens, w)` returns the equivalent of:

Example
```
Product(List(w, i -> gens[i]));
```

see also `Factorization` (13.5.2).

for elements of a semigroup

When *gens* is a list of elements of a semigroup and *w* is a list of positive integers less than or equal to the length of *gens*, this operation returns the product `gens[w[1]] * gens[w[2]] * .. . * gens[w[n]]` when the length of *w* is n.

for elements of an inverse semigroup

When *gens* is a list of elements with a semigroup inverse and *w* is a list of non-zero integers whose absolute value does not exceed the length of *gens*, this operation returns the product `gens[AbsInt(w[1])] ^ SignInt(w[1]) * .. . * gens[AbsInt(w[n])] ^ SignInt(w[n])` where n is the length of *w*.

Note that `EvaluateWord(gens, [])` returns `One(gens)` if *gens* belongs to the category `IsMultiplicativeElementWithOne` (**Reference: IsMultiplicativeElementWithOne**).

Example
```
gap> gens := [
> Transformation([2, 4, 4, 6, 8, 8, 6, 6]),
```

```
> Transformation([2, 7, 4, 1, 4, 6, 5, 2]),
> Transformation([3, 6, 2, 4, 2, 2, 2, 8]),
> Transformation([4, 3, 6, 4, 2, 1, 2, 6]),
> Transformation([4, 5, 1, 3, 8, 5, 8, 2])];;
gap> S := Semigroup(gens);;
gap> x := Transformation([1, 4, 6, 1, 7, 2, 7, 6]);;
gap> word := Factorization(S, x);
[ 4, 2 ]
gap> EvaluateWord(gens, word);
Transformation( [ 1, 4, 6, 1, 7, 2, 7, 6 ] )
gap> S := SymmetricInverseMonoid(10);;
gap> x := PartialPerm([2, 6, 7, 0, 0, 9, 0, 1, 0, 5]);
[3,7][8,1,2,6,9][10,5]
gap> word := Factorization(S, x);
[ -2, -2, -2, -2, -3, -2, -2, -2, -2, -2, 5, 2, 5, 5, 2, 5, 2, 2, 2,
  2, -3, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 3, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), word);
[3,7][8,1,2,6,9][10,5]
```

13.5.2 Factorization

▷ `Factorization(S, x)` (operation)

Returns: A word in the generators.

for semigroups

When S is a semigroup and x belongs to S, `Factorization` return a word in the generators of S that is equal to x. In this case, a word is a list of positive integers where an entry `i` corresponds to `GeneratorsOfSemigroups(S)[i]`. More specifically,

Example
```
EvaluateWord(GeneratorsOfSemigroup(S), Factorization(S, x)) = x;
```

for inverse semigroups

When S is an inverse semigroup and x belongs to S, `Factorization` return a word in the generators of S that is equal to x. In this case, a word is a list of non-zero integers where an entry `i` corresponds to `GeneratorsOfSemigroup(S)[i]` and `-i` corresponds to `GeneratorsOfSemigroup(S)[i] ^ -1`. As in the previous case,

Example
```
EvaluateWord(GeneratorsOfSemigroup(S), Factorization(S, x)) = x;
```

Note that `Factorization` does not always return a word of minimum length; see `MinimalFactorization` (13.5.3).

See also `EvaluateWord` (13.5.1) and `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**).

Example
```
gap> gens := [Transformation([2, 2, 9, 7, 4, 9, 5, 5, 4, 8]),
>             Transformation([4, 10, 5, 6, 4, 1, 2, 7, 1, 2])];;
gap> S := Semigroup(gens);;
gap> x := Transformation([1, 10, 2, 10, 1, 2, 7, 10, 2, 7]);;
gap> word := Factorization(S, x);
[ 2, 2, 1, 2 ]
```

```
gap> EvaluateWord(gens, word);
Transformation( [ 1, 10, 2, 10, 1, 2, 7, 10, 2, 7 ] )
gap> S := SymmetricInverseMonoid(8);
<symmetric inverse monoid of degree 8>
gap> x := PartialPerm([1, 2, 3, 4, 5, 8], [7, 1, 4, 3, 2, 6]);
[5,2,1,7][8,6](3,4)
gap> word := Factorization(S, x);
[ -2, -2, -2, -2, -2, -2, 2, 4, 4, 2, 3, 2, -3, -2, -2, 3, 2, -3, -2,
  -2, 4, -3, -4, 2, 2, 3, -2, -3, 4, -3, -4, 2, 2, 3, -2, -3, 2, 2,
  3, -2, -3, 2, 2, 3, -2, -3, 4, -3, -4, 3, 2, -3, -2, -2, 3, 2, -3,
  -2, -2, 4, 3, -4, 3, 2, -3, -2, -2, 3, 2, -3, -2, -2, 3, 2, 2, 3,
  2, 2, 2, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), word);
[5,2,1,7][8,6](3,4)
gap> S := DualSymmetricInverseMonoid(6);;
gap> x := S.1 * S.2 * S.3 * S.2 * S.1;
<block bijection: [ 1, 6, -4 ], [ 2, -2, -3 ], [ 3, -5 ], [ 4, -6 ],
 [ 5, -1 ]>
gap> word := Factorization(S, x);
[ -2, -2, -2, -2, -2, 4, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), word);
<block bijection: [ 1, 6, -4 ], [ 2, -2, -3 ], [ 3, -5 ], [ 4, -6 ],
 [ 5, -1 ]>
```

13.5.3 MinimalFactorization

▷ MinimalFactorization(S, x) (operation)

Returns: A minimal word in the generators.

This operation returns a minimal length word in the generators of the semigroup S that equals the element x. In this case, a word is a list of positive integers where an entry i corresponds to GeneratorsOfSemigroups(S)[i]. More specifically,

Example
```
EvaluateWord(GeneratorsOfSemigroup(S), MinimalFactorization(S, x)) = x;
```

MinimalFactorization involves exhaustively enumerating S until the element x is found, and so MinimalFactorization may be less efficient than Factorization (13.5.2) for some semigroups.

Unlike Factorization (13.5.2) this operation does not distinguish between semigroups and inverse semigroups. See also EvaluateWord (13.5.1) and GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**).

Example
```
gap> S := Semigroup(Transformation([2, 2, 9, 7, 4, 9, 5, 5, 4, 8]),
>                   Transformation([4, 10, 5, 6, 4, 1, 2, 7, 1, 2]));
<transformation semigroup of degree 10 with 2 generators>
gap> x := Transformation([8, 8, 2, 2, 9, 2, 8, 8, 9, 9]);
Transformation( [ 8, 8, 2, 2, 9, 2, 8, 8, 9, 9 ] )
gap> Factorization(S, x);
[ 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1 ]
gap> MinimalFactorization(S, x);
[ 1, 2, 1, 1, 1, 1, 2, 2, 1 ]
```

13.6 Generating sets

13.6.1 Generators

▷ `Generators(S)` (attribute)

Returns: A list of generators.

`Generators` returns a generating set that can be used to define the semigroup *S*. The generators of a monoid or inverse semigroup *S*, say, can be defined in several ways, for example, including or excluding the identity element, including or not the inverses of the generators. `Generators` uses the definition that returns the least number of generators. If no generating set for *S* is known, then `GeneratorsOfSemigroup` is used by default.

for a group
`Generators(S)` is a synonym for `GeneratorsOfGroup` (**Reference: GeneratorsOfGroup**).

for an ideal of semigroup
`Generators(S)` is a synonym for `GeneratorsOfSemigroupIdeal` (7.2.1).

for a semigroup
`Generators(S)` is a synonym for `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**).

for a monoid
`Generators(S)` is a synonym for `GeneratorsOfMonoid` (**Reference: GeneratorsOfMonoid**).

for an inverse semigroup
`Generators(S)` is a synonym for `GeneratorsOfInverseSemigroup` (**Reference: GeneratorsOfInverseSemigroup**).

for an inverse monoid
`Generators(S)` is a synonym for `GeneratorsOfInverseMonoid` (**Reference: GeneratorsOfInverseMonoid**).

Example

```
gap> M := Monoid([
> Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
> Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9])]);;
gap> GeneratorsOfSemigroup(M);
[ IdentityTransformation,
  Transformation( [ 1, 4, 6, 2, 5, 3, 7, 8, 9, 9 ] ),
  Transformation( [ 6, 3, 2, 7, 5, 1, 8, 8, 9, 9 ] ) ]
gap> GeneratorsOfMonoid(M);
[ Transformation( [ 1, 4, 6, 2, 5, 3, 7, 8, 9, 9 ] ),
  Transformation( [ 6, 3, 2, 7, 5, 1, 8, 8, 9, 9 ] ) ]
gap> Generators(M);
[ Transformation( [ 1, 4, 6, 2, 5, 3, 7, 8, 9, 9 ] ),
  Transformation( [ 6, 3, 2, 7, 5, 1, 8, 8, 9, 9 ] ) ]
gap> S := Semigroup(Generators(M));;
gap> Generators(S);
[ Transformation( [ 1, 4, 6, 2, 5, 3, 7, 8, 9, 9 ] ),
  Transformation( [ 6, 3, 2, 7, 5, 1, 8, 8, 9, 9 ] ) ]
```

```
gap> GeneratorsOfSemigroup(S);
[ Transformation( [ 1, 4, 6, 2, 5, 3, 7, 8, 9, 9 ] ),
  Transformation( [ 6, 3, 2, 7, 5, 1, 8, 8, 9, 9 ] ) ]
```

13.6.2 SmallGeneratingSet

▷ `SmallGeneratingSet(coll)` (attribute)
▷ `SmallSemigroupGeneratingSet(coll)` (attribute)
▷ `SmallMonoidGeneratingSet(coll)` (attribute)
▷ `SmallInverseSemigroupGeneratingSet(coll)` (attribute)
▷ `SmallInverseMonoidGeneratingSet(coll)` (attribute)

Returns: A small generating set for a semigroup.

The attributes `SmallXGeneratingSet` return a relatively small generating subset of the collection of elements *coll*, which can also be a semigroup. The returned value of `SmallXGeneratingSet`, where applicable, has the property that

Example
```
X(SmallXGeneratingSet(coll)) = X(coll);
```

where `X` is any of `Semigroup` (**Reference: Semigroup**), `Monoid` (**Reference: Monoid**), `InverseSemigroup` (**Reference: InverseSemigroup**), or `InverseMonoid` (**Reference: InverseMonoid**).

If the number of generators for *S* is already relatively small, then these functions will often return the original generating set. These functions may return different results in different GAP sessions.

`SmallGeneratingSet` returns the smallest of the returned values of `SmallXGeneratingSet` which is applicable to *coll*; see `Generators` (13.6.1).

As neither irredundancy, nor minimal length are proven, these functions usually return an answer much more quickly than `IrredundantGeneratingSubset` (13.6.3). These functions can be used whenever a small generating set is desired which does not necessarily needs to be minimal.

Example
```
gap> S := Semigroup([
> Transformation([1, 2, 3, 2, 4]),
> Transformation([1, 5, 4, 3, 2]),
> Transformation([2, 1, 4, 2, 2]),
> Transformation([2, 4, 4, 2, 1]),
> Transformation([3, 1, 4, 3, 2]),
> Transformation([3, 2, 3, 4, 1]),
> Transformation([4, 4, 3, 3, 5]),
> Transformation([5, 1, 5, 5, 3]),
> Transformation([5, 4, 3, 5, 2]),
> Transformation([5, 5, 4, 5, 5])]);;
gap> SmallGeneratingSet(S);
[ Transformation( [ 1, 5, 4, 3, 2 ] ), Transformation( [ 3, 2, 3, 4, 1 ] ),
  Transformation( [ 5, 4, 3, 5, 2 ] ), Transformation( [ 1, 2, 3, 2, 4 ] ),
  Transformation( [ 4, 4, 3, 3, 5 ] ) ]
gap> S := RandomInverseMonoid(IsPartialPermMonoid, 10000, 10);;
gap> SmallGeneratingSet(S);
[ [ 1 .. 10 ] -> [ 3, 2, 4, 5, 6, 1, 7, 10, 9, 8 ],
  [ 1 .. 10 ] -> [ 5, 10, 8, 9, 3, 2, 4, 7, 6, 1 ],
  [ 1, 3, 4, 5, 6, 7, 8, 9, 10 ] -> [ 1, 6, 4, 8, 2, 10, 7, 3, 9 ] ]
gap> M := MathieuGroup(24);;
```

```
gap> mat := List([1 .. 1000], x -> Random(M));;
gap> Append(mat, [1 .. 1000] * 0);
gap> mat := List([1 .. 138], x -> List([1 .. 57], x -> Random(mat)));;
gap> R := ReesZeroMatrixSemigroup(M, mat);;
gap> U := Semigroup(List([1 .. 200], x -> Random(R)));
<subsemigroup of 57x138 Rees 0-matrix semigroup with 100 generators>
gap> Length(SmallGeneratingSet(U));
84
gap> S := RandomSemigroup(IsBipartitionSemigroup, 100, 4);
<bipartition semigroup of degree 4 with 96 generators>
gap> Length(SmallGeneratingSet(S));
13
```

13.6.3 IrredundantGeneratingSubset

▷ IrredundantGeneratingSubset(*coll*) (operation)

Returns: A list of irredundant generators.

If *coll* is a collection of elements of a semigroup, then this function returns a subset U of *coll* such that no element of U is generated by the other elements of U.

Example

```
gap> S := Semigroup([
> Transformation([5, 1, 4, 6, 2, 3]),
> Transformation([1, 2, 3, 4, 5, 6]),
> Transformation([4, 6, 3, 4, 2, 5]),
> Transformation([5, 4, 6, 3, 1, 3]),
> Transformation([2, 2, 6, 5, 4, 3]),
> Transformation([3, 5, 5, 1, 2, 4]),
> Transformation([6, 5, 1, 3, 3, 4]),
> Transformation([1, 3, 4, 3, 2, 1])]);;
gap> IrredundantGeneratingSubset(S);
[ Transformation( [ 1, 3, 4, 3, 2, 1 ] ),
  Transformation( [ 2, 2, 6, 5, 4, 3 ] ),
  Transformation( [ 3, 5, 5, 1, 2, 4 ] ),
  Transformation( [ 5, 1, 4, 6, 2, 3 ] ),
  Transformation( [ 5, 4, 6, 3, 1, 3 ] ),
  Transformation( [ 6, 5, 1, 3, 3, 4 ] ) ]
gap> S := RandomInverseMonoid(IsPartialPermMonoid, 1000, 10);
<inverse partial perm monoid of degree 10 with 1000 generators>
gap> SmallGeneratingSet(S);
[ [ 1 .. 10 ] -> [ 6, 5, 1, 9, 8, 3, 10, 4, 7, 2 ],
  [ 1 .. 10 ] -> [ 1, 4, 6, 2, 8, 5, 7, 10, 3, 9 ],
  [ 1, 2, 3, 4, 6, 7, 8, 9 ] -> [ 7, 5, 10, 1, 8, 4, 9, 6 ]
  [ 1 .. 9 ] -> [ 4, 3, 5, 7, 10, 9, 1, 6, 8 ] ]
gap> IrredundantGeneratingSubset(last);
[ [ 1 .. 9 ] -> [ 4, 3, 5, 7, 10, 9, 1, 6, 8 ],
  [ 1 .. 10 ] -> [ 1, 4, 6, 2, 8, 5, 7, 10, 3, 9 ],
  [ 1 .. 10 ] -> [ 6, 5, 1, 9, 8, 3, 10, 4, 7, 2 ] ]
gap> S := RandomSemigroup(IsBipartitionSemigroup, 1000, 4);
<bipartition semigroup of degree 4 with 749 generators>
gap> SmallGeneratingSet(S);
[ <bipartition: [ 1, -3 ], [ 2, -2 ], [ 3, -1 ], [ 4, -4 ]>,
  <bipartition: [ 1, 3, -2 ], [ 2, -1, -3 ], [ 4, -4 ]>,
```

```
  <bipartition: [ 1, -4 ], [ 2, 4, -1, -3 ], [ 3, -2 ]>,
  <bipartition: [ 1, -1, -3 ], [ 2, -4 ], [ 3, 4, -2 ]>,
  <bipartition: [ 1, -2, -4 ], [ 2 ], [ 3, -3 ], [ 4, -1 ]>,
  <bipartition: [ 1, -2 ], [ 2, -1, -3 ], [ 3, 4, -4 ]>,
  <bipartition: [ 1, 3, -1 ], [ 2, -3 ], [ 4, -2, -4 ]>,
  <bipartition: [ 1, -1 ], [ 2, 4, -4 ], [ 3, -2, -3 ]>,
  <bipartition: [ 1, 3, -1 ], [ 2, -2 ], [ 4, -3, -4 ]>,
  <bipartition: [ 1, 2, -2 ], [ 3, -1, -4 ], [ 4, -3 ]>,
  <bipartition: [ 1, -2, -3 ], [ 2, -4 ], [ 3 ], [ 4, -1 ]>,
  <bipartition: [ 1, -1 ], [ 2, 4, -3 ], [ 3, -2 ], [ -4 ]>,
  <bipartition: [ 1, -3 ], [ 2, -1 ], [ 3, 4, -4 ], [ -2 ]>,
  <bipartition: [ 1, 2, -4 ], [ 3, -1 ], [ 4, -2 ], [ -3 ]>,
  <bipartition: [ 1, -3 ], [ 2, -4 ], [ 3, -1, -2 ], [ 4 ]> ]
gap> IrredundantGeneratingSubset(last);
[ <bipartition: [ 1, 2, -4 ], [ 3, -1 ], [ 4, -2 ], [ -3 ]>,
  <bipartition: [ 1, 3, -1 ], [ 2, -2 ], [ 4, -3, -4 ]>,
  <bipartition: [ 1, 3, -2 ], [ 2, -1, -3 ], [ 4, -4 ]>,
  <bipartition: [ 1, -1 ], [ 2, 4, -3 ], [ 3, -2 ], [ -4 ]>,
  <bipartition: [ 1, -3 ], [ 2, -1 ], [ 3, 4, -4 ], [ -2 ]>,
  <bipartition: [ 1, -3 ], [ 2, -2 ], [ 3, -1 ], [ 4, -4 ]>,
  <bipartition: [ 1, -3 ], [ 2, -4 ], [ 3, -1, -2 ], [ 4 ]>,
  <bipartition: [ 1, -2, -3 ], [ 2, -4 ], [ 3 ], [ 4, -1 ]>,
  <bipartition: [ 1, -2, -4 ], [ 2 ], [ 3, -3 ], [ 4, -1 ]> ]
```

13.6.4 MinimalSemigroupGeneratingSet

▷ `MinimalSemigroupGeneratingSet(S)` (attribute)
▷ `MinimalMonoidGeneratingSet(S)` (attribute)
▷ `MinimalInverseSemigroupGeneratingSet(S)` (attribute)
▷ `MinimalInverseMonoidGeneratingSet(S)` (attribute)

Returns: A minimal generating set for a semigroup.

WARNING: currently, no methods are installed to compute these attributes.

The attributes `MinimalXGeneratingSet` return a minimal generating set for the semigroup S, with respect to length. The returned value of `MinimalXGeneratingSet`, where applicable, is a minimal-length list of elements of S with the property that

Example
```
      X(MinimalXGeneratingSet(S)) = S;
```

where `X` is any of `Semigroup` (**Reference: Semigroup**), `Monoid` (**Reference: Monoid**), `InverseSemigroup` (**Reference: InverseSemigroup**), or `InverseMonoid` (**Reference: InverseMonoid**).

For certain types of semigroup, for example monogenic semigroups, a `MinimalXGeneratingSet` may be known a priori, or may be deduced as a by-product of other functions. However, since there are no methods installed to compute these attributes directly, for most semigroups it is not currently possible to find a `MinimalXGeneratingSet` with the Semigroups package.

See also `SmallGeneratingSet` (13.6.2) and `IrredundantGeneratingSubset` (13.6.3).

Example
```
gap> S := MonogenicSemigroup(3, 6);;
gap> MinimalSemigroupGeneratingSet(S);
```

```
[ Transformation( [ 2, 3, 4, 5, 6, 1, 6, 7, 8 ] ) ]
gap> S := Semigroup([
>  PartialPerm([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),
>  PartialPerm([1, 2, 3, 4], [5, 2, 4, 1]),
>  PartialPerm([1, 2, 4, 5], [4, 2, 3, 1])]);
<partial perm monoid of rank 5 with 2 generators>
gap> IsMonogenicInverseMonoid(S);
true
gap> MinimalInverseMonoidGeneratingSet(S);
[ [3,4,1,5](2) ]
```

13.6.5 GeneratorsSmallest (for a semigroup)

▷ `GeneratorsSmallest(S)` (attribute)

Returns: A set of elements.

For a semigroup S, `GeneratorsSmallest` returns the lexicographically least set of elements `X` such that `X` generates S as a semigroup, and such that `X` is lexicographically ordered and has the property that each `X[i]` is not generated by `X[1], X[2], ..., X[i-1]`.

It can be difficult to find the set of generators `X`, and it might contain a substantial proportion of the elements of S.

Two semigroups have the same set of elements if and only if their smallest generating sets are equal. However, due to the complexity of determining the `GeneratorsSmallest`, this is not the method used by the Semigroups package when comparing semigroups.

Example

```
gap> S := Monoid([
>  Transformation([1, 3, 4, 1]),
>  Transformation([2, 4, 1, 2]),
>  Transformation([3, 1, 1, 3]),
>  Transformation([3, 3, 4, 1])]);
<transformation monoid of degree 4 with 4 generators>
gap> GeneratorsSmallest(S);
[ Transformation( [ 1, 1, 1, 1 ] ), Transformation( [ 1, 1, 1, 2 ] ),
  Transformation( [ 1, 1, 1, 3 ] ), Transformation( [ 1, 1, 1 ] ),
  Transformation( [ 1, 1, 2, 1 ] ), Transformation( [ 1, 1, 2, 2 ] ),
  Transformation( [ 1, 1, 3, 1 ] ), Transformation( [ 1, 1, 3, 3 ] ),
  Transformation( [ 1, 1 ] ), Transformation( [ 1, 1, 4, 1 ] ),
  Transformation( [ 1, 2, 1, 1 ] ), Transformation( [ 1, 2, 2, 1 ] ),
  IdentityTransformation, Transformation( [ 1, 3, 1, 1 ] ),
  Transformation( [ 1, 3, 4, 1 ] ), Transformation( [ 2, 1, 1, 2 ] ),
  Transformation( [ 2, 2, 2 ] ), Transformation( [ 2, 4, 1, 2 ] ),
  Transformation( [ 3, 3, 3 ] ), Transformation( [ 3, 3, 4, 1 ] ) ]
gap> T := Semigroup(Bipartition([[1, 2, 3], [4, -1], [-2], [-3], [-4]]),
>                   Bipartition([[1, -3, -4], [2, 3, 4, -2], [-1]]),
>                   Bipartition([[1, 2, 3, 4, -2], [-1, -4], [-3]]),
>                   Bipartition([[1, 2, 3, 4], [-1], [-2], [-3, -4]]),
>                   Bipartition([[1, 2, -1, -2], [3, 4, -3], [-4]]));
<bipartition semigroup of degree 4 with 5 generators>
gap> GeneratorsSmallest(T);
[ <bipartition: [ 1, 2, 3, 4, -1, -2, -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -1, -2 ], [ -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -1 ], [ -2 ], [ -3 ], [ -4 ]>,
```

```
  <bipartition: [ 1, 2, 3, 4, -2, -3, -4 ], [ -1 ]>,
  <bipartition: [ 1, 2, 3, 4, -2 ], [ -1, -4 ], [ -3 ]>,
  <bipartition: [ 1, 2, 3, 4, -2 ], [ -1 ], [ -3, -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -3 ], [ -1, -2 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4 ], [ -1, -2, -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -3, -4 ], [ -1 ], [ -2 ]>,
  <bipartition: [ 1, 2, 3 ], [ 4, -1, -2, -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, -1, -2 ], [ 3, 4, -3 ], [ -4 ]>,
  <bipartition: [ 1, -3 ], [ 2, 3, 4, -1, -2 ], [ -4 ]>,
  <bipartition: [ 1, -3, -4 ], [ 2, 3, 4, -2 ], [ -1 ]> ]
```

13.7 Minimal ideals and multiplicative zeros

In this section we describe the attributes of a semigroup that can be found using the Semigroups package.

13.7.1 MinimalIdeal

▷ `MinimalIdeal(S)` (attribute)

Returns: The minimal ideal of a semigroup.

The minimal ideal of a semigroup is the least ideal with respect to containment.

It is significantly easier to find the minimal $\mathscr{D}$-class of a semigroup, than to find its $\mathscr{D}$-classes.

See also `RepresentativeOfMinimalIdeal` (13.7.2), `PartialOrderOfDClasses` (12.1.10), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), and `MinimalDClass` (12.1.6).

Example

```
gap> S := Semigroup(
> Transformation([3, 4, 1, 3, 6, 3, 4, 6, 10, 1]),
> Transformation([8, 2, 3, 8, 4, 1, 3, 4, 9, 7]));;
gap> MinimalIdeal(S);
<simple transformation semigroup ideal of degree 10 with 1 generator>
gap> Elements(MinimalIdeal(S));
[ Transformation( [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ),
  Transformation( [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ] ),
  Transformation( [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ] ),
  Transformation( [ 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ] ),
  Transformation( [ 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 ] ) ]
gap> x := Transformation([8, 8, 8, 8, 8, 8, 8, 8, 8, 8]);;
gap> D := DClass(S, x);;
gap> ForAll(GreensDClasses(S), x -> IsGreensLessThanOrEqual(D, x));
true
gap> MinimalIdeal(POI(10));
<partial perm group of rank 10>
gap> MinimalIdeal(BrauerMonoid(6));
<simple bipartition *-semigroup ideal of degree 6 with 1 generator>
```

13.7.2 RepresentativeOfMinimalIdeal

▷ `RepresentativeOfMinimalIdeal(S)` (attribute)

▷ `RepresentativeOfMinimalDClass(S)` (attribute)

Returns: An element of the minimal ideal of a semigroup.

The minimal ideal of a semigroup is the least ideal with respect to containment.

This method returns a representative element of the minimal ideal of S without having to create the minimal ideal itself. In general, beyond being a member of the minimal ideal, the returned element is not guaranteed to have any special properties. However, the element will coincide with the zero element of S if one exists.

This method works particularly well if S is a semigroup of transformations or partial permutations.

See also `MinimalIdeal` (13.7.1) and `MinimalDClass` (12.1.6).

Example
```
gap> S := SymmetricInverseSemigroup(10);;
gap> RepresentativeOfMinimalIdeal(S);
<empty partial perm>
gap> B := Semigroup([
> Bipartition([[1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
> Bipartition([[1, -1], [2], [3], [4, -3], [5, 6, -5, -6],
>   [-2, -4]])]);;
gap> RepresentativeOfMinimalIdeal(B);
<bipartition: [ 1, 2 ], [ 3, 6 ], [ 4, 5 ], [ -1, -5, -6 ],
 [ -2, -4 ], [ -3 ]>
gap> S := Semigroup(Transformation([5, 1, 6, 2, 2, 4]),
>                   Transformation([3, 5, 5, 1, 6, 2]));;
gap> RepresentativeOfMinimalDClass(S);
Transformation( [ 1, 2, 2, 5, 5, 1 ] )
gap> MinimalDClass(S);
<Green's D-class: Transformation( [ 1, 2, 2, 5, 5, 1 ] )>
```

13.7.3 MultiplicativeZero

▷ `MultiplicativeZero(S)` (attribute)

Returns: The zero element of a semigroup.

`MultiplicativeZero` returns the zero element of the semigroup S if it exists and `fail` if it does not. See also `MultiplicativeZero` (**Reference: MultiplicativeZero**).

Example
```
gap> S := Semigroup(Transformation([1, 4, 2, 6, 6, 5, 2]),
>                   Transformation([1, 6, 3, 6, 2, 1, 6]));;
gap> MultiplicativeZero(S);
Transformation( [ 1, 1, 1, 1, 1, 1, 1 ] )
gap> S := Semigroup(Transformation([2, 8, 3, 7, 1, 5, 2, 6]),
>                   Transformation([3, 5, 7, 2, 5, 6, 3, 8]),
>                   Transformation([6, 7, 4, 1, 4, 1, 6, 2]),
>                   Transformation([8, 8, 5, 1, 7, 5, 2, 8]));;
gap> MultiplicativeZero(S);
fail
gap> S := InverseSemigroup(
> PartialPerm([1, 3, 4], [5, 3, 1]),
> PartialPerm([1, 2, 3, 4], [4, 3, 1, 2]),
> PartialPerm([1, 3, 4, 5], [2, 4, 5, 3]));;
gap> MultiplicativeZero(S);
<empty partial perm>
gap> S := PartitionMonoid(6);
<regular bipartition *-monoid of size 4213597, degree 6 with 4
```

```
 generators>
gap> MultiplicativeZero(S);
fail
gap> S := DualSymmetricInverseMonoid(6);
<inverse block bijection monoid of degree 6 with 3 generators>
gap> MultiplicativeZero(S);
<block bijection: [ 1, 2, 3, 4, 5, 6, -1, -2, -3, -4, -5, -6 ]>
```

13.7.4 UnderlyingSemigroupOfSemigroupWithAdjoinedZero

▷ UnderlyingSemigroupOfSemigroupWithAdjoinedZero(*S*) (attribute)

Returns: A semigroup, or fail.

If *S* is a semigroup for which the property IsSemigroupWithAdjoinedZero (14.1.19) is true, (i.e. *S* has a MultiplicativeZero (13.7.3) and the set $S \setminus \{0\}$ is a subsemigroup of *S*), then this method returns the semigroup $S \setminus \{0\}$.

Otherwise, if *S* is a semigroup for which the property IsSemigroupWithAdjoinedZero (14.1.19) is false, then this method returns fail.

Example

```
gap> S := Semigroup([
> Transformation([2, 3, 4, 5, 1, 6]),
> Transformation([2, 1, 3, 4, 5, 6]),
> Transformation([6, 6, 6, 6, 6, 6])]);
<transformation semigroup of degree 6 with 3 generators>
gap> MultiplicativeZero(S);
Transformation( [ 6, 6, 6, 6, 6, 6 ] )
gap> G := UnderlyingSemigroupOfSemigroupWithAdjoinedZero(S);
<transformation semigroup of degree 5 with 2 generators>
gap> IsGroupAsSemigroup(G);
true
gap> IsZeroGroup(S);
true
gap> S := SymmetricInverseMonoid(6);;
gap> MultiplicativeZero(S);
<empty partial perm>
gap> G := UnderlyingSemigroupOfSemigroupWithAdjoinedZero(S);
fail
```

13.8 Group of units and identity elements

13.8.1 GroupOfUnits

▷ GroupOfUnits(*S*) (attribute)

Returns: The group of units of a semigroup or fail.

GroupOfUnits returns the group of units of the semigroup *S* as a subsemigroup of *S* if it exists and returns fail if it does not. Use IsomorphismPermGroup (6.5.5) if you require a permutation representation of the group of units.

If a semigroup *S* has an identity e, then the *group of units* of *S* is the set of those s in *S* such that there exists t in *S* where s*t=t*s=e. Equivalently, the group of units is the $\mathscr{H}$-class of the identity of *S*.

See also `GreensHClassOfElement` (**Reference: GreensHClassOfElement**), `IsMonoidAsSemigroup` (14.1.12), and `MultiplicativeNeutralElement` (**Reference: MultiplicativeNeutralElement**).

Example

```
gap> S := Semigroup(
> Transformation([1, 2, 5, 4, 3, 8, 7, 6]),
> Transformation([1, 6, 3, 4, 7, 2, 5, 8]),
> Transformation([2, 1, 6, 7, 8, 3, 4, 5]),
> Transformation([3, 2, 3, 6, 1, 6, 1, 2]),
> Transformation([5, 2, 3, 6, 3, 4, 7, 4]));;
gap> Size(S);
5304
gap> StructureDescription(GroupOfUnits(S));
"C2 x S4"
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
>             [2, 4, 5, 3, 6, 7, 10, 9, 8, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 7, 8, 10],
>             [8, 2, 3, 1, 4, 5, 10, 6, 9]));;
gap> StructureDescription(GroupOfUnits(S));
"C8"
gap> S := InverseSemigroup(
> PartialPerm([1, 3, 4], [4, 3, 5]),
> PartialPerm([1, 2, 3, 5], [3, 1, 5, 2]));;
gap> GroupOfUnits(S);
fail
gap> S := Semigroup(
> Bipartition([[1, 2, 3, -1, -3], [-2]]),
> Bipartition([[1, -1], [2, 3, -2, -3]]),
> Bipartition([[1, -2], [2, -3], [3, -1]]),
> Bipartition([[1], [2, 3, -2], [-1, -3]]));;
gap> StructureDescription(GroupOfUnits(S));
"C3"
```

13.9 Idempotents

13.9.1 Idempotents

▷ `Idempotents(`*obj*`[, `*n*`])` (attribute)

Returns: A list of idempotents.

The argument *obj* should be a semigroup, $\mathscr{D}$-class, $\mathscr{H}$-class, $\mathscr{L}$-class, or $\mathscr{R}$-class.

If the optional second argument *n* is present and *obj* is a semigroup, then a list of the idempotents in *obj* of rank *n* is returned. If you are only interested in the idempotents of a given rank, then the second version of the function will probably be faster. However, if the optional second argument is present, then nothing is stored in *obj* and so every time the function is called the computation must be repeated.

This functions produce essentially the same output as the GAP library function with the same name; see `Idempotents` (**Reference: Idempotents**). The main difference is that this function can be applied to a wider class of objects as described above.

See also IsRegularDClass (**Reference: IsRegularDClass**), IsRegularGreensClass (12.3.2) IsGroupHClass (**Reference: IsGroupHClass**), NrIdempotents (13.9.2), and GroupHClass (12.4.1).

Example

```
gap> S := Semigroup(Transformation([2, 3, 4, 1]),
>                   Transformation([3, 3, 1, 1]));;
gap> Idempotents(S, 1);
[  ]
gap> AsSet(Idempotents(S, 2));
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 3, 3, 1 ] ),
  Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> AsSet(Idempotents(S));
[ Transformation( [ 1, 1, 3, 3 ] ), IdentityTransformation,
  Transformation( [ 1, 3, 3, 1 ] ), Transformation( [ 2, 2, 4, 4 ] ),
  Transformation( [ 4, 2, 2, 4 ] ) ]
gap> x := Transformation([2, 2, 4, 4]);;
gap> R := GreensRClassOfElement(S, x);;
gap> Idempotents(R);
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 2, 2, 4, 4 ] ) ]
gap> x := Transformation([4, 2, 2, 4]);;
gap> L := GreensLClassOfElement(S, x);;
gap> AsSet(Idempotents(L));
[ Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> D := DClassOfLClass(L);;
gap> AsSet(Idempotents(D));
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 3, 3, 1 ] ),
  Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> L := GreensLClassOfElement(S, Transformation([3, 1, 1, 3]));;
gap> AsSet(Idempotents(L));
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 3, 3, 1 ] ) ]
gap> H := GroupHClass(D);
<Green's H-class: Transformation( [ 1, 1, 3, 3 ] )>
gap> Idempotents(H);
[ Transformation( [ 1, 1, 3, 3 ] ) ]
gap> S := InverseSemigroup(
> PartialPerm([10, 6, 3, 4, 9, 0, 1]),
> PartialPerm([6, 10, 7, 4, 8, 2, 9, 1]));;
gap> Idempotents(S, 1);
[ <identity partial perm on [ 4 ]> ]
gap> Idempotents(S, 0);
[  ]
```

13.9.2 NrIdempotents

▷ NrIdempotents(*obj*) (attribute)

Returns: A positive integer.

This function returns the number of idempotents in *obj* where *obj* can be a semigroup, $\mathscr{D}$-, $\mathscr{L}$-, $\mathscr{H}$-, or $\mathscr{R}$-class. If the actual idempotents are not required, then it is more efficient to use NrIdempotents(obj) than Length(Idempotents(obj)) since the idempotents themselves are not created when NrIdempotents is called.

See also Idempotents (**Reference: Idempotents**) and Idempotents (13.9.1), IsRegularDClass (**Reference: IsRegularDClass**), IsRegularGreensClass (12.3.2) IsGroupHClass (**Reference: IsGroupHClass**), and GroupHClass (12.4.1).

Example
```
gap> S := Semigroup(Transformation([2, 3, 4, 1]),
>                   Transformation([3, 3, 1, 1]));;
gap> NrIdempotents(S);
5
gap> f := Transformation([2, 2, 4, 4]);;
gap> R := GreensRClassOfElement(S, f);;
gap> NrIdempotents(R);
2
gap> f := Transformation([4, 2, 2, 4]);;
gap> L := GreensLClassOfElement(S, f);;
gap> NrIdempotents(L);
2
gap> D := DClassOfLClass(L);;
gap> NrIdempotents(D);
4
gap> L := GreensLClassOfElement(S, Transformation([3, 1, 1, 3]));;
gap> NrIdempotents(L);
2
gap> H := GroupHClass(D);;
gap> NrIdempotents(H);
1
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 5, 7, 9, 10],
>              [6, 7, 2, 9, 1, 5, 3]),
> PartialPerm([1, 2, 3, 5, 6, 7, 9, 10],
>             [8, 1, 9, 4, 10, 5, 6, 7]));;
gap> NrIdempotents(S);
236
gap> f := PartialPerm([2, 3, 7, 9, 10],
>                     [7, 2, 1, 5, 3]);;
gap> D := DClassNC(S, f);;
gap> NrIdempotents(D);
13
```

13.9.3 IdempotentGeneratedSubsemigroup

▷ IdempotentGeneratedSubsemigroup(*S*) (attribute)

Returns: A semigroup.

IdempotentGeneratedSubsemigroup returns the subsemigroup of the semigroup *S* generated by the idempotents of *S*.

See also Idempotents (13.9.1) and SmallGeneratingSet (13.6.2).

Example
```
gap> S := Semigroup(Transformation([1, 1]),
>                   Transformation([2, 1]),
>                   Transformation([1, 2, 2]),
>                   Transformation([1, 2, 3, 4, 5, 1]),
>                   Transformation([1, 2, 3, 4, 5, 5]),
>                   Transformation([1, 2, 3, 4, 6, 5]),
>                   Transformation([1, 2, 3, 5, 4]),
```

```
>                   Transformation([1, 2, 3, 7, 4, 5, 7]),
>                   Transformation([1, 2, 4, 8, 8, 3, 8, 7]),
>                   Transformation([1, 2, 8, 4, 5, 6, 7, 8]),
>                   Transformation([7, 7, 7, 4, 5, 6, 1]));;
gap> IdempotentGeneratedSubsemigroup(S) =
> Monoid(Transformation([1, 1]),
>        Transformation([1, 2, 1]),
>        Transformation([1, 2, 2]),
>        Transformation([1, 2, 3, 1]),
>        Transformation([1, 2, 3, 2]),
>        Transformation([1, 2, 3, 4, 1]),
>        Transformation([1, 2, 3, 4, 2]),
>        Transformation([1, 2, 3, 4, 4]),
>        Transformation([1, 2, 3, 4, 5, 1]),
>        Transformation([1, 2, 3, 4, 5, 2]),
>        Transformation([1, 2, 3, 4, 5, 5]),
>        Transformation([1, 2, 3, 4, 5, 7, 7]),
>        Transformation([1, 2, 3, 4, 7, 6, 7]),
>        Transformation([1, 2, 3, 6, 5, 6]),
>        Transformation([1, 2, 3, 7, 5, 6, 7]),
>        Transformation([1, 2, 8, 4, 5, 6, 7, 8]),
>        Transformation([2, 2]));
true
gap> S := SymmetricInverseSemigroup(5);
<symmetric inverse monoid of degree 5>
gap> IdempotentGeneratedSubsemigroup(S);
<inverse partial perm monoid of rank 5 with 5 generators>
gap> S := DualSymmetricInverseSemigroup(5);
<inverse block bijection monoid of degree 5 with 3 generators>
gap> IdempotentGeneratedSubsemigroup(S);
<inverse block bijection monoid of degree 5 with 10 generators>
gap> IsSemilattice(last);
true
```

13.10 Maximal subsemigroups

The Semigroups package provides methods to calculate the maximal subsemigroups of a finite semigroup, subject to various conditions. A *maximal subsemigroup* of a semigroup is a proper subsemigroup that is contained in no other proper subsemigroup of the semigroup.

When computing the maximal subsemigroups of a regular Rees (0-)matrix semigroup over a group, additional functionality is available. As described in [GGR68], a maximal subsemigroup of a finite regular Rees (0-)matrix semigroup over a group is one of 6 possible types. Using the Semigroups package, it is possible to search for only those maximal subsemigroups of certain types.

A maximal subsemigroup of such a Rees (0-)matrix semigroup `R` over a group `G` is either:

1. `{0}`;

2. formed by removing `0`;

3. formed by removing a column (a non-zero $\mathscr{L}$-class);

4. formed by removing a row (a non-zero $\mathscr{R}$-class);
5. formed by removing a set of both rows and columns;
6. isomorphic to a Rees (0-)matrix semigroup of the same dimensions over a maximal subgroup of `G` (in particular, the maximal subsemigroup intersects every $\mathscr{H}$-class of `R`).

Note that if `R` is a Rees matrix semigroup then it has no maximal subsemigroups of types 1, 2, or 5. Only types 3, 4, and 6 are relevant to a Rees matrix semigroup.

13.10.1 MaximalSubsemigroups (for a finite semigroup)

▷ `MaximalSubsemigroups(S)` (attribute)
▷ `MaximalSubsemigroups(S, opts)` (operation)

Returns: The maximal subsemigroups of `S`.

If `S` is a finite semigroup, then the attribute `MaximalSubsemigroups` returns a list of the non-empty maximal subsemigroups of `S`. The methods used by `MaximalSubsemigroups` are based on [GGR68], and are described in [DMW16].

It is computationally expensive to search for the maximal subsemigroups of a semigroup, and so computations involving `MaximalSubsemigroups` may be very lengthy. A substantial amount of information on the progress of `MaximalSubsemigroups` is provided through the info class `InfoSemigroups` (2.6.1), with increasingly detailed information given at levels 1, 2, and 3.

The behaviour of `MaximalSubsemigroups` can be altered via the second argument `opts`, which should be a record. The optional components of `opts` are:

gens (a boolean)

If `opts.gens` is `false` or unspecified, then the maximal subsemigroups themselves are returned and not just generating sets for these subsemigroups.

It can be more computationally expensive to return the generating sets for the maximal subsemigroups, than to return the maximal subsemigroups themselves.

contain (a list)

If `opts.contain` is duplicate-free list of elements of `S`, then `MaximalSubsemigroups` will search for the maximal subsemigroups of `S` which contain those elements.

D (a $\mathscr{D}$-class)

For a maximal subsemigroup `M` of a finite semigroup `S`, there exists a unique $\mathscr{D}$-class which contains the complement of `M` in `S`. In other words, the elements of `S` which `M` lacks are contained in a unique $\mathscr{D}$-class.

If `opts.D` is a $\mathscr{D}$-class of `S`, then `MaximalSubsemigroups` will search exclusively for those maximal subsemigroups of `S` whose complement is contained in `opts.D`.

types (a list)

This option is relevant only if S is a regular Rees (0-)matrix semigroup over a group.

As described at the start of this subsection, 13.10, a maximal subsemigroup of a regular Rees (0-)matrix semigroup over a group is one of 6 possible types.

If `S` is a regular Rees (0-)matrix semigroup over a group and `opts.types` is a subset of `[1 .. 6]`, then `MaximalSubsemigroups` will search for those maximal subsemigroups of `S` of the types enumerated by `opts.types`.

The default value for this option is `[1 .. 6]` (i.e. no restriction).

Example
```
gap> S := FullTransformationSemigroup(3);
<full transformation monoid of degree 3>
gap> MaximalSubsemigroups(S);
[ <transformation semigroup of degree 3 with 7 generators>,
  <transformation semigroup of degree 3 with 7 generators>,
  <transformation semigroup of degree 3 with 7 generators>,
  <transformation semigroup of degree 3 with 7 generators>,
  <transformation monoid of degree 3 with 5 generators> ]
gap> MaximalSubsemigroups(S,
> rec(gens := true, D := DClass(S, Transformation([2, 2, 3]))));
[ [ Transformation( [ 1, 1, 1 ] ), Transformation( [ 3, 3, 3 ] ),
      Transformation( [ 2, 2, 2 ] ), IdentityTransformation,
      Transformation( [ 2, 3, 1 ] ), Transformation( [ 2, 1 ] ) ] ]
gap> MaximalSubsemigroups(S,
> rec(contain := [Transformation([2, 3, 1])]));
[ <transformation semigroup of degree 3 with 7 generators>,
  <transformation monoid of degree 3 with 5 generators> ]
gap> R := PrincipalFactor(
> DClass(FullTransformationMonoid(4), Transformation([2, 2])));
<Rees 0-matrix semigroup 6x4 over Group([ (2,3,4), (2,4) ])>
gap> MaximalSubsemigroups(R, rec(types := [5],
>   contain := [RMSElement(R, 1, (), 1),
>               RMSElement(R, 1, (2, 3), 2)]));
[ <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators> ]
```

13.10.2 NrMaximalSubsemigroups

▷ `NrMaximalSubsemigroups(S)` (attribute)

Returns: The number of maximal subsemigroups of *S*.

If *S* is a finite semigroup, then `NrMaximalSubsemigroups` returns the number of non-empty maximal subsemigroups of *S*. The methods used by `MaximalSubsemigroups` are based on [GGR68], and are described in [DMW16].

It can be significantly faster to find the number of maximal subsemigroups of a semigroup than to find the maximal subsemigroups themselves.

Unless the maximal subsemigroups of *S* are already known, the command `NrMaximalSubsemigroups(S)` simply calls the command `MaximalSubsemigroups(S, rec(number := true))`.

For more information about searching for maximal subsemigroups of a finite semigroup in the Semigroups package, and for information about the options available to alter the search, see `MaximalSubsemigroups` (13.10.1). By supplying the additional option `opts.number := true`, the number of maximal subsemigroups will be returned rather than the subsemigroups themselves.

Example
```
gap> S := FullTransformationSemigroup(3);
<full transformation monoid of degree 3>
gap> NrMaximalSubsemigroups(S);
```

```
5
gap> S := RectangularBand(3, 4);;
gap> NrMaximalSubsemigroups(S);
7
gap> R := PrincipalFactor(
> DClass(FullTransformationMonoid(4), Transformation([2, 2])));
<Rees 0-matrix semigroup 6x4 over Group([ (2,3,4), (2,4) ])>
gap> MaximalSubsemigroups(R, rec(number := true, types := [3, 4]));
10
```

13.10.3 IsMaximalSubsemigroup

▷ `IsMaximalSubsemigroup(S, T)` (operation)

Returns: `true` or `false`.

If S and T are semigroups, then `IsMaximalSubsemigroup` returns `true` if and only if T is a maximal subsemigroup of S.

A *maximal subsemigroup* of S is a proper subsemigroup of S which is contained in no other proper subsemigroup of S.

Example

```
gap> S := ZeroSemigroup(2);;
gap> IsMaximalSubsemigroup(S, Semigroup(MultiplicativeZero(S)));
true
gap> S := FullTransformationSemigroup(4);
<full transformation monoid of degree 4>
gap> T := Semigroup(Transformation([3, 4, 1, 2]),
>                   Transformation([1, 4, 2, 3]),
>                   Transformation([2, 1, 1, 3]));
<transformation semigroup of degree 4 with 3 generators>
gap> IsMaximalSubsemigroup(S, T);
true
gap> R := Semigroup(Transformation([3, 4, 1, 2]),
>                   Transformation([1, 4, 2, 2]),
>                   Transformation([2, 1, 1, 3]));
<transformation semigroup of degree 4 with 3 generators>
gap> IsMaximalSubsemigroup(S, R);
false
```

13.11 The normalizer of a semigroup

13.11.1 Normalizer (for a perm group, semigroup, record)

▷ `Normalizer(G, S[, opts])` (operation)

▷ `Normalizer(S[, opts])` (operation)

Returns: A permutation group.

In its first form, this function returns the normalizer of the transformation, partial perm, or bipartition semigroup S in the permutation group G. In its second form, the normalizer of S in the symmetric group on `[1 .. n]` where `n` is the degree of S is returned.

The NORMALIZER of a transformation semigroup S in a permutation group G in the subgroup `H` of G consisting of those elements in g in G conjugating S to S, i.e. `S ^ g = S`.

Analogous definitions can be given for a partial perm and bipartition semigroups.

The method used by this operation is based on Section 3 in [ABMN10].

The optional final argument *opts* allows you to specify various options, which determine how the normalizer is calculated. The values of these options can dramatically change the time it takes for this operation to complete. In different situations, different options give the best performance.

The argument *opts* should be a record, and the available options are:

random

If this option has the value `true`, then the non-deterministic algorithms in genss are used in `Normalizer`. So, there is some chance that `Normalizer` will return an incorrect result in this case, but these methods can also be much faster than the deterministic algorithms which are used if this option is `false`.

The default value for this option is `false`.

lambdastab

If this option has the value `true`, then `Normalizer` initially finds the setwise stabilizer of the images or right blocks of the semigroup *S*. Sometimes this improves the performance of `Normalizer` and sometimes it does not. If this option in `false`, then this setwise stabilizer is not found.

The default value for this option is `true`.

rhostab

If this option has the value `true`, then `Normalizer` initially finds the setwise stabilizer of the kernels, domains, or left blocks of the semigroup *S*. Sometimes this improves the performance of `Normalizer` and sometimes it does not. If this option is `false`, the this setwise stabilizer is not found.

If *S* is an inverse semigroup, then this option is ignored.

The default value for this option is `true`.

Example

```
gap> S := BrauerMonoid(8);
<regular bipartition *-monoid of degree 8 with 3 generators>
gap> StructureDescription(Normalizer(S));
"S8"
gap> S := InverseSemigroup(PartialPerm([2, 5, 6, 3, 8]),
>                          PartialPerm([3, 6, 0, 2, 0, 0, 5, 7]));;
gap> Normalizer(S, rec(random := true, lambdastab := false));
#I  Have 33389 points.
#I  Have 40136 points in new orbit.
Group(())
```

13.12 Attributes of transformations and transformation semigroups

13.12.1 ComponentRepsOfTransformationSemigroup

▷ `ComponentRepsOfTransformationSemigroup(S)` (attribute)

Returns: The representatives of components of a transformation semigroup.

This function returns the representatives of the components of the action of the transformation semigroup *S* on the set of positive integers not greater than the degree of *S*.

The representatives are the least set of points such that every point can be reached from some representative under the action of S.

Example
```
gap> S := Semigroup(
> Transformation([11, 11, 9, 6, 4, 1, 4, 1, 6, 7, 12, 5]),
> Transformation([12, 10, 7, 10, 4, 1, 12, 9, 11, 9, 1, 12]));;
gap> ComponentRepsOfTransformationSemigroup(S);
[ 2, 3, 8 ]
```

13.12.2 ComponentsOfTransformationSemigroup

▷ `ComponentsOfTransformationSemigroup(S)` (attribute)

Returns: The components of a transformation semigroup.

This function returns the components of the action of the transformation semigroup S on the set of positive integers not greater than the degree of S; the components of S partition this set.

Example
```
gap> S := Semigroup(
> Transformation([11, 11, 9, 6, 4, 1, 4, 1, 6, 7, 12, 5]),
> Transformation([12, 10, 7, 10, 4, 1, 12, 9, 11, 9, 1, 12]));;
gap> ComponentsOfTransformationSemigroup(S);
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ] ]
```

13.12.3 CyclesOfTransformationSemigroup

▷ `CyclesOfTransformationSemigroup(S)` (attribute)

Returns: The cycles of a transformation semigroup.

This function returns the cycles, or strongly connected components, of the action of the transformation semigroup S on the set of positive integers not greater than the degree of S.

Example
```
gap> S := Semigroup(
> Transformation([11, 11, 9, 6, 4, 1, 4, 1, 6, 7, 12, 5]),
> Transformation([12, 10, 7, 10, 4, 1, 12, 9, 11, 9, 1, 12]));;
gap> CyclesOfTransformationSemigroup(S);
[ [ 12 ], [ 1, 11 ], [ 1, 11, 12, 5, 4, 6 ],
  [ 1, 11, 12, 5, 4, 10, 9, 6 ], [ 1, 12, 5, 4, 6 ],
  [ 1, 12, 5, 4, 10, 9, 6 ], [ 1, 12, 5, 4, 10, 9, 11 ],
  [ 11, 12, 5, 4, 10, 9 ], [ 12, 5, 4, 10, 7 ], [ 4, 10, 7 ] ]
```

13.12.4 DigraphOfActionOnPairs (for a transformation semigroup)

▷ `DigraphOfActionOnPairs(S)` (attribute)

▷ `DigraphOfActionOnPairs(S, n)` (attribute)

Returns: A digraph, or `fail`.

If S is a transformation semigroup and n is a non-negative integer such that S acts on the points `[1 .. n]`, then `DigraphOfActionOnPairs(S, n)` returns a digraph representing the `OnSets` (**Reference: OnSets**) action of S on the pairs of points in `[1 .. n]`.

If the optional argument n is not specified, then by default the degree of S will be chosen for n; see `DegreeOfTransformationSemigroup` (**Reference: DegreeOfTransformationSemigroup**). If the semigroup S does not act on `[1 .. n]`, then `DigraphOfActionOnPairs(S, n)` returns `fail`.

The digraph returned by `DigraphOfActionOnPairs` has $n + \binom{n}{2}$ vertices: the vertices `[1 .. n]` correspond to the points in `[1 .. n]`, and the remaining vertices correspond to the pairs of points in `[1 .. n]`. This correspondence is stored in the vertex labels of the digraph; see `DigraphVertexLabels` (**Digraphs: DigraphVertexLabels**).

The edges of the digraph are defined as follows. For each pair `{i, j}` in `[1 .. n]`, and for each generator `f` in `GeneratorsOfSemigroup(S)`, there is an edge from the vertex corresponding to `{i, j}` to the vertex corresponding to `{i ^ f, j ^ f}`. Since `f` is a transformation, the set `{i ^ f, j ^ f}` may correspond to a pair (in the case that `i ^ f <> j ^ f`), or to a point (in the case that `i ^ f = j ^ f`). The label of an edge is the position of the first transformation within `GeneratorsOfSemigroup(S)` that maps the pair corresponding to the source vertex to the pair/point corresponding to the range vertex. See `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**) and `DigraphEdgeLabels` (**Digraphs: DigraphEdgeLabels**) for further information.

Note that the digraph returned by `DigraphOfActionOnPairs` has no multiple edges; see `IsMultiDigraph` (**Digraphs: IsMultiDigraph**).

Example

```
gap> x := Transformation([2, 4, 3, 4, 7, 1, 6]);;
gap> y := Transformation([3, 3, 2, 3, 5, 1, 5]);;
gap> S := Semigroup(x, y);
<transformation semigroup of degree 7 with 2 generators>
gap> gr := DigraphOfActionOnPairs(S);
<digraph with 28 vertices, 41 edges>
gap> OnSets([2, 5], x);
[ 4, 7 ]
gap> DigraphVertexLabel(gr, 16);
[ 2, 5 ]
gap> DigraphVertexLabel(gr, 25);
[ 4, 7 ]
gap> DigraphEdgeLabel(gr, 16, 25);
1
gap> gr := DigraphOfActionOnPairs(S, 4);
<digraph with 10 vertices, 11 edges>
gap> DigraphVertexLabels(gr);
[ 1, 2, 3, 4, [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ],
  [ 3, 4 ] ]
gap> DigraphOfActionOnPairs(S, 5);
fail
```

13.12.5 DigraphOfActionOnPoints (for a transformation semigroup)

▷ `DigraphOfActionOnPoints(S)` (attribute)

▷ `DigraphOfActionOnPoints(S, n)` (attribute)

Returns: A digraph, or `fail`.

If S is a transformation semigroup and n is a non-negative integer such that S acts on the points `[1 .. n]`, then `DigraphOfActionOnPoints(S, n)` returns a digraph representing the `OnPoints` (**Reference: OnPoints**) action of S on the set `[1 .. n]`.

If the optional argument n is not specified, then by default the degree of S will be chosen for n; see `DegreeOfTransformationSemigroup` (**Reference: DegreeOfTransformationSemigroup**). If the semigroup S does not act on `[1 .. n]`, then `DigraphOfActionOnPairs(S, n)` returns `fail`.

The digraph returned by `DigraphOfActionOnPairs` has n vertices, where the vertex `i` corresponds to the point `i`. For each point `i` in `[1 .. n]`, and for each generator `f` in `GeneratorsOfSemigroup(S)`, there is an edge from the vertex `i` to the vertex `i ^ f`. See `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**) for further information.

Note that the digraph returned by `DigraphOfActionOnPoints` has no multiple edges; see `IsMultiDigraph` (**Digraphs: IsMultiDigraph**).

Example

```
gap> x := Transformation([2, 4, 2, 4, 7, 1, 6]);;
gap> y := Transformation([3, 3, 2, 3, 5, 1, 5]);;
gap> S := Semigroup(x, y);
<transformation semigroup of degree 7 with 2 generators>
gap> gr := DigraphOfActionOnPoints(S);
<digraph with 7 vertices, 12 edges>
gap> OnPoints(2, x);
4
gap> gr2 := DigraphOfActionOnPoints(S, 4);
<digraph with 4 vertices, 7 edges>
gap> gr2 = InducedSubdigraph(gr, [1 .. 4]);
true
gap> DigraphOfActionOnPoints(S, 5);
fail
```

13.12.6 FixedPointsOfTransformationSemigroup (for a transformation semigroup)

▷ `FixedPointsOfTransformationSemigroup(S)` (attribute)

Returns: A set of positive integers.

If *S* is a transformation semigroup, then `FixedPointsOfTransformationSemigroup(S)` returns the set of points `i` in `[1 .. DegreeOfTransformationSemigroup(S)]` such that `i ^ f = i` for all `f` in *S*.

Example

```
gap> f := Transformation([1, 4, 2, 4, 3, 7, 7]);
Transformation( [ 1, 4, 2, 4, 3, 7, 7 ] )
gap> S := Semigroup(f);
<commutative transformation semigroup of degree 7 with 1 generator>
gap> FixedPointsOfTransformationSemigroup(S);
[ 1, 4, 7 ]
```

13.12.7 IsTransitive (for a transformation semigroup and a set)

▷ `IsTransitive(S[, X])` (property)

▷ `IsTransitive(S[, n])` (property)

Returns: `true` or `false`.

A transformation semigroup *S* is *transitive* or *strongly connected* on the set *X* if for every `i, j` in *X* there is an element `s` in *S* such that `i ^ s = j`.

If the optional second argument is a positive integer *n*, then `IsTransitive` returns `true` if *S* is transitive on `[1 .. n]`, and `false` if it is not.

If the optional second argument is not provided, then the degree of *S* is used by default; see `DegreeOfTransformationSemigroup` (**Reference: DegreeOfTransformationSemigroup**).

Example

```
gap> S := Semigroup([
> Bipartition([
```

```
>    [1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
>   Bipartition([
>    [1, -4], [2, 3, 4, 5], [6], [-1, -6], [-2, -3], [-5]])]);
<bipartition semigroup of degree 6 with 2 generators>
gap> AsSemigroup(IsTransformationSemigroup, S);
<transformation semigroup of size 11, degree 12 with 2 generators>
gap> IsTransitive(last);
false
gap> IsTransitive(AsSemigroup(Group((1, 2, 3))));
true
```

13.12.8 SmallestElementSemigroup

▷ `SmallestElementSemigroup(S)` (attribute)
▷ `LargestElementSemigroup(S)` (attribute)

Returns: A transformation.

These attributes return the smallest and largest element of the transformation semigroup *S*, respectively. Smallest means the first element in the sorted set of elements of *S* and largest means the last element in the set of elements.

It is not necessary to find the elements of the semigroup to determine the smallest or largest element, and this function has considerable better performance than the equivalent `Elements(S)[1]` and `Elements(S)[Size(S)]`.

Example

```
gap> S := Monoid(
> Transformation([1, 4, 11, 11, 7, 2, 6, 2, 5, 5, 10]),
> Transformation([2, 4, 4, 2, 10, 5, 11, 11, 11, 6, 7]));
<transformation monoid of degree 11 with 2 generators>
gap> SmallestElementSemigroup(S);
IdentityTransformation
gap> LargestElementSemigroup(S);
Transformation( [ 11, 11, 10, 10, 7, 6, 5, 6, 2, 2, 4 ] )
```

13.12.9 CanonicalTransformation

▷ `CanonicalTransformation(trans[, n])` (function)

Returns: A transformation.

If *trans* is a transformation, and *n* is a non-negative integer such that the restriction of *trans* to `[1 .. n]` defines a transformation of `[1 .. n]`, then `CanonicalTransformation` returns a canonical representative of the transformation *trans* restricted to `[1 .. n]`.

More specifically, let `C(n)` be a class of transformations of degree *n* such that `AsDigraph` returns isomorphic digraphs for every pair of element elements in `C(n)`. Recall that for a transformation *trans* and integer *n* the function `AsDigraph` returns a digraph with *n* vertices and an edge with source `x` and range `x^trans` for every `x` in `[1 .. n]`. See `AsDigraph` (**Digraphs: AsDigraph**). Then `CanonicalTransformation` returns a canonical representative of the class `C(n)` that contains *trans*.

Example

```
gap> x := Transformation([5, 1, 4, 1, 1]);
Transformation( [ 5, 1, 4, 1, 1 ] )
gap> y := Transformation([3, 3, 2, 3, 1]);
```

```
Transformation( [ 3, 3, 2, 3, 1 ] )
gap> CanonicalTransformation(x);
Transformation( [ 5, 5, 1, 5, 4 ] )
gap> CanonicalTransformation(y);
Transformation( [ 5, 5, 1, 5, 4 ] )
```

13.12.10 IsConnectedTransformationSemigroup (for a transformation semigroup)

▷ `IsConnectedTransformationSemigroup(S)` (property)

Returns: `true` or `false`.

A transformation semigroup S is connected if the digraph returned by the function `DigraphOfActionOnPoints` is connected. See `IsConnectedDigraph` (**Digraphs: IsConnectedDigraph**) and `DigraphOfActionOnPoints` (13.12.5). The function `IsConnectedTransformationSemigroup` returns `true` if the semigroup S is connected and `false` otherwise.

Example
```
gap> S := Semigroup([
>  Transformation([2, 4, 3, 4]),
>  Transformation([3, 3, 2, 3, 3])]);
<transformation semigroup of degree 5 with 2 generators>
gap> IsConnectedTransformationSemigroup(S);
true
```

13.13 Attributes of partial perm semigroups

13.13.1 ComponentRepsOfPartialPermSemigroup

▷ `ComponentRepsOfPartialPermSemigroup(S)` (attribute)

Returns: The representatives of components of a partial perm semigroup.

This function returns the representatives of the components of the action of the partial perm semigroup S on the set of positive integers where it is defined.

The representatives are the least set of points such that every point can be reached from some representative under the action of S.

Example
```
gap> S := Semigroup([
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19],
>             [9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 16, 18, 19, 20],
>             [13, 1, 8, 5, 4, 14, 11, 12, 9, 20, 2, 18, 7, 3, 19])]);;
gap> ComponentRepsOfPartialPermSemigroup(S);
[ 1, 4, 6, 10, 15, 17 ]
```

13.13.2 ComponentsOfPartialPermSemigroup

▷ `ComponentsOfPartialPermSemigroup(S)` (attribute)

Returns: The components of a partial perm semigroup.

This function returns the components of the action of the partial perm semigroup S on the set of positive integers where it is defined; the components of S partition this set.

Example

```
gap> S := Semigroup([
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19],
>             [9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 16, 18, 19, 20],
>             [13, 1, 8, 5, 4, 14, 11, 12, 9, 20, 2, 18, 7, 3, 19])]);;
gap> ComponentsOfPartialPermSemigroup(S);
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20 ],
  [ 15 ], [ 17 ] ]
```

13.13.3 CyclesOfPartialPerm

▷ CyclesOfPartialPerm(*x*) (attribute)

Returns: The cycles of a partial perm.

This function returns the cycles, or strongly connected components, of the action of the partial perm *x* on the set of positive integers where it is defined.

Example

```
gap> x := PartialPerm([3, 1, 4, 2, 5, 0, 0, 6, 0, 7]);
[8,6][10,7](1,3,4,2)(5)
gap> CyclesOfPartialPerm(x);
[ [ 3, 4, 2, 1 ], [ 5 ] ]
```

13.13.4 CyclesOfPartialPermSemigroup

▷ CyclesOfPartialPermSemigroup(*S*) (attribute)

Returns: The cycles of a partial perm semigroup.

This function returns the cycles, or strongly connected components, of the action of the partial perm semigroup *S* on the set of positive integers where it is defined.

Example

```
gap> S := Semigroup([
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19],
>             [9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 16, 18, 19, 20],
>             [13, 1, 8, 5, 4, 14, 11, 12, 9, 20, 2, 18, 7, 3, 19])]);;
gap> CyclesOfPartialPermSemigroup(S);
[ [ 1, 9, 12, 14, 2, 20, 19, 3, 8, 11 ] ]
```

The content in this chapter is based partly on work by Zachary Mesyan. A full description of the objects described can be found in [MM16].

13.14 Attributes of Rees (0-)matrix semigroups

13.14.1 RZMSDigraph

▷ RZMSDigraph(*R*) (attribute)

Returns: A digraph.

If *R* is an n by m Rees 0-matrix semigroup $M^0[I,T,\Lambda;P]$ (so that $I = \{1,2,\dots,n\}$ and $\Lambda = \{1,2,\dots,m\}$) then RZMSDigraph returns a symmetric bipartite digraph with $n+m$ vertices. An index $i \in I$ corresponds to the vertex i and an index $j \in \Lambda$ corresponds to the vertex $j+n$.

Two vertices v and w in `RZMSDigraph(R)` are adjacent if and only if $v \in I$, $w - n \in \Lambda$, and `P[w - n][v]` $\neq 0$.

This digraph is commonly called the *Graham-Houghton graph* of `R`.

Example
```
gap> R := PrincipalFactor(
> DClass(FullTransformationMonoid(5),
>        Transformation([2, 4, 1, 5, 5])));
<Rees 0-matrix semigroup 10x5 over Group([ (1,2,3,4), (1,2) ])>
gap> gr := RZMSDigraph(R);
<digraph with 15 vertices, 40 edges>
gap> e := DigraphEdges(gr)[1];
[ 1, 11 ]
gap> Matrix(R)[e[2] - 10][e[1]] <> 0;
true
```

13.14.2 RZMSConnectedComponents

▷ `RZMSConnectedComponents(R)` (attribute)

Returns: The connected components of a Rees 0-matrix semigroup.

If `R` is an n by m Rees 0-matrix semigroup $M^0[I,T,\Lambda;P]$ (so that $I = \{1,2,\ldots,n\}$ and $\Lambda = \{1,2,\ldots,m\}$) then `RZMSConnectedComponents` returns the connected components of `R`.

Connectedness is an equivalence relation on the indices of `R`: the equivalence classes of the relation are called the *connected components* of `R`, and two indices in $I \cup \Lambda$ are connected if and only if their corresponding vertices in `RZMSDigraph(R)` are connected (see `RZMSDigraph` (13.14.1)). If `R` has n connected components, then `RZMSConnectedComponents` will return a list of pairs:

`[ [` I_1,Λ_1 `], ..., [` I_k,Λ_k `] ]`

where $I = I_1 \sqcup \cdots \sqcup I_k$, $\Lambda = \Lambda_1 \sqcup \cdots \sqcup \Lambda_k$, and for each l the set $I_l \cup \Lambda_l$ is a connected component of `R`. Note that at most one of I_l and Λ_l is possibly empty. The ordering of the connected components in the result in unspecified.

Example
```
gap> R := ReesZeroMatrixSemigroup(SymmetricGroup(5),
> [[(), 0, (1, 3), (4, 5), 0],
>  [0, (), 0, 0, (1, 3, 4, 5)],
>  [0, 0, (1, 5)(2, 3), 0, 0],
>  [0, (2, 3)(1, 4), 0, 0, 0]]);
<Rees 0-matrix semigroup 5x4 over Sym( [ 1 .. 5 ] )>
gap> RZMSConnectedComponents(R);
[ [ [ 1, 3, 4 ], [ 1, 3 ] ], [ [ 2, 5 ], [ 2, 4 ] ] ]
```

13.15 Changing the representation of a semigroup

13.15.1 IsomorphismReesMatrixSemigroup (for a semigroup)

▷ `IsomorphismReesMatrixSemigroup(S)` (attribute)
▷ `IsomorphismReesZeroMatrixSemigroup(S)` (attribute)
▷ `IsomorphismReesMatrixSemigroupOverPermGroup(S)` (attribute)
▷ `IsomorphismReesZeroMatrixSemigroupOverPermGroup(S)` (attribute)

Returns: An isomorphism.

If the semigroup S is finite and simple, then `IsomorphismReesMatrixSemigroup` returns an isomorphism to a Rees matrix semigroup over some group (usually a permutation group), and `IsomorphismReesMatrixSemigroupOverPermGroup` returns an isomorphism to a Rees matrix semigroup over a permutation group.

If S is finite and 0-simple, then `IsomorphismReesZeroMatrixSemigroup` returns an isomorphism to a Rees 0-matrix semigroup over some group (usually a permutation group), and `IsomorphismReesZeroMatrixSemigroupOverPermGroup` returns an isomorphism to a Rees 0-matrix semigroup over a permutation group.

See also `InjectionPrincipalFactor` (12.4.7).

Example

```
gap> S := Semigroup(PartialPerm([1]));
<trivial partial perm group of rank 1 with 1 generator>
gap> IsomorphismReesMatrixSemigroup(S);
MappingByFunction( <trivial partial perm group of rank 1 with
  1 generator>, <Rees matrix semigroup 1x1 over Group(())>
, function( x ) ... end, function( x ) ... end )
gap> S := Semigroup(PartialPerm([1]), PartialPerm([]));
<partial perm monoid of rank 1 with 2 generators>
gap> Range(IsomorphismReesZeroMatrixSemigroup(S));
<Rees 0-matrix semigroup 1x1 over Group(())>
```

Chapter 14

Properties of semigroups

In this chapter we decribe the methods that are available in Semigroups for determining various properties of a semigroup.

14.1 Properties of semigroups

In this section we describe the properties of a semigroup that can be determined using the Semigroups package.

14.1.1 IsBand

▷ IsBand(*S*) (property)

Returns: true or false.

IsBand returns true if every element of the semigroup *S* is an idempotent and false if it is not. An inverse semigroup is band if and only if it is a semilattice; see IsSemilattice (14.1.20).

Example

```
gap> S := Semigroup(
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 1]),
> Transformation([2, 2, 2, 5, 5, 5, 8, 8, 8, 2]),
> Transformation([3, 3, 3, 6, 6, 6, 9, 9, 9, 3]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 4]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 7]));;
gap> IsBand(S);
true
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 4, 8, 9], [5, 8, 7, 6, 9, 1]),
> PartialPerm([1, 3, 4, 7, 8, 9, 10], [2, 3, 8, 7, 10, 6, 1]));;
gap> IsBand(S);
false
gap> IsBand(IdempotentGeneratedSubsemigroup(S));
true
gap> S := PartitionMonoid(4);
<regular bipartition *-monoid of size 4140, degree 4 with 4
 generators>
gap> M := MinimalIdeal(S);
<simple bipartition *-semigroup ideal of degree 4 with 1 generator>
```

```
gap> IsBand(M);
true
```

14.1.2 IsBlockGroup

▷ `IsBlockGroup(S)` (property)

Returns: `true` or `false`.

`IsBlockGroup` returns `true` if the semigroup S is a block group and `false` if it is not.

A semigroup S is a *block group* if every $\mathscr{L}$-class and every $\mathscr{R}$-class of S contains at most one idempotent. Every semigroup of partial permutations is a block group.

Example

```
gap> S := Semigroup(Transformation([5, 6, 7, 3, 1, 4, 2, 8]),
>                   Transformation([3, 6, 8, 5, 7, 4, 2, 8]));;
gap> IsBlockGroup(S);
true
gap> S := Semigroup(
> Transformation([2, 1, 10, 4, 5, 9, 7, 4, 8, 4]),
> Transformation([10, 7, 5, 6, 1, 3, 9, 7, 10, 2]));;
gap> IsBlockGroup(S);
false
gap> S := Semigroup(PartialPerm([1, 2], [5, 4]),
>                   PartialPerm([1, 2, 3], [1, 2, 5]),
>                   PartialPerm([1, 2, 3], [2, 1, 5]),
>                   PartialPerm([1, 3, 4], [3, 1, 2]),
>                   PartialPerm([1, 3, 4, 5], [5, 4, 3, 2]));;
gap> T := AsSemigroup(IsBlockBijectionSemigroup, S);
<block bijection semigroup of degree 6 with 5 generators>
gap> IsBlockGroup(T);
true
gap> IsBlockGroup(AsSemigroup(IsBipartitionSemigroup, S));
true
gap> S := Semigroup(
> Bipartition([[1, -2], [2, -3], [3, -4], [4, -1]]),
> Bipartition([[1, -2], [2, -1], [3, -3], [4, -4]]),
> Bipartition([[1, 2, -3], [3, -1, -2], [4, -4]]),
> Bipartition([[1, -1], [2, -2], [3, -3], [4, -4]]));;
gap> IsBlockGroup(S);
true
```

14.1.3 IsCommutativeSemigroup

▷ `IsCommutativeSemigroup(S)` (property)

Returns: `true` or `false`.

`IsCommutativeSemigroup` returns `true` if the semigroup S is commutative and `false` if it is not. The function `IsCommutative` (**Reference: IsCommutative**) can also be used to test if a semigroup is commutative.

A semigroup S is *commutative* if `x * y = y * x` for all `x, y` in S.

Example

```
gap> S := Semigroup(Transformation([2, 4, 5, 3, 7, 8, 6, 9, 1]),
>                   Transformation([3, 5, 6, 7, 8, 1, 9, 2, 4]));;
```

```
gap> IsCommutativeSemigroup(S);
true
gap> IsCommutative(S);
true
gap> S := InverseSemigroup(
>  PartialPerm([1, 2, 3, 4, 5, 6], [2, 5, 1, 3, 9, 6]),
>  PartialPerm([1, 2, 3, 4, 6, 8], [8, 5, 7, 6, 2, 1]));;
gap> IsCommutativeSemigroup(S);
false
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 6, 7, -1, -4, -6],
>              [4, 5, 8, -2, -3, -5, -7, -8]]),
>  Bipartition([[1, 2, -3, -4], [3, -5], [4, -6], [5, -7],
>               [6, -8], [7, -1], [8, -2]]));;
gap> IsCommutativeSemigroup(S);
true
```

14.1.4 IsCompletelyRegularSemigroup

▷ IsCompletelyRegularSemigroup(*S*) (property)

Returns: true or false.

IsCompletelyRegularSemigroup returns true if every element of the semigroup *S* is contained in a subgroup of *S*.

An inverse semigroup is completely regular if and only if it is a Clifford semigroup; see IsCliffordSemigroup (15.2.1).

Example
```
gap> S := Semigroup(Transformation([1, 2, 4, 3, 6, 5, 4]),
>                   Transformation([1, 2, 5, 6, 3, 4, 5]),
>                   Transformation([2, 1, 2, 2, 2, 2, 2]));;
gap> IsCompletelyRegularSemigroup(S);
true
gap> IsInverseSemigroup(S);
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsCompletelyRegularSemigroup(T);
true
gap> IsCliffordSemigroup(T);
true
gap> S := Semigroup(
> Bipartition([[1, 3, -4], [2, 4, -1, -2], [-3]]),
> Bipartition([[1, -1], [2, 3, 4, -3], [-2, -4]]));;
gap> IsCompletelyRegularSemigroup(S);
false
```

14.1.5 IsCongruenceFreeSemigroup

▷ IsCongruenceFreeSemigroup(*S*) (property)

Returns: true or false.

IsCongruenceFreeSemigroup returns true if the semigroup *S* is a congruence-free semigroup and false if it is not.

A semigroup S is *congruence-free* if it has no non-trivial proper congruences.

A semigroup with zero is congruence-free if and only if it is isomorphic to a regular Rees 0-matrix semigroup `R` whose underlying semigroup is the trivial group, no two rows of the matrix of `R` are identical, and no two columns are identical; see Theorem 3.7.1 in [How95].

A semigroup without zero is congruence-free if and only if it is a simple group or has order 2; see Theorem 3.7.2 in [How95].

Example
```
gap> S := Semigroup(Transformation([4, 2, 3, 3, 4]));;
gap> IsCongruenceFreeSemigroup(S);
true
gap> S := Semigroup(Transformation([2, 2, 4, 4]),
>                   Transformation([5, 3, 4, 4, 6, 6]));;
gap> IsCongruenceFreeSemigroup(S);
false
```

14.1.6 IsGroupAsSemigroup

▷ `IsGroupAsSemigroup(S)` (property)

Returns: `true` or `false`.

`IsGroupAsSemigroup` returns `true` if and only if the semigroup S is mathematically a group.

Example
```
gap> S := Semigroup(Transformation([2, 4, 5, 3, 7, 8, 6, 9, 1]),
>                   Transformation([3, 5, 6, 7, 8, 1, 9, 2, 4]));;
gap> IsGroupAsSemigroup(S);
true
gap> G := SymmetricGroup(5);;
gap> IsGroupAsSemigroup(G);
true
gap> S := AsSemigroup(IsPartialPermSemigroup, G);
<partial perm group of size 120, rank 5 with 2 generators>
gap> IsGroupAsSemigroup(S);
true
gap> G := SymmetricGroup([1, 2, 10]);;
gap> T := AsSemigroup(IsBlockBijectionSemigroup, G);
<inverse block bijection semigroup of size 6, degree 11 with 2
 generators>
gap> IsGroupAsSemigroup(T);
true
```

14.1.7 IsIdempotentGenerated

▷ `IsIdempotentGenerated(S)` (property)

▷ `IsSemiband(S)` (property)

Returns: `true` or `false`.

`IsIdempotentGenerated` and `IsSemiband` return `true` if the semigroup S is generated by its idempotents and `false` if it is not. See also `Idempotents` (13.9.1) and `IdempotentGeneratedSubsemigroup` (13.9.3).

An inverse semigroup is idempotent-generated if and only if it is a semilattice; see `IsSemilattice` (14.1.20).

The terms semiband and idempotent-generated are synonymous in this context.

Example

```
gap> S := SingularTransformationSemigroup(4);
<regular transformation semigroup ideal of degree 4 with 1 generator>
gap> IsIdempotentGenerated(S);
true
gap> S := SingularBrauerMonoid(5);
<regular bipartition *-semigroup ideal of degree 5 with 1 generator>
gap> IsIdempotentGenerated(S);
true
```

14.1.8 IsLeftSimple

▷ IsLeftSimple(*S*) (property)

▷ IsRightSimple(*S*) (property)

Returns: true or false.

IsLeftSimple and IsRightSimple returns true if the semigroup *S* has only one $\mathscr{L}$-class or one $\mathscr{R}$-class, respectively, and returns false if it has more than one.

An inverse semigroup is left simple if and only if it is right simple if and only if it is a group; see IsGroupAsSemigroup (14.1.6).

Example

```
gap> S := Semigroup(Transformation([6, 7, 9, 6, 8, 9, 8, 7, 6]),
>                   Transformation([6, 8, 9, 6, 8, 8, 7, 9, 6]),
>                   Transformation([6, 8, 9, 7, 8, 8, 7, 9, 6]),
>                   Transformation([6, 9, 8, 6, 7, 9, 7, 8, 6]),
>                   Transformation([6, 9, 9, 6, 8, 8, 7, 9, 6]),
>                   Transformation([6, 9, 9, 7, 8, 8, 6, 9, 7]),
>                   Transformation([7, 8, 8, 7, 9, 9, 7, 8, 6]),
>                   Transformation([7, 9, 9, 7, 6, 9, 6, 8, 7]),
>                   Transformation([8, 7, 6, 9, 8, 6, 8, 7, 9]),
>                   Transformation([9, 6, 6, 7, 8, 8, 7, 6, 9]),
>                   Transformation([9, 6, 6, 7, 9, 6, 9, 8, 7]),
>                   Transformation([9, 6, 7, 9, 6, 6, 9, 7, 8]),
>                   Transformation([9, 6, 8, 7, 9, 6, 9, 8, 7]),
>                   Transformation([9, 7, 6, 8, 7, 7, 9, 6, 8]),
>                   Transformation([9, 7, 7, 8, 9, 6, 9, 7, 8]),
>                   Transformation([9, 8, 8, 9, 6, 7, 6, 8, 9]));;
gap> IsRightSimple(S);
false
gap> IsLeftSimple(S);
true
gap> IsGroupAsSemigroup(S);
false
gap> NrRClasses(S);
16
gap> S := BrauerMonoid(6);;
gap> S := Semigroup(RClass(S, Random(MinimalDClass(S))));;
gap> IsLeftSimple(S);
false
gap> IsRightSimple(S);
true
```

14.1.9 IsLeftZeroSemigroup

▷ `IsLeftZeroSemigroup(S)` (property)

Returns: `true` or `false`.

`IsLeftZeroSemigroup` returns `true` if the semigroup S is a left zero semigroup and `false` if it is not.

A semigroup is a *left zero semigroup* if `x*y=x` for all `x,y`. An inverse semigroup is a left zero semigroup if and only if it is trivial.

Example
```
gap> S := Semigroup(Transformation([2, 1, 4, 3, 5]),
>                   Transformation([3, 2, 3, 1, 1]));;
gap> IsRightZeroSemigroup(S);
false
gap> S := Semigroup(Transformation([1, 2, 3, 3, 1]),
>                   Transformation([1, 2, 3, 3, 3]));;
gap> IsLeftZeroSemigroup(S);
true
```

14.1.10 IsMonogenicSemigroup

▷ `IsMonogenicSemigroup(S)` (property)

Returns: `true` or `false`.

`IsMonogenicSemigroup` returns `true` if the semigroup S is monogenic and it returns `false` if it is not.

A semigroup is *monogenic* if it is generated by a single element. See also `IsMonogenicMonoid` (14.1.11), `IsMonogenicInverseSemigroup` (15.2.7), and `IsMonogenicInverseMonoid` (15.2.8).

Example
```
gap> S := Semigroup(
> Transformation(
>  [2, 2, 2, 11, 10, 8, 10, 11, 2, 11, 10, 2, 11, 11, 10]),
> Transformation(
>  [2, 2, 2, 8, 11, 15, 11, 10, 2, 10, 11, 2, 10, 4, 7]),
> Transformation(
>  [2, 2, 2, 11, 10, 8, 10, 11, 2, 11, 10, 2, 11, 11, 10]),
> Transformation(
>  [2, 2, 12, 7, 8, 14, 8, 11, 2, 11, 10, 2, 11, 15, 4]));;
gap> IsMonogenicSemigroup(S);
true
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -2, -5, -7, -9],
>              [-1, -10], [-3, -4, -6, -8]]),
>  Bipartition([[1, 4, 7, 8, -2], [2, 3, 5, 10, -5],
>               [6, 9, -7, -9], [-1, -10], [-3, -4, -6, -8]]));;
gap> IsMonogenicSemigroup(S);
true
gap> S := FullTransformationSemigroup(5);;
gap> IsMonogenicSemigroup(S);
false
```

14.1.11 IsMonogenicMonoid

▷ `IsMonogenicMonoid(S)` (property)

Returns: `true` or `false`.

`IsMonogenicMonoid` returns `true` if the monoid S is a monogenic monoid and it returns `false` if it is not.

A monoid is *monogenic* if it is generated as a monoid by a single element. See also `IsMonogenicSemigroup` (14.1.10) and `IsMonogenicInverseMonoid` (15.2.8).

Example
```
gap> x := PartialPerm([1, 2, 3, 6, 8, 10], [2, 6, 7, 9, 1, 5]);;
gap> S := Monoid(x, x ^ 2, x ^ 3);;
gap> IsMonogenicSemigroup(S);
false
gap> IsMonogenicMonoid(S);
true
gap> S := FullTransformationMonoid(5);;
gap> IsMonogenicMonoid(S);
false
```

14.1.12 IsMonoidAsSemigroup

▷ `IsMonoidAsSemigroup(S)` (property)

Returns: `true` or `false`.

`IsMonoidAsSemigroup` returns `true` if and only if the semigroup S is mathematically a monoid, i.e. if and only if it contains a `MultiplicativeNeutralElement` (**Reference: MultiplicativeNeutralElement**).

It is possible that a semigroup which satisfies `IsMonoidAsSemigroup` is not in the GAP category `IsMonoid` (**Reference: IsMonoid**). This is possible if the `MultiplicativeNeutralElement` (**Reference: MultiplicativeNeutralElement**) of S is not equal to the `One` (**Reference: One**) of any element in S. Therefore a semigroup satisfying `IsMonoidAsSemigroup` may not possess the attributes of a monoid (such as, `GeneratorsOfMonoid` (**Reference: GeneratorsOfMonoid**)).

See also `One` (**Reference: One**), `IsInverseMonoid` (**Reference: IsInverseMonoid**) and `IsomorphismTransformationMonoid` (**Reference: IsomorphismTransformationMonoid**).

Example
```
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> IsMonoidAsSemigroup(S);
true
gap> IsMonoid(S);
false
gap> MultiplicativeNeutralElement(S);
Transformation( [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 9 ] )
gap> T := AsSemigroup(IsBipartitionSemigroup, S);;
gap> IsMonoidAsSemigroup(T);
true
gap> IsMonoid(T);
false
gap> One(T);
fail
gap> S := Monoid(Transformation([8, 2, 8, 9, 10, 6, 2, 8, 7, 8]),
```

```
>               Transformation([9, 2, 6, 3, 6, 4, 5, 5, 3, 2]));;
gap> IsMonoidAsSemigroup(S);
true
```

14.1.13 IsOrthodoxSemigroup

▷ IsOrthodoxSemigroup(S) (property)

Returns: true or false.

IsOrthodoxSemigroup returns true if the semigroup S is orthodox and false if it is not.

A semigroup is *orthodox* if it is regular and its idempotent elements form a subsemigroup. Every inverse semigroup is also an orthodox semigroup.

See also IsRegularSemigroup (14.1.16) and IsRegularSemigroup (**Reference: IsRegularSemigroup**).

Example
```
gap> S := Semigroup(Transformation([1, 1, 1, 4, 5, 4]),
>                   Transformation([1, 2, 3, 1, 1, 2]),
>                   Transformation([1, 2, 3, 1, 1, 3]),
>                   Transformation([5, 5, 5, 5, 5, 5]));;
gap> IsOrthodoxSemigroup(S);
true
gap> S := DualSymmetricInverseMonoid(5);;
gap> S := Semigroup(GeneratorsOfSemigroup(S));;
gap> IsOrthodoxSemigroup(S);
true
```

14.1.14 IsRectangularBand

▷ IsRectangularBand(S) (property)

Returns: true or false.

IsRectangularBand returns true if the semigroup S is a rectangular band and false if it is not.

A semigroup S is a *rectangular band* if for all x, y, z in S we have that $x^2 = x$ and $xyz = xz$.

Equivalently, S is a *rectangular band* if S is isomorphic to a semigroup of the form $I \times \Lambda$ with multiplication $(i, \lambda)(j, \mu) = (i, \mu)$. In this case, S is called an $|I| \times |\Lambda|$ *rectangular band*.

An inverse semigroup is a rectangular band if and only if it is a group.

Example
```
gap> S := Semigroup(
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 1]),
> Transformation([2, 2, 2, 5, 5, 5, 8, 8, 8, 2]),
> Transformation([3, 3, 3, 6, 6, 6, 9, 9, 9, 3]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 4]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 7]));;
gap> IsRectangularBand(S);
true
gap> IsRectangularBand(MinimalIdeal(PartitionMonoid(4)));
true
```

14.1.15 IsRectangularGroup

▷ IsRectangularGroup(S) (property)

Returns: true or false.

A semigroup is *rectangular group* if it is the direct product of a group and a rectangular band. Or equivalently, if it is orthodox and simple.

Example
```
gap> G := AsSemigroup(IsTransformationSemigroup, MathieuGroup(11));
<transformation group of size 7920, degree 11 with 2 generators>
gap> R := RectangularBand(3, 2);
<regular transformation semigroup of size 6, degree 6 with 3
 generators>
gap> S := DirectProduct(G, R);
<transformation semigroup of size 47520, degree 17 with 5 generators>
gap> IsRectangularGroup(R);
true
gap> IsRectangularGroup(G);
true
gap> IsRectangularGroup(S);
true
gap> IsRectangularGroup(JonesMonoid(3));
false
```

14.1.16 IsRegularSemigroup

▷ `IsRegularSemigroup(S)` (property)

Returns: `true` or `false`.

`IsRegularSemigroup` returns `true` if the semigroup *S* is regular and `false` if it is not.

A semigroup `S` is *regular* if for all `x` in `S` there exists `y` in `S` such that `x * y * x = x`. Every inverse semigroup is regular, and a semigroup of partial permutations is regular if and only if it is an inverse semigroup.

See also `IsRegularDClass` (**Reference: IsRegularDClass**), `IsRegularGreensClass` (12.3.2), and `IsRegularSemigroupElement` (**Reference: IsRegularSemigroupElement**).

Example
```
gap> IsRegularSemigroup(FullTransformationSemigroup(5));
true
gap> IsRegularSemigroup(JonesMonoid(5));
true
```

14.1.17 IsRightZeroSemigroup

▷ `IsRightZeroSemigroup(S)` (property)

Returns: `true` or `false`.

`IsRightZeroSemigroup` returns `true` if the *S* is a right zero semigroup and `false` if it is not.

A semigroup `S` is a *right zero semigroup* if `x * y = y` for all `x, y` in `S`. An inverse semigroup is a right zero semigroup if and only if it is trivial.

Example
```
gap> S := Semigroup(Transformation([2, 1, 4, 3, 5]),
>                   Transformation([3, 2, 3, 1, 1]));;
gap> IsRightZeroSemigroup(S);
false
gap> S := Semigroup(Transformation([1, 2, 3, 3, 1]),
>                   Transformation([1, 2, 4, 4, 1]));;
```

```
gap> IsRightZeroSemigroup(S);
true
```

14.1.18 IsXTrivial

▷ IsRTrivial(*S*) (property)
▷ IsLTrivial(*S*) (property)
▷ IsHTrivial(*S*) (property)
▷ IsDTrivial(*S*) (property)
▷ IsAperiodicSemigroup(*S*) (property)
▷ IsCombinatorialSemigroup(*S*) (property)

Returns: true or false.

IsXTrivial returns true if Green's $\mathscr{R}$-relation, $\mathscr{L}$-relation, $\mathscr{H}$-relation, $\mathscr{D}$-relation, respectively, on the semigroup S is trivial and false if it is not. These properties can also be applied to a Green's class instead of a semigroup where applicable.

For inverse semigroups, the properties of being $\mathscr{R}$-trivial, $\mathscr{L}$-trivial, $\mathscr{D}$-trivial, and a semilattice are equivalent; see IsSemilattice (14.1.20).

A semigroup is *aperiodic* if its contains no non-trivial subgroups (equivalently, all of its group $\mathscr{H}$-classes are trivial). A finite semigroup is aperiodic if and only if it is $\mathscr{H}$-trivial.

Combinatorial is a synonym for aperiodic in this context.

Example

```
gap> S := Semigroup(
>  Transformation([1, 5, 1, 3, 7, 10, 6, 2, 7, 10]),
>  Transformation([4, 4, 5, 6, 7, 7, 7, 4, 3, 10]));;
gap> IsHTrivial(S);
true
gap> Size(S);
108
gap> IsRTrivial(S);
false
gap> IsLTrivial(S);
false
```

14.1.19 IsSemigroupWithAdjoinedZero

▷ IsSemigroupWithAdjoinedZero(*S*) (property)

Returns: true or false.

IsSemigroupWithAdjoinedZero returns true if the semigroup S can be expressed as the disjoint union of subsemigroups $S \setminus \{0\}$ and $\{0\}$ (where 0 is the MultiplicativeZero (13.7.3) of S).

If this is not the case, then either S lacks a multiplicative zero, or the set $S \setminus \{0\}$ is not a subsemigroup of S, and so IsSemigroupWithAdjoinedZero returns false.

Example

```
gap> S := Semigroup(Transformation([2, 3, 4, 5, 1, 6]),
>                   Transformation([2, 1, 3, 4, 5, 6]),
>                   Transformation([6, 6, 6, 6, 6, 6]));
<transformation semigroup of degree 6 with 3 generators>
gap> IsZeroGroup(S);
true
gap> IsSemigroupWithAdjoinedZero(S);
```

```
true
gap> S := FullTransformationMonoid(4);;
gap> IsSemigroupWithAdjoinedZero(S);
false
```

14.1.20 IsSemilattice

▷ IsSemilattice(*S*) (property)

Returns: true or false.

IsSemilattice returns true if the semigroup *S* is a semilattice and false if it is not.

A semigroup is a *semilattice* if it is commutative and every element is an idempotent. The idempotents of an inverse semigroup form a semilattice.

Example

```
gap> S := Semigroup(Transformation([2, 5, 1, 7, 3, 7, 7]),
>                   Transformation([3, 6, 5, 7, 2, 1, 7]));;
gap> Size(S);
631
gap> IsInverseSemigroup(S);
true
gap> A := Semigroup(Idempotents(S));
<transformation semigroup of degree 7 with 32 generators>
gap> IsSemilattice(A);
true
gap> S := FactorisableDualSymmetricInverseMonoid(5);;
gap> S := IdempotentGeneratedSubsemigroup(S);;
gap> IsSemilattice(S);
true
```

14.1.21 IsSimpleSemigroup

▷ IsSimpleSemigroup(*S*) (property)

▷ IsCompletelySimpleSemigroup(*S*) (property)

Returns: true or false.

IsSimpleSemigroup returns true if the semigroup *S* is simple and false if it is not.

A semigroup is *simple* if it has no proper 2-sided ideals. A semigroup is *completely simple* if it is simple and possesses minimal left and right ideals. A finite semigroup is simple if and only if it is completely simple. An inverse semigroup is simple if and only if it is a group.

Example

```
gap> S := Semigroup(
> Transformation([2, 2, 4, 4, 6, 6, 8, 8, 10, 10, 12, 12, 2]),
> Transformation([1, 1, 3, 3, 5, 5, 7, 7, 9, 9, 11, 11, 3]),
> Transformation([1, 7, 3, 9, 5, 11, 7, 1, 9, 3, 11, 5, 5]),
> Transformation([7, 7, 9, 9, 11, 11, 1, 1, 3, 3, 5, 5, 7]));;
gap> IsSimpleSemigroup(S);
true
gap> IsCompletelySimpleSemigroup(S);
true
gap> IsSimpleSemigroup(MinimalIdeal(BrauerMonoid(6)));
true
gap> R := Range(IsomorphismReesMatrixSemigroup(
```

```
> MinimalIdeal(BrauerMonoid(6))));
<Rees matrix semigroup 15x15 over Group(())>
```

14.1.22 IsSynchronizingSemigroup (for a transformation semigroup)

▷ IsSynchronizingSemigroup(*S*) (property)
▷ IsSynchronizingSemigroup(*S*, *n*) (property)

Returns: true or false.

For a positive integer *n*, IsSynchronizingSemigroup returns true if the semigroup of transformations *S* contains a transformation with constant value on [1 .. *n*]. Note that this function will return true whenever *n* = 1. See also ConstantTransformation (**Reference: ConstantTransformation**).

If the optional second argument is not specified, then *n* will be taken to be the value of DegreeOfTransformationSemigroup (**Reference: DegreeOfTransformationSemigroup**) for *S*.

Note that the semigroup consisting of the identity transformation is the unique transformation semigroup with degree 0. In this special case, the function IsSynchronizingSemigroup will return false.

Example

```
gap> S := Semigroup(
> Transformation([1, 1, 8, 7, 6, 6, 4, 1, 8, 9]),
> Transformation([5, 8, 7, 6, 10, 8, 7, 6, 9, 7]));;
gap> IsSynchronizingSemigroup(S, 10);
true
gap> S := Semigroup(
> Transformation([3, 8, 1, 1, 9, 9, 8, 7, 9, 6]),
> Transformation([7, 6, 8, 7, 5, 6, 8, 7, 8, 9]));;
gap> IsSynchronizingSemigroup(S, 10);
false
gap> Representative(MinimalIdeal(S));
Transformation( [ 7, 8, 8, 7, 8, 8, 8, 7, 8, 8 ] )
```

14.1.23 IsUnitRegularMonoid

▷ IsUnitRegularMonoid(*S*) (property)

Returns: true if the semigroup *S* is unit regular and false if it is not.

A monoid is *unit regular* if and only if for every >x in *S* there exists an element y in the group of units of *S* such that x*y*x=x.

Example

```
gap> IsUnitRegularMonoid(FullTransformationMonoid(3));
true
```

14.1.24 IsZeroGroup

▷ IsZeroGroup(*S*) (property)

Returns: true or false.

IsZeroGroup returns true if the semigroup *S* is a zero group and false if it is not.

A semigroup S is a *zero group* if there exists an element z in S such that S without z is a group and x*z=z*x=z for all x in S. Every zero group is an inverse semigroup.

Example
```
gap> S := Semigroup(Transformation([2, 2, 3, 4, 6, 8, 5, 5, 9]),
>                   Transformation([3, 3, 8, 2, 5, 6, 4, 4, 9]),
>                   ConstantTransformation(9, 9));;
gap> IsZeroGroup(S);
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsZeroGroup(T);
true
gap> IsZeroGroup(JonesMonoid(2));
true
```

14.1.25 IsZeroRectangularBand

▷ `IsZeroRectangularBand(S)` (property)

Returns: `true` or `false`.

`IsZeroRectangularBand` returns `true` if the semigroup S is a zero rectangular band and `false` if it is not.

A semigroup is a *0-rectangular band* if it is 0-simple and $\mathscr{H}$-trivial; see also `IsZeroSimpleSemigroup` (14.1.27) and `IsHTrivial` (14.1.18). An inverse semigroup is a 0-rectangular band if and only if it is a 0-group; see `IsZeroGroup` (14.1.24).

Example
```
gap> S := Semigroup(
>  Transformation([1, 3, 7, 9, 1, 12, 13, 1, 15, 9, 1, 18, 1, 1, 13,
>                  1, 1, 21, 1, 1, 1, 1, 1, 25, 26, 1]),
> Transformation([1, 5, 1, 5, 11, 1, 1, 14, 1, 16, 17, 1, 1, 19, 1,
>                 11, 1, 1, 1, 23, 1, 16, 19, 1, 1, 1]),
> Transformation([1, 4, 8, 1, 10, 1, 8, 1, 1, 1, 10, 1, 8, 10, 1, 1,
>                 20, 1, 22, 1, 8, 1, 1, 1, 1, 1]),
> Transformation([1, 6, 6, 1, 1, 1, 6, 1, 1, 1, 1, 1, 6, 1, 6, 1, 1,
>                 6, 1, 1, 24, 1, 1, 1, 1, 6]));;
gap> D := DClass(S,
> Transformation([1, 8, 1, 1, 8, 1, 1, 1, 1, 1, 8, 1, 1, 8, 1, 1, 1,
>                 1, 1, 1, 1, 1, 1, 1, 1, 1]));;
gap> IsZeroRectangularBand(Semigroup(D));
true
gap> IsZeroRectangularBand(Semigroup(GreensDClasses(S)[1]));
false
```

14.1.26 IsZeroSemigroup

▷ `IsZeroSemigroup(S)` (property)

Returns: `true` or `false`.

`IsZeroSemigroup` returns `true` if the semigroup S is a zero semigroup and `false` if it is not.

A semigroup `S` is a *zero semigroup* if there exists an element `z` in `S` such that `x*y=z` for all `x,y` in `S`. An inverse semigroup is a zero semigroup if and only if it is trivial.

Example
```
gap> S := Semigroup(
> Transformation([4, 7, 6, 3, 1, 5, 3, 6, 5, 9]),
> Transformation([5, 3, 5, 1, 9, 3, 8, 7, 4, 3]));;
```

```
gap> IsZeroSemigroup(S);
false
gap> S := Semigroup(
>  Transformation([7, 8, 8, 8, 5, 8, 8, 8]),
>  Transformation([8, 8, 8, 8, 5, 7, 8, 8]),
>  Transformation([8, 7, 8, 8, 5, 8, 8, 8]),
>  Transformation([8, 8, 8, 7, 5, 8, 8, 8]),
>  Transformation([8, 8, 7, 8, 5, 8, 8, 8]));;
gap> IsZeroSemigroup(S);
true
gap> MultiplicativeZero(S);
Transformation( [ 8, 8, 8, 8, 5, 8, 8, 8 ] )
```

14.1.27 IsZeroSimpleSemigroup

▷ `IsZeroSimpleSemigroup(S)` (property)

Returns: `true` or `false`.

`IsZeroSimpleSemigroup` returns `true` if the semigroup *S* is 0-simple and `false` if it is not.

A semigroup is a *0-simple* if it has no two-sided ideals other than itself and the set containing the zero element; see also `MultiplicativeZero` (13.7.3). An inverse semigroup is 0-simple if and only if it is a Brandt semigroup; see `IsBrandtSemigroup` (15.2.2).

Example
```
gap> S := Semigroup(
>  Transformation([1, 17, 17, 17, 17, 17, 17, 17, 17, 17, 5, 17,
>                  17, 17, 17, 17, 17]),
>  Transformation([1, 17, 17, 17, 11, 17, 17, 17, 17, 17, 17, 17,
>                  17, 17, 17, 17, 17]),
>  Transformation([1, 17, 17, 17, 17, 17, 17, 17, 17, 17, 4, 17,
>                  17, 17, 17, 17, 17]),
>  Transformation([1, 17, 17, 5, 17, 17, 17, 17, 17, 17, 17, 17,
>                  17, 17, 17, 17, 17]));;
gap> IsZeroSimpleSemigroup(S);
true
gap> S := Semigroup(
> Transformation([2, 3, 4, 5, 1, 8, 7, 6, 2, 7]),
> Transformation([2, 3, 4, 5, 6, 8, 7, 1, 2, 2]));;
gap> IsZeroSimpleSemigroup(S);
false
```

Chapter 15

Properties and attributes specific to inverse semigroups

In this chapter we describe the attributes which are specific to inverse semigroups that can be determined using Semigroups.

The functions

- `IsJoinIrreducible` (15.2.5)
- `IsMajorantlyClosed` (15.2.6)
- `JoinIrreducibleDClasses` (15.1.2)
- `MajorantClosure` (15.1.3)
- `Minorants` (15.1.4)
- `RightCosetsOfInverseSemigroup` (15.1.6)
- `SmallerDegreePartialPermRepresentation` (15.1.8)
- `VagnerPrestonRepresentation` (15.1.9)

were written by Wilf A. Wilson and Robert Hancock.

The function `CharacterTableOfInverseSemigroup` (15.1.10) was written by Jhevon Smith and Ben Steinberg.

15.1 Attributes specific to inverse semigroups

15.1.1 NaturalLeqInverseSemigroup

▷ `NaturalLeqInverseSemigroup(S)` (attribute)

Returns: An function.

`NaturalLeqInverseSemigroup` returns a function that, when given two elements `x, y` of the inverse semigroup S, returns `true` if `x` is less than or equal to `y` in the natural partial order on S.

Example
```
gap> S := Monoid(Transformation([1, 3, 4, 4]),
>               Transformation([1, 4, 2, 4]));
<transformation monoid of degree 4 with 2 generators>
gap> IsInverseSemigroup(S);
true
gap> Size(S);
6
gap> NaturalPartialOrder(S);
[ [ 2, 5, 6 ], [ 6 ], [ 6 ], [ 6 ], [ 6 ], [  ] ]
```

15.1.2 JoinIrreducibleDClasses

▷ `JoinIrreducibleDClasses(S)` (attribute)

Returns: A list of $\mathscr{D}$-classes.

`JoinIrreducibleDClasses` returns a list of the join irreducible $\mathscr{D}$-classes of the inverse semigroup of partial permutations, block bijections or partial permutation bipartitions S.

A *join irreducible* $\mathscr{D}$*-class* is a $\mathscr{D}$-class containing only join irreducible elements. See `IsJoinIrreducible` (15.2.5). If a $\mathscr{D}$-class contains one join irreducible element, then all of the elements in the $\mathscr{D}$-class are join irreducible.

Example
```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> JoinIrreducibleDClasses(S);
[ <Green's D-class: <identity partial perm on [ 2 ]>> ]
gap> T := InverseSemigroup([
>   PartialPerm([1, 2, 4, 3]),
>   PartialPerm([1]),
>   PartialPerm([0, 2])]);
<inverse partial perm semigroup of rank 4 with 3 generators>
gap> JoinIrreducibleDClasses(T);
[ <Green's D-class: <identity partial perm on [ 1, 2, 3, 4 ]>>,
  <Green's D-class: <identity partial perm on [ 1 ]>>,
  <Green's D-class: <identity partial perm on [ 2 ]>> ]
gap> D := DualSymmetricInverseSemigroup(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> JoinIrreducibleDClasses(D);
[ <Green's D-class: <block bijection: [ 1, 2, -1, -2 ], [ 3, -3 ]>> ]
```

15.1.3 MajorantClosure

▷ `MajorantClosure(S, T)` (operation)

Returns: A majorantly closed list of elements.

`MajorantClosure` returns a majorantly closed subset of an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions, S, as a list. See `IsMajorantlyClosed` (15.2.6).

The result contains all elements of S which are greater than or equal to any element of T (with respect to the natural partial order `NaturalLeqPartialPerm` (**Reference: NaturalLeqPartialPerm**)). In particular, the result is a superset of T.

Note that T can be a subset of S or a subsemigroup of S.

Example

```
gap> S := SymmetricInverseSemigroup(4);
<symmetric inverse monoid of degree 4>
gap> T := [PartialPerm([1, 0, 3, 0])];
[ <identity partial perm on [ 1, 3 ]> ]
gap> U := MajorantClosure(S, T);
[ <identity partial perm on [ 1, 3 ]>,
  <identity partial perm on [ 1, 2, 3 ]>, [2,4](1)(3), [4,2](1)(3),
  <identity partial perm on [ 1, 3, 4 ]>,
  <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2,4)(3) ]
gap> B := InverseSemigroup([
> Bipartition([[1, -2], [2, -1], [3, -3], [4, 5, -4, -5]]),
> Bipartition([[1, -3], [2, -4], [3, -2], [4, -1], [5, -5]])]);;
gap> T := [Bipartition([[1, -2], [2, 3, 5, -1, -3, -5], [4, -4]]),
> Bipartition([[1, -4], [2, 3, 5, -1, -3, -5], [4, -2]])];;
gap> IsMajorantlyClosed(B, T);
false
gap> MajorantClosure(B, T);
[ <block bijection: [ 1, -2 ], [ 2, 3, 5, -1, -3, -5 ], [ 4, -4 ]>,
  <block bijection: [ 1, -4 ], [ 2, 3, 5, -1, -3, -5 ], [ 4, -2 ]>,
  <block bijection: [ 1, -2 ], [ 2, 5, -1, -5 ], [ 3, -3 ], [ 4, -4 ]>
    , <block bijection: [ 1, -2 ], [ 2, -1 ], [ 3, 5, -3, -5 ],
     [ 4, -4 ]>,
  <block bijection: [ 1, -4 ], [ 2, 5, -3, -5 ], [ 3, -1 ], [ 4, -2 ]>
    , <block bijection: [ 1, -4 ], [ 2, -3 ], [ 3, 5, -1, -5 ],
     [ 4, -2 ]>, <block bijection: [ 1, -4 ], [ 2, -3 ], [ 3, -1 ],
     [ 4, -2 ], [ 5, -5 ]> ]
gap> IsMajorantlyClosed(B, last);
true
```

15.1.4 Minorants

▷ `Minorants(S, f)` (operation)

Returns: A list of elements.

`Minorants` takes an element f from an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions S, and returns a list of the minorants of f in S.

A *minorant* of f is an element of S which is strictly less than f in the natural partial order of S. See `NaturalLeqPartialPerm` (**Reference: NaturalLeqPartialPerm**).

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> x := Elements(S)[13];
[1,3](2)
gap> Minorants(S, x);
[ <empty partial perm>, [1,3], <identity partial perm on [ 2 ]> ]
gap> x := PartialPerm([3, 2, 4, 0]);
[1,3,4](2)
gap> S := InverseSemigroup(x);
<inverse partial perm semigroup of rank 4 with 1 generator>
gap> Minorants(S, x);
[ <identity partial perm on [ 2 ]>, [1,3](2), [3,4](2) ]
```

15.1.5 PrimitiveIdempotents

▷ PrimitiveIdempotents(*S*) (attribute)

Returns: A list of elements.

An idempotent in an inverse semigroup *S* is *primitive* if it is non-zero and minimal with respect to the NaturalPartialOrder (**Reference: NaturalPartialOrder**) on *S*. PrimitiveIdempotents returns the list of primitive idempotents in the inverse semigroup *S*.

Example

```
gap> S := InverseMonoid(
> PartialPerm([1], [4]),
> PartialPerm([1, 2, 3], [2, 1, 3]),
> PartialPerm([1, 2, 3], [3, 1, 2]));;
gap> MultiplicativeZero(S);
<empty partial perm>
gap> Set(PrimitiveIdempotents(S));
[ <identity partial perm on [ 1 ]>, <identity partial perm on [ 2 ]>,
  <identity partial perm on [ 3 ]>, <identity partial perm on [ 4 ]> ]
gap> S := DualSymmetricInverseMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> Set(PrimitiveIdempotents(S));
[ <block bijection: [ 1, 2, 3, -1, -2, -3 ], [ 4, -4 ]>,
  <block bijection: [ 1, 2, 4, -1, -2, -4 ], [ 3, -3 ]>,
  <block bijection: [ 1, 2, -1, -2 ], [ 3, 4, -3, -4 ]>,
  <block bijection: [ 1, 3, 4, -1, -3, -4 ], [ 2, -2 ]>,
  <block bijection: [ 1, 3, -1, -3 ], [ 2, 4, -2, -4 ]>,
  <block bijection: [ 1, 4, -1, -4 ], [ 2, 3, -2, -3 ]>,
  <block bijection: [ 1, -1 ], [ 2, 3, 4, -2, -3, -4 ]> ]
```

15.1.6 RightCosetsOfInverseSemigroup

▷ RightCosetsOfInverseSemigroup(*S*, *T*) (operation)

Returns: A list of lists of elements.

RightCosetsOfInverseSemigroup takes a majorantly closed inverse subsemigroup *T* of an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions *S*. See IsMajorantlyClosed (15.2.6). The result is a list of the right cosets of *T* in *S*.

For $s \in S$, the right coset $\overline{Ts}$ is defined if and only if $ss^{-1} \in T$, in which case it is defined to be the majorant closure of the set Ts. See MajorantClosure (15.1.3). Distinct cosets are disjoint but do not necessarily partition *S*.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> T := InverseSemigroup(MajorantClosure(S, [PartialPerm([1])]));
<inverse partial perm monoid of rank 3 with 6 generators>
gap> IsMajorantlyClosed(S, T);
true
gap> RC := RightCosetsOfInverseSemigroup(S, T);
[ [ <identity partial perm on [ 1 ]>,
      <identity partial perm on [ 1, 2 ]>, [2,3](1), [3,2](1),
      <identity partial perm on [ 1, 3 ]>,
      <identity partial perm on [ 1, 2, 3 ]>, (1)(2,3) ],
```

```
  [ [1,3], [2,1,3], [1,3](2), (1,3), [1,3,2], (1,3,2), (1,3)(2) ],
  [ [1,2], (1,2), [1,2,3], [3,1,2], [1,2](3), (1,2)(3), (1,2,3) ] ]
```

15.1.7 SameMinorantsSubgroup

▷ `SameMinorantsSubgroup(H)` (attribute)

Returns: A list of elements of the group $\mathscr{H}$-class *H*.

Given a group $\mathscr{H}$-class *H* in an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions `S`, `SameMinorantsSubgroup` returns a list of the elements of *H* which have the same strict minorants as the identity element of *H*. A *strict minorant* of `x` in *H* is an element of `S` which is less than `x` (with respect to the natural partial order), but is not equal to `x`.

The returned list of elements of *H* describe a subgroup of *H*.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> H := GroupHClass(DClass(S, PartialPerm([1, 2, 3])));
<Green's H-class: <identity partial perm on [ 1, 2, 3 ]>>
gap> Elements(H);
[ <identity partial perm on [ 1, 2, 3 ]>, (1)(2,3), (1,2)(3),
  (1,2,3), (1,3,2), (1,3)(2) ]
gap> SameMinorantsSubgroup(H);
[ <identity partial perm on [ 1, 2, 3 ]> ]
gap> T := InverseSemigroup(
> PartialPerm([1, 2, 3, 4], [1, 2, 4, 3]),
> PartialPerm([1], [1]), PartialPerm([2], [2]));
<inverse partial perm semigroup of rank 4 with 3 generators>
gap> Elements(T);
[ <empty partial perm>, <identity partial perm on [ 1 ]>,
  <identity partial perm on [ 2 ]>,
  <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
gap> x := GroupHClass(DClass(T, PartialPerm([1, 2, 3, 4])));
<Green's H-class: <identity partial perm on [ 1, 2, 3, 4 ]>>
gap> Elements(x);
[ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
gap> AsSet(SameMinorantsSubgroup(x));
[ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
```

15.1.8 SmallerDegreePartialPermRepresentation

▷ `SmallerDegreePartialPermRepresentation(S)` (attribute)

Returns: An isomorphism.

`SmallerDegreePartialPermRepresentation` attempts to find an isomorphism from the inverse semigroup *S* to an inverse semigroup of partial permutations with small degree. If *S* is already a partial permutation semigroup, and the function cannot reduce the degree, the identity mapping is returned.

There is no guarantee that the smallest possible degree representation is returned. For more information see [Sch92].

Example

```
gap> S := InverseSemigroup(PartialPerm([2, 1, 4, 3, 6, 5, 8, 7]));
<partial perm group of rank 8 with 1 generator>
```

```
gap> Elements(S);
[ <identity partial perm on [ 1, 2, 3, 4, 5, 6, 7, 8 ]>,
  (1,2)(3,4)(5,6)(7,8) ]
gap> T := SmallerDegreePartialPermRepresentation(S);
MappingByFunction( <partial perm group of size 2, rank 8 with
  1 generator>, <partial perm group of rank 2 with 1 generator>
, function( x ) ... end, function( x ) ... end )
gap> R := Range(T);
<partial perm group of rank 2 with 1 generator>
gap> Elements(R);
[ <identity partial perm on [ 1, 2 ]>, (1,2) ]
gap> S := DualSymmetricInverseMonoid(5);;
gap> T := Range(IsomorphismPartialPermSemigroup(S));
<inverse partial perm monoid of size 6721, rank 6721 with 3
 generators>
gap> SmallerDegreePartialPermRepresentation(T);
MappingByFunction( <inverse partial perm monoid of size 6721,
 rank 6721 with 3 generators>, <inverse partial perm semigroup of
 rank 30 with 3 generators>
 , function( x ) ... end, function( x ) ... end )
```

15.1.9 VagnerPrestonRepresentation

▷ `VagnerPrestonRepresentation(S)` (attribute)

Returns: An isomorphism to an inverse semigroup of partial permutations.

`VagnerPrestonRepresentation` returns an isomorphism from an inverse semigroup S where the elements of S have a unique semigroup inverse accessible via `Inverse` (**Reference: Inverse**), to the inverse semigroup of partial permutations T of degree equal to the size of S, which is obtained using the Vagner-Preston Representation Theorem.

More precisely, if $f : S \to T$ is the isomorphism returned by `VagnerPrestonRepresentation(S)` and x is in S, then $f(x)$ is the partial permutation with domain Sx^{-1} and range $Sx^{-1}x$ defined by $f(x) : sx^{-1} \mapsto sx^{-1}x$.

In many cases, it is possible to find a smaller degree representation than that provided by `VagnerPrestonRepresentation` using `IsomorphismPartialPermSemigroup` (**Reference: IsomorphismPartialPermSemigroup**) or `SmallerDegreePartialPermRepresentation` (15.1.8).

Example

```
gap> S := SymmetricInverseSemigroup(2);
<symmetric inverse monoid of degree 2>
gap> Size(S);
7
gap> iso := VagnerPrestonRepresentation(S);
MappingByFunction( <symmetric inverse monoid of degree 2>,
<inverse partial perm monoid of rank 7 with 2 generators>
 , function( x ) ... end, function( x ) ... end )
gap> RespectsMultiplication(iso);
true
gap> inv := InverseGeneralMapping(iso);;
gap> ForAll(S, x -> (x ^ iso) ^ inv = x);
true
gap> V := InverseSemigroup(
```

```
> Bipartition([[1, -4], [2, -1], [3, -5], [4], [5], [-2], [-3]]),
> Bipartition([[1, -5], [2, -1], [3, -3], [4], [5], [-2], [-4]]),
> Bipartition([[1, -2], [2, -4], [3, -5], [4, -1], [5, -3]]));
<inverse bipartition semigroup of degree 5 with 3 generators>
gap> IsInverseSemigroup(V);
true
gap> VagnerPrestonRepresentation(V);
MappingByFunction( <inverse bipartition semigroup of size 394,
 degree 5 with 3 generators>, <inverse partial perm semigroup of
 rank 394 with 5 generators>
 , function( x ) ... end, function( x ) ... end )
```

15.1.10 CharacterTableOfInverseSemigroup

▷ `CharacterTableOfInverseSemigroup(S)` (attribute)

Returns: The character table of the inverse semigroup S and a list of conjugacy class representatives of S.

Returns a list with two entries: the first entry being the character table of the inverse semigroup S as a matrix, while the second entry is a list of conjugacy class representatives of S.

The order of the columns in the character table matrix follows the order of the conjugacy class representatives list. The conjugacy representatives are grouped by $\mathscr{D}$-class and then sorted by rank. Also, as is typical of character tables, the rows of the matrix correspond to the irreducible characters and the columns correspond to the conjugacy classes.

This function was contributed by Jhevon Smith and Ben Steinberg.

Example
```
gap> S := InverseMonoid([
>   PartialPerm([1, 2], [3, 1]),
>  PartialPerm([1, 2, 3], [1, 3, 4]),
>  PartialPerm([1, 2, 3], [2, 4, 1]),
>  PartialPerm([1, 3, 4], [3, 4, 1])]);;
gap> CharacterTableOfInverseSemigroup(S);
[ [ [ 1, 0, 0, 0, 0, 0, 0, 0 ], [ 3, 1, 1, 1, 0, 0, 0, 0 ],
      [ 3, 1, E(3), E(3)^2, 0, 0, 0, 0 ],
      [ 3, 1, E(3)^2, E(3), 0, 0, 0, 0 ], [ 6, 3, 0, 0, 1, -1, 0, 0 ],
      [ 6, 3, 0, 0, 1, 1, 0, 0 ], [ 4, 3, 0, 0, 2, 0, 1, 0 ],
      [ 1, 1, 1, 1, 1, 1, 1, 1 ] ],
  [ <identity partial perm on [ 1, 2, 3, 4 ]>,
      <identity partial perm on [ 1, 3, 4 ]>, (1,3,4), (1,4,3),
      <identity partial perm on [ 1, 3 ]>, (1,3),
      <identity partial perm on [ 3 ]>, <empty partial perm> ] ]
gap> S := SymmetricInverseMonoid(4);;
gap> CharacterTableOfInverseSemigroup(S);
[ [ [ 1, -1, 1, 1, -1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 3, -1, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 2, 0, -1, 2, 0, 0, 0, 0, 0, 0, 0, 0 ],
      [ 3, 1, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 4, -2, 1, 0, 0, 1, -1, 1, 0, 0, 0, 0 ],
      [ 8, 0, -1, 0, 0, 2, 0, -1, 0, 0, 0, 0 ],
      [ 4, 2, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0 ],
      [ 6, 0, 0, -2, 0, 3, -1, 0, 1, -1, 0, 0 ],
```

```
        [ 6, 2, 0, 2, 0, 3, 1, 0, 1, 1, 0, 0 ],
        [ 4, 2, 1, 0, 0, 3, 1, 0, 2, 0, 1, 0 ],
        [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ],
    [ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4),
        (1)(2,3,4), (1,2)(3,4), (1,2,3,4),
        <identity partial perm on [ 1, 2, 3 ]>, (1)(2,3), (1,2,3),
        <identity partial perm on [ 2, 3 ]>, (2,3),
        <identity partial perm on [ 1 ]>, <empty partial perm> ] ]
```

15.2 Properties of inverse semigroups

15.2.1 IsCliffordSemigroup

▷ `IsCliffordSemigroup(S)` (property)

Returns: `true` or `false`.

`IsCliffordSemigroup` returns `true` if the semigroup S is regular and its idempotents are central, and `false` if it is not.

Example

```
gap> S := Semigroup(Transformation([1, 2, 4, 5, 6, 3, 7, 8]),
>                   Transformation([3, 3, 4, 5, 6, 2, 7, 8]),
>                   Transformation([1, 2, 5, 3, 6, 8, 4, 4]));;
gap> IsCliffordSemigroup(S);
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsCliffordSemigroup(S);
true
gap> S := DualSymmetricInverseMonoid(5);;
gap> T := IdempotentGeneratedSubsemigroup(S);;
gap> IsCliffordSemigroup(T);
true
```

15.2.2 IsBrandtSemigroup

▷ `IsBrandtSemigroup(S)` (property)

Returns: `true` or `false`.

`IsBrandtSemigroup` return `true` if the semigroup S is a finite 0-simple inverse semigroup, and `false` if it is not. See also `IsZeroSimpleSemigroup` (14.1.27) and `IsInverseSemigroup` (**Reference: IsInverseSemigroup**).

Example

```
gap> S := Semigroup(
> Transformation([2, 8, 8, 8, 8, 8, 8, 8]),
> Transformation([5, 8, 8, 8, 8, 8, 8, 8]),
> Transformation([8, 3, 8, 8, 8, 8, 8, 8]),
> Transformation([8, 6, 8, 8, 8, 8, 8, 8]),
> Transformation([8, 8, 1, 8, 8, 8, 8, 8]),
> Transformation([8, 8, 8, 1, 8, 8, 8, 8]),
> Transformation([8, 8, 8, 8, 4, 8, 8, 8]),
> Transformation([8, 8, 8, 8, 8, 7, 8, 8]),
> Transformation([8, 8, 8, 8, 8, 8, 2, 8]));;
gap> IsBrandtSemigroup(S);
```

```
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsBrandtSemigroup(T);
true
gap> S := DualSymmetricInverseMonoid(4);;
gap> D := DClass(S,
>               Bipartition([[1, 2, 3, -1, -2, -3], [4, -4]]));;
gap> R := InjectionPrincipalFactor(D);;
gap> S := Semigroup(PreImages(R, GeneratorsOfSemigroup(Range(R))));;
gap> IsBrandtSemigroup(S);
true
```

15.2.3 IsEUnitaryInverseSemigroup

▷ IsEUnitaryInverseSemigroup(*S*) (property)

Returns: true or false.

As described in Section 5.9 of [How95], an inverse semigroup S with semilattice of idempotents E is *E-unitary* if for

$$s \in S \text{ and } e \in E\text{: } es \in E \Rightarrow s \in E.$$

Equivalently, S is *E-unitary* if E is closed in the natural partial order (see Proposition 5.9.1 in [How95]):

$$\text{for } s \in S \text{ and } e \in E\text{: } e \leq s \Rightarrow s \in E.$$

This condition is equivalent to E being majorantly closed in S. See IdempotentGeneratedSubsemigroup (13.9.3) and IsMajorantlyClosed (15.2.6). Hence an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions is *E-unitary* if and only if the idempotent semilattice is majorantly closed.

Example

```
gap> S := InverseSemigroup(
>  PartialPerm([1, 2, 3, 4], [2, 3, 1, 6]),
>  PartialPerm([1, 2, 3, 5], [3, 2, 1, 6]));;
gap> IsEUnitaryInverseSemigroup(S);
true
gap> e := IdempotentGeneratedSubsemigroup(S);;
gap> ForAll(Difference(S, e), x -> not ForAny(e, y -> y * x in e));
true
gap> T := InverseSemigroup([
>  PartialPerm([1, 3, 4, 6, 8], [2, 5, 10, 7, 9]),
>  PartialPerm([1, 2, 3, 5, 6, 7, 8], [5, 8, 9, 2, 10, 1, 3]),
>  PartialPerm([1, 2, 3, 5, 6, 7, 9], [9, 8, 4, 1, 6, 7, 2])]);;
gap> IsEUnitaryInverseSemigroup(T);
false
gap> U := InverseSemigroup([
>  PartialPerm([1, 2, 3, 4, 5], [2, 3, 4, 5, 1]),
>  PartialPerm([1, 2, 3, 4, 5], [2, 1, 3, 4, 5])]);;
gap> IsEUnitaryInverseSemigroup(U);
true
gap> IsGroupAsSemigroup(U);
true
gap> StructureDescription(U);
"S5"
```

15.2.4 IsFactorisableInverseMonoid

▷ `IsFactorisableInverseMonoid(S)` (property)

Returns: `true` or `false`.

An inverse monoid is *factorisable* if every element is the product of an element of the group of units and an idempotent; see also `GroupOfUnits` (13.8.1) and `Idempotents` (13.9.1). Hence an inverse semigroup of partial permutations is factorisable if and only if each of its generators is the restriction of some element in the group of units.

Example

```
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 4], [3, 1, 4]),
> PartialPerm([1, 2, 3, 5], [4, 1, 5, 2]));;
gap> IsFactorisableInverseMonoid(S);
false
gap> IsFactorisableInverseMonoid(SymmetricInverseSemigroup(5));
true
gap> IsFactorisableInverseMonoid(DualSymmetricInverseMonoid(5));
false
gap> S := FactorisableDualSymmetricInverseMonoid(5);;
gap> IsFactorisableInverseMonoid(S);
true
```

15.2.5 IsJoinIrreducible

▷ `IsJoinIrreducible(S, x)` (operation)

Returns: `true` or `false`.

`IsJoinIrreducible` determines whether an element *x* of an inverse semigroup *S* of partial permutations, block bijections or partial permutation bipartitions is join irreducible.

An element *x* is *join irreducible* when it is not the least upper bound (with respect to the natural partial order `NaturalLeqPartialPerm` (**Reference: NaturalLeqPartialPerm**)) of any subset of *S* not containing *x*.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> x := PartialPerm([1, 2, 3]);
<identity partial perm on [ 1, 2, 3 ]>
gap> IsJoinIrreducible(S, x);
false
gap> T := InverseSemigroup([
>  PartialPerm([1, 2, 4, 3]),
>  PartialPerm([1]),
>  PartialPerm([0, 2])]);
<inverse partial perm semigroup of rank 4 with 3 generators>
gap> y := PartialPerm([1, 2, 3, 4]);
<identity partial perm on [ 1, 2, 3, 4 ]>
gap> IsJoinIrreducible(T, y);
true
gap> B := InverseSemigroup([
>  Bipartition([
>    [1, -5], [2, -2], [3, 5, 6, 7, -1, -4, -6, -7], [4, -3]]),
>  Bipartition([
```

```
>    [1, -1], [2, -3], [3, -4], [4, 5, 7, -2, -6, -7], [6, -5]]),
>  Bipartition([
>    [1, -2], [2, -4], [3, -6], [4, -1], [5, 7, -3, -7], [6, -5]]),
>  Bipartition([
>    [1, -5], [2, -1], [3, -6], [4, 5, 7, -2, -4, -7], [6, -3]])]);
<inverse block bijection semigroup of degree 7 with 4 generators>
gap> x := Bipartition([
>   [1, 2, 3, 5, 6, 7, -2, -3, -4, -5, -6, -7], [4, -1]]);
<block bijection: [ 1, 2, 3, 5, 6, 7, -2, -3, -4, -5, -6, -7 ],
 [ 4, -1 ]>
gap> IsJoinIrreducible(B, x);
true
gap> IsJoinIrreducible(B, B.1);
false
```

15.2.6 IsMajorantlyClosed

▷ `IsMajorantlyClosed(S, T)` (operation)

Returns: `true` or `false`.

`IsMajorantlyClosed` determines whether the subset T of the inverse semigroup of partial permutations, block bijections or partial permutation bipartitions S is majorantly closed in S. See also `MajorantClosure` (15.1.3).

We say that T is *majorantly closed* in S if it contains all elements of S which are greater than or equal to any element of T, with respect to the natural partial order. See `NaturalLeqPartialPerm` (**Reference: NaturalLeqPartialPerm**).

Note that T can be a subset of S or a subsemigroup of S.

Example

```
gap> S := SymmetricInverseSemigroup(2);
<symmetric inverse monoid of degree 2>
gap> T := [Elements(S)[2]];
[ <identity partial perm on [ 1 ]> ]
gap> IsMajorantlyClosed(S, T);
false
gap> U := [Elements(S)[2], Elements(S)[6]];
[ <identity partial perm on [ 1 ]>, <identity partial perm on [ 1, 2 ]
    > ]
gap> IsMajorantlyClosed(S, U);
true
gap> D := DualSymmetricInverseSemigroup(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> x := Bipartition([[1, -2], [2, -3], [3, -1]]);;
gap> IsMajorantlyClosed(D, [x]);
true
gap> y := Bipartition([[1, 2, -1, -2], [3, -3]]);;
gap> IsMajorantlyClosed(D, [x, y]);
false
```

15.2.7 IsMonogenicInverseSemigroup

▷ IsMonogenicInverseSemigroup(*S*) (property)

Returns: true or false.

IsMonogenicInverseSemigroup returns true if the semigroup *S* is a monogenic inverse semigroup and it returns false if it is not.

A inverse semigroup is *monogenic* if it is generated as an inverse semigroup by a single element. See also IsMonogenicSemigroup (14.1.10) and IsMonogenicInverseMonoid (15.2.8).

Example
```
gap> x := PartialPerm([1, 2, 3, 6, 8, 10], [2, 6, 7, 9, 1, 5]);;
gap> S := InverseSemigroup(x, x ^ 2, x ^ 3);;
gap> IsMonogenicSemigroup(S);
false
gap> IsMonogenicInverseSemigroup(S);
true
gap> x := RandomBlockBijection(100);;
gap> S := InverseSemigroup(x, x ^ 2, x ^ 20);;
gap> IsMonogenicInverseSemigroup(S);
true
gap> S := SymmetricInverseSemigroup(5);;
gap> IsMonogenicInverseSemigroup(S);
false
```

15.2.8 IsMonogenicInverseMonoid

▷ IsMonogenicInverseMonoid(*S*) (property)

Returns: true or false.

IsMonogenicInverseMonoid returns true if the monoid *S* is a monogenic inverse monoid and it returns false if it is not.

A inverse monoid is *monogenic* if it is generated as an inverse monoid by a single element. See also IsMonogenicInverseSemigroup (15.2.7) and IsMonogenicMonoid (14.1.11).

Example
```
gap> x := PartialPerm([1, 2, 3, 6, 8, 10], [2, 6, 7, 9, 1, 5]);;
gap> S := InverseMonoid(x, x ^ 2, x ^ 3);;
gap> IsMonogenicMonoid(S);
false
gap> IsMonogenicInverseSemigroup(S);
false
gap> IsMonogenicInverseMonoid(S);
true
gap> x := RandomBlockBijection(100);;
gap> S := InverseMonoid(x, x ^ 2, x ^ 20);;
gap> IsMonogenicInverseMonoid(S);
true
gap> S := SymmetricInverseMonoid(5);;
gap> IsMonogenicInverseMonoid(S);
false
```

Chapter 16

Congruences

Congruences in Semigroups can be described in several different ways:

- Generating pairs – the minimal congruence which contains these pairs
- Rees congruences – the congruence specified by a given ideal
- Universal congruences – the unique congruence with only one class
- Linked triples – only for simple or 0-simple semigroups (see below)
- Kernel and trace – only for inverse semigroups

The operation `SemigroupCongruence` (16.2.1) can be used to create any of these, interpreting the arguments in a smart way. The usual way of specifying a congruence will be by giving a set of generating pairs, but a user with an ideal could instead create a Rees congruence or universal congruence.

If a congruence is specified by generating pairs on a simple, 0-simple, or inverse semigroup, then the congruence may be converted automatically to one of the last two items in the above list, to reduce the complexity of any calculations to be performed. The user need not manually specify, or even be aware of, the congruence's linked triple or kernel and trace.

We can also create left congruences and right congruences, using the `LeftSemigroupCongruence` (16.2.2) and `RightSemigroupCongruence` (16.2.3) functions.

Please note that congruence objects made in GAP before loading the Semigroups package may not behave correctly after Semigroups is loaded. If Semigroups is loaded at the beginning of the session, or before any congruence work is done, then the objects should behave correctly.

16.1 Semigroup congruence objects

16.1.1 IsSemigroupCongruence

▷ `IsSemigroupCongruence(obj)` (property)

A semigroup congruence `cong` is an equivalence relation on a semigroup `S` which respects left and right multiplication.

That is, if (a,b) is a pair in `cong`, and x is an element of `S`, then (ax,bx) and (xa,xb) are both in `cong`.

The simplest way of creating a congruence in Semigroups is by a set of *generating pairs*. See SemigroupCongruence (16.2.1).

Example
```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>               Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>               Transformation([3, 4, 3, 4, 3])];;
gap> cong := SemigroupCongruence(S, [pair1, pair2]);
<semigroup congruence over <simple transformation semigroup of
 degree 5 with 4 generators> with linked triple (2,4,1)>
gap> IsSemigroupCongruence(cong);
true
```

16.1.2 IsLeftSemigroupCongruence

▷ IsLeftSemigroupCongruence(*obj*) (property)

A left semigroup congruence cong is an equivalence relation on a semigroup S which respects left multiplication.

That is, if (a,b) is a pair in cong, and x is an element of S, then (xa,xb) is also in cong.

The simplest way of creating a left congruence in Semigroups is by a set of *generating pairs*. See LeftSemigroupCongruence (16.2.2).

Example
```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>               Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>               Transformation([3, 4, 3, 4, 3])];;
gap> cong := LeftSemigroupCongruence(S, [pair1, pair2]);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
gap> IsLeftSemigroupCongruence(cong);
true
```

16.1.3 IsRightSemigroupCongruence

▷ IsRightSemigroupCongruence(*obj*) (property)

A right semigroup congruence cong is an equivalence relation on a semigroup S which respects right multiplication.

That is, if (a,b) is a pair in cong, and x is an element of S, then (ax,bx) is also in cong.

The simplest way of creating a right congruence in Semigroups is by a set of *generating pairs*. See RightSemigroupCongruence (16.2.3).

Example
```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> RightSemigroupCongruence(S, [pair1, pair2]);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
gap> IsRightSemigroupCongruence(cong);
true
```

16.2 Creating congruences

16.2.1 SemigroupCongruence

▷ SemigroupCongruence(*S*, *pairs*) (function)

Returns: A semigroup congruence.

This function returns a semigroup congruence over the semigroup *S*.

If *pairs* is a list of lists of size 2 with elements from *S*, then this function will return the semigroup congruence defined by these generating pairs. The individual pairs may instead be given as separate arguments.

Example
```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> SemigroupCongruence(S, [pair1, pair2]);
<semigroup congruence over <simple transformation semigroup of
 degree 5 with 4 generators> with linked triple (2,4,1)>
gap> SemigroupCongruence(S, pair1, pair2);
<semigroup congruence over <simple transformation semigroup of
 degree 5 with 4 generators> with linked triple (2,4,1)>
```

16.2.2 LeftSemigroupCongruence

▷ LeftSemigroupCongruence(*S*, *pairs*) (function)

Returns: A left semigroup congruence.

This function returns a left semigroup congruence over the semigroup *S*.

If *pairs* is a list of lists of size 2 with elements from S, then this function will return the least left semigroup congruence on S which contains these generating pairs. The individual pairs may instead be given as separate arguments.

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> LeftSemigroupCongruence(S, [pair1, pair2]);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
gap> LeftSemigroupCongruence(S, pair1, pair2);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
```

16.2.3 RightSemigroupCongruence

▷ RightSemigroupCongruence(*S*, *pairs*) (function)

Returns: A right semigroup congruence.

This function returns a right semigroup congruence over the semigroup S.

If *pairs* is a list of lists of size 2 with elements from S, then this function will return the least right semigroup congruence on S which contains these generating pairs. The individual pairs may instead be given as separate arguments.

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> RightSemigroupCongruence(S, [pair1, pair2]);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
gap> RightSemigroupCongruence(S, pair1, pair2);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
```

16.2.4 GeneratingPairsOfSemigroupCongruence

▷ GeneratingPairsOfSemigroupCongruence(*cong*) (attribute)

▷ GeneratingPairsOfLeftSemigroupCongruence(*cong*) (attribute)

▷ `GeneratingPairsOfRightSemigroupCongruence(cong)` (attribute)

Returns: A list of lists.

If *cong* is a semigroup congruence, then `GeneratingPairsOfSemigroupCongruence` returns a list of pairs of elements from `Range(cong)` that *generates* the congruence; i.e. *cong* is the least congruence on the semigroup which contains all the pairs in the list.

If *cong* is a left or right semigroup congruence, then `GeneratingPairsOfLeft/RightSemigroupCongruence` will instead give a list of pairs which generate it as a left or right congruence. Note that, although a congruence is also a left and right congruence, its generating pairs as a left or right congruence may differ from its generating pairs as a two-sided congruence.

A congruence can be defined using a set of generating pairs: see `SemigroupCongruence` (16.2.1), `LeftSemigroupCongruence` (16.2.2), and `RightSemigroupCongruence` (16.2.3).

Example
```
gap> S := Semigroup([Transformation([3, 3, 2, 3]),
>                    Transformation([3, 4, 4, 1])]);;
gap> pairs :=
>      [[Transformation([1, 1, 1, 1]), Transformation([2, 2, 2, 3])],
>       [Transformation([2, 2, 3, 2]), Transformation([3, 3, 2, 3])]];;
gap> cong := SemigroupCongruence(S, pairs);;
gap> GeneratingPairsOfSemigroupCongruence(cong);
[ [ Transformation( [ 1, 1, 1, 1 ] ),
      Transformation( [ 2, 2, 2, 3 ] ) ],
  [ Transformation( [ 2, 2, 3, 2 ] ),
      Transformation( [ 3, 3, 2, 3 ] ) ] ]
```

16.3 Congruence classes

16.3.1 IsCongruenceClass

▷ `IsCongruenceClass(obj)` (category)

This category contains any object which is an equivalence class of a semigroup congruence (see `IsSemigroupCongruence` (16.1.1)). An object will only be in this category if the relation is known to be a semigroup congruence when the congruence class is created.

Example
```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S, [Transformation([1, 2, 1]),
>                                    Transformation([2, 1, 2])]);;
gap> class := EquivalenceClassOfElement(cong,
>                                       Transformation([3, 1, 1]));
<congruence class of Transformation( [ 3, 1, 1 ] )>
gap> IsCongruenceClass(class);
true
```

16.3.2 IsLeftCongruenceClass

▷ `IsLeftCongruenceClass(obj)` (category)

This category contains any object which is an equivalence class of a left semigroup congruence (see IsLeftSemigroupCongruence (16.1.2)). An object will only be in this category if the relation is known to be a left semigroup congruence when the class is created.

Example
```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> pairs := [Transformation([1, 2, 1]),
>              Transformation([2, 1, 2])];;
gap> cong := LeftSemigroupCongruence(S, pairs);;
gap> class := EquivalenceClassOfElement(cong,
>                                       Transformation([3, 1, 1]));
<left congruence class of Transformation( [ 3, 1, 1 ] )>
gap> IsLeftCongruenceClass(class);
true
```

16.3.3 IsRightCongruenceClass

▷ IsRightCongruenceClass(*obj*) (category)

This category contains any object which is an equivalence class of a right semigroup congruence (see IsRightSemigroupCongruence (16.1.3)). An object will only be in this category if the relation is known to be a right semigroup congruence when the class is created.

Example
```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> pairs := [Transformation([1, 2, 1]),
>              Transformation([2, 1, 2])];;
gap> cong := RightSemigroupCongruence(S, pairs);;
gap> class := EquivalenceClassOfElement(cong,
>                                       Transformation([3, 1, 1]));
<right congruence class of Transformation( [ 3, 1, 1 ] )>
gap> IsRightCongruenceClass(class);
true
```

16.3.4 CongruenceClassOfElement

▷ CongruenceClassOfElement(*cong*, *elm*) (operation)
▷ LeftCongruenceClassOfElement(*cong*, *elm*) (operation)
▷ RightCongruenceClassOfElement(*cong*, *elm*) (operation)

Returns: An equivalence class.

These operations act as a synonym of EquivalenceClassOfElement in the case that the argument *cong* is a congruence, left congruence, or right congruence (respectively) of a semigroup.

See IsLeftSemigroupCongruence (16.1.2), IsRightSemigroupCongruence (16.1.3), and IsSemigroupCongruence (16.1.1).

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong := CongruencesOfSemigroup(S)[3];;
gap> elm := ReesZeroMatrixSemigroupElement(S, 1, (1, 3, 2), 1);;
```

```
gap> CongruenceClassOfElement(cong, elm);
<congruence class of (1,(1,3,2),1)>
```

16.3.5 CongruenceClasses

▷ CongruenceClasses(*cong*) (operation)
▷ LeftCongruenceClasses(*cong*) (operation)
▷ RightCongruenceClasses(*cong*) (operation)

Returns: A list of equivalence classes.

These operations act as a synonym of `EquivalenceClasses` in the case that the argument *cong* is a congruence, left congruence, or right congruence (respectively) of a semigroup.

See `IsLeftSemigroupCongruence` (16.1.2), `IsRightSemigroupCongruence` (16.1.3), and `IsSemigroupCongruence` (16.1.1).

Example
```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> pair := [Transformation([1, 2, 1]), Transformation([2, 1, 2])];;
gap> cong := SemigroupCongruence(S, pair);;
gap> classes := CongruenceClasses(cong);;
gap> Set(classes);
[ <congruence class of Transformation( [ 3, 3, 3 ] )>,
  <congruence class of Transformation( [ 2, 1, 2 ] )>,
  <congruence class of Transformation( [ 1, 2, 2 ] )>,
  <congruence class of IdentityTransformation>,
  <congruence class of Transformation( [ 3, 1, 3 ] )>,
  <congruence class of Transformation( [ 3, 1, 1 ] )> ]
```

16.3.6 NonTrivialEquivalenceClasses

▷ NonTrivialEquivalenceClasses(*eq*) (attribute)

Returns: A list of equivalence classes.

If *eq* is an equivalence relation, then this attribute returns a list of all equivalence classes of *eq* which contain more than one element.

Example
```
gap> S := Monoid([Transformation([1, 2, 2]),
>                 Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S, [Transformation([1, 2, 1]),
>                                    Transformation([2, 1, 2])]);;
gap> classes := NonTrivialEquivalenceClasses(cong);;
gap> Set(classes);
[ <congruence class of Transformation( [ 3, 3, 3 ] )>,
  <congruence class of Transformation( [ 2, 1, 2 ] )>,
  <congruence class of Transformation( [ 1, 2, 2 ] )>,
  <congruence class of Transformation( [ 3, 1, 3 ] )>,
  <congruence class of Transformation( [ 3, 1, 1 ] )> ]
```

16.3.7 NonTrivialCongruenceClasses

▷ NonTrivialCongruenceClasses(*cong*) (operation)
▷ NonTrivialLeftCongruenceClasses(*cong*) (operation)

▷ `NonTrivialRightCongruenceClasses(cong)` (operation)

Returns: A list of equivalence classes.

These operations act as a synonym of `NonTrivialEquivalenceClasses` in the case that the argument *cong* is a congruence, left congruence, or right congruence (respectively) of a semigroup.

See `IsLeftSemigroupCongruence` (16.1.2), `IsRightSemigroupCongruence` (16.1.3), and `IsSemigroupCongruence` (16.1.1).

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := RightSemigroupCongruence(S, [Transformation([1, 2, 1]),
>                                         Transformation([2, 1, 2])]);;
gap> classes := NonTrivialRightCongruenceClasses(cong);;
gap> Set(classes);
[ <right congruence class of Transformation( [ 2, 1, 2 ] )>,
  <right congruence class of Transformation( [ 3, 1, 3 ] )> ]
```

16.3.8 NrEquivalenceClasses

▷ `NrEquivalenceClasses(eq)` (attribute)

Returns: A positive integer.

If *eq* is an equivalence relation, then this attribute describes the number of equivalence classes it has.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
>                                  [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong := CongruencesOfSemigroup(S)[3];;
gap> NrEquivalenceClasses(cong);
9
```

16.3.9 NrCongruenceClasses

▷ `NrCongruenceClasses(cong)` (operation)

▷ `NrLeftCongruenceClasses(cong)` (operation)

▷ `NrRightCongruenceClasses(cong)` (operation)

Returns: A list of equivalence classes.

These operations act as a synonym of `NrEquivalenceClasses` in the case that the argument *cong* is a congruence, left congruence, or right congruence (respectively) of a semigroup.

See `IsLeftSemigroupCongruence` (16.1.2), `IsRightSemigroupCongruence` (16.1.3), and `IsSemigroupCongruence` (16.1.1).

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> pair := [Transformation([1, 2, 1]), Transformation([2, 1, 2])];;
gap> cong := SemigroupCongruence(S, pair);;
gap> NrCongruenceClasses(cong);
6
gap> cong := RightSemigroupCongruence(S, pair);;
gap> NrRightCongruenceClasses(cong);
10
```

16.3.10 EquivalenceRelationLookup

▷ `EquivalenceRelationLookup(`*cong*`)` (attribute)

Returns: A list.

This attribute describes the (left, right or two-sided) semigroup congruence *cong* as a list of positive integers with length the size of the finite semigroup over which *cong* is defined.

Each position in the list corresponds to an element of the semigroup (in a consistent canonical order) and the integer at that position is a unique identifier for that element's congruence class under *cong*. Two elements of the semigroup on which the congruence is defined are related in the congruence if and only if they have the same number at their respective positions in the lookup.

Note that the order in which numbers appear in the list is non-deterministic, and two congruence objects which describe the same equivalence relation might therefore have different lookups. Note also that the maximum value of the list may not be the number of classes of *cong*, and that any integer might not be included. However, see `EquivalenceRelationCanonicalLookup` (16.3.11).

See also `EquivalenceRelationPartition` (**Reference: EquivalenceRelationPartition**).

Example
```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S,
> [Transformation([1, 2, 1]), Transformation([2, 1, 2])]);;
gap> lookup := EquivalenceRelationLookup(cong);;
gap> lookup[3] = lookup[8];
true
gap> lookup[2] = lookup[9];
false
```

16.3.11 EquivalenceRelationCanonicalLookup

▷ `EquivalenceRelationCanonicalLookup(`*cong*`)` (attribute)

Returns: A list.

This attribute describes the semigroup congruence *cong* as a list of positive integers with length the size of the finite semigroup over which *cong* is defined.

Each position in the list corresponds to an element of the semigroup (in a consistent canonical order) and the integer at that position is a unique identifier for that element's congruence class under *cong*. The value of `EquivalenceRelationCanonicalLookup` has the property that the first appearance of the value `i` is strictly later than the first appearance of `i-1`, and that all entries in the list will be from the range `[1 .. NrEquivalenceClasses(`*cong*`)]`. As such, two congruences on a given semigroup are equal if and only if their canonical lookups are equal.

Two elements of the semigroup on which the congruence is defined are related in the congruence if and only if they have the same number at their respective positions in the lookup.

See also `EquivalenceRelationLookup` (16.3.10) and `EquivalenceRelationPartition` (**Reference: EquivalenceRelationPartition**).

Example
```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S,
> [Transformation([1, 2, 1]), Transformation([2, 1, 2])]);;
gap> EquivalenceRelationCanonicalLookup(cong);
[ 1, 2, 3, 4, 5, 6, 2, 3, 6, 4, 5, 6 ]
```

16.3.12 EquivalenceRelationCanonicalPartition

▷ `EquivalenceRelationCanonicalPartition(cong)` (attribute)

Returns: A list of lists.

This attribute returns a list of lists of elements of the underlying set of the semigroup congruence *cong*. These lists are precisely the nontrivial equivalence classes of *cong*. The order in which the classes appear is deterministic, and the order of the elements inside each class is also deterministic. Hence, two congruence objects have the same `EquivalenceRelationCanonicalPartition` if and only if they describe the same relation.

See also `EquivalenceRelationPartition` (**Reference: EquivalenceRelationPartition**), a similar attribute which does not have canonical ordering, but which is likely to be faster.

Example

```
gap> S := Semigroup(Transformation([1, 4, 3, 3]),
>                   Transformation([2, 4, 3, 3]));;
gap> cong := SemigroupCongruence(S, [Transformation([1, 4, 3, 3]),
>                                    Transformation([1, 3, 3, 3])]);;
gap> EquivalenceRelationCanonicalPartition(cong);
[ [ Transformation( [ 1, 3, 3, 3 ] ),
      Transformation( [ 1, 4, 3, 3 ] ) ],
  [ Transformation( [ 3, 3, 3, 3 ] ),
      Transformation( [ 4, 3, 3, 3 ] ) ] ]
```

16.3.13 OnLeftCongruenceClasses

▷ `OnLeftCongruenceClasses(class, elm)` (operation)

Returns: A left congruence class.

If *class* is an equivalence class of the left semigroup congruence `cong` on the semigroup `S`, and *elm* is an element of `S`, then this operation returns the equivalence class of `cong` containing the element `elm * x`, where `x` is any element of *class*. The result is well-defined by the definition of a left congruence.

See `IsLeftSemigroupCongruence` (16.1.2) and `IsLeftCongruenceClass` (16.3.2).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := LeftSemigroupCongruence(S, [pair1, pair2]);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
gap> x := Transformation([3, 4, 3, 4, 3]);;
gap> class := LeftCongruenceClassOfElement(cong, x);
<left congruence class of Transformation( [ 3, 4, 3, 4, 3 ] )>
gap> elm := Transformation([1, 2, 2, 1, 2]);;
gap> OnLeftCongruenceClasses(class, elm);
<left congruence class of Transformation( [ 3, 4, 4, 3, 4 ] )>
```

16.3.14 OnRightCongruenceClasses

▷ `OnRightCongruenceClasses(class, elm)` (operation)

Returns: A right congruence class.

If *class* is an equivalence class of the right semigroup congruence `cong` on the semigroup `S`, and *elm* is an element of `S`, then this operation returns the equivalence class of `cong` containing the element `x * elm`, where `x` is any element of *class*. The result is well-defined by the definition of a right congruence.

See `IsRightSemigroupCongruence` (16.1.3) and `IsRightCongruenceClass` (16.3.3).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := RightSemigroupCongruence(S, [pair1, pair2]);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
gap> x := Transformation([3, 4, 3, 4, 3]);;
gap> class := RightCongruenceClassOfElement(cong, x);
<right congruence class of Transformation( [ 3, 4, 3, 4, 3 ] )>
gap> elm := Transformation([1, 2, 2, 1, 2]);;
gap> OnRightCongruenceClasses(class, elm);
<right congruence class of Transformation( [ 2, 1, 2, 1, 2 ] )>
```

16.4 Finding the congruences of a semigroup

16.4.1 CongruencesOfSemigroup (for a semigroup)

▷ `CongruencesOfSemigroup(S)` (attribute)
▷ `LeftCongruencesOfSemigroup(S)` (attribute)
▷ `RightCongruencesOfSemigroup(S)` (attribute)
▷ `CongruencesOfSemigroup(S, restriction)` (operation)
▷ `LeftCongruencesOfSemigroup(S, restriction)` (operation)
▷ `RightCongruencesOfSemigroup(S, restriction)` (operation)

Returns: The congruences of a semigroup.

This attribute gives a list of the left, right, or 2-sided congruences of the semigroup *S*.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See also `LatticeOfCongruences` (16.4.5).

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
>                                 [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> congs := CongruencesOfSemigroup(S);;
```

```
gap> Length(congs);
4
gap> Set(congs, NrCongruenceClasses);
[ 1, 5, 9, 25 ]
gap> pos := Position(congs, UniversalSemigroupCongruence(S));;
gap> congs[pos];
<universal semigroup congruence over
<Rees 0-matrix semigroup 2x2 over Sym( [ 1 .. 3 ] )>>
```

16.4.2 MinimalCongruencesOfSemigroup (for a semigroup)

▷ `MinimalCongruencesOfSemigroup(S)` (attribute)
▷ `MinimalLeftCongruencesOfSemigroup(S)` (attribute)
▷ `MinimalRightCongruencesOfSemigroup(S)` (attribute)
▷ `MinimalCongruencesOfSemigroup(S, restriction)` (operation)
▷ `MinimalLeftCongruencesOfSemigroup(S, restriction)` (operation)
▷ `MinimalRightCongruencesOfSemigroup(S, restriction)` (operation)

Returns: The congruences of a semigroup.

If *S* is a semigroup, then the attribute `MinimalCongruencesOfSemigroup` gives a list of all the congruences on *S* which are *minimal*. A congruence is minimal iff it is non-trivial and contains no other congruences as subrelations (apart from the trivial congruence).

`MinimalLeftCongruencesOfSemigroup` and `MinimalRightCongruencesOfSemigroup` do the same thing, but for left congruences and right congruences respectively. Note that any congruence is also a left congruence, but that a minimal congruence may not be a minimal left congruence.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See also `CongruencesOfSemigroup` (16.4.1) and `PrincipalCongruencesOfSemigroup` (16.4.3).

Example
```
gap> S := Semigroup(Transformation([1, 3, 2]),
>                   Transformation([3, 1, 3]));;
gap> min := MinimalCongruencesOfSemigroup(S);
[ <semigroup congruence over <transformation semigroup of size 13,
     degree 3 with 2 generators> with 1 generating pairs> ]
gap> minl := MinimalLeftCongruencesOfSemigroup(S);
[ <left semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <left semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <left semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>
 ]
```

16.4.3 PrincipalCongruencesOfSemigroup (for a semigroup)

▷ `PrincipalCongruencesOfSemigroup(S)` (attribute)
▷ `PrincipalLeftCongruencesOfSemigroup(S)` (attribute)
▷ `PrincipalRightCongruencesOfSemigroup(S)` (attribute)

▷ `PrincipalCongruencesOfSemigroup(S, restriction)` (operation)
▷ `PrincipalLeftCongruencesOfSemigroup(S, restriction)` (operation)
▷ `PrincipalRightCongruencesOfSemigroup(S, restriction)` (operation)

Returns: A list.

If *S* is a semigroup, then the attribute `PrincipalCongruencesOfSemigroup` gives a list of all the congruences on *S* which are *principal*. A congruence is principal if and only if it is non-trivial and can be defined by a single generating pair.

`PrincipalLeftCongruencesOfSemigroup` and `PrincipalRightCongruencesOfSemigroup` do the same thing, but for left congruences and right congruences respectively. Note that any congruence is a left congruence and a right congruence, but that a principal congruence may not be a principal left congruence or a principal right congruence.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See also `CongruencesOfSemigroup` (16.4.1) and `MinimalCongruencesOfSemigroup` (16.4.2).

Example
```
gap> S := Semigroup(Transformation([1, 3, 2]),
>                   Transformation([3, 1, 3]));;
gap> congs := PrincipalCongruencesOfSemigroup(S);
[ <semigroup congruence over <transformation semigroup of size 13,
     degree 3 with 2 generators> with 1 generating pairs>,
  <semigroup congruence over <transformation semigroup of size 13,
     degree 3 with 2 generators> with 1 generating pairs>,
  <semigroup congruence over <transformation semigroup of size 13,
     degree 3 with 2 generators> with 1 generating pairs>,
  <semigroup congruence over <transformation semigroup of size 13,
     degree 3 with 2 generators> with 1 generating pairs>,
  <semigroup congruence over <transformation semigroup of size 13,
     degree 3 with 2 generators> with 1 generating pairs> ]
```

16.4.4 IsCongruencePoset

▷ `IsCongruencePoset(poset)` (Category)

Returns: `true` or `false`.

This category contains all congruence posets. A *congruence poset* is a partially ordered set of congruences over a specific semigroup, where the ordering is defined by containment according to `IsSubrelation` (16.5.1): given two congruences `cong1` and `cong2`, we say that `cong1` < `cong2` if and only if `cong1` is a subrelation (a refinement) of `cong2`. The congruences in a congruence poset can be left, right, or two-sided.

A congruence poset is displayed as a list of lists, which describes the partial order of its congruences: the integer `j` appears in list `i` if and only if the congruence numbered `j` is a subrelation of the congruence numbered `i`. The list of congruences can be obtained using `CongruencesOfPoset` (16.4.7). Congruence posets can be created using `PosetOfCongruences` (16.4.9), `JoinSemilatticeOfCongruences` (16.4.10), and `LatticeOfCongruences` (16.4.5).

Example
```
gap> S := SymmetricInverseMonoid(2);;
gap> poset := LatticeOfCongruences(S);
[ [  ], [ 1 ], [ 1, 2, 4 ], [ 1, 2 ] ]
```

```
gap> IsCongruencePoset(poset);
true
gap> poset[3];
[ 1, 2, 4 ]
gap> T := FullTransformationMonoid(3);;
gap> congs := PrincipalCongruencesOfSemigroup(T);;
gap> poset := JoinSemilatticeOfCongruences(congs,
>                                            JoinSemigroupCongruences);
[ [ 4, 6 ], [ 1, 3, 4, 5, 6 ], [ 1, 4, 5, 6 ], [ 6 ], [ 1, 4, 6 ],
  [  ] ]
gap> IsCongruencePoset(poset);
true
gap> Length(poset);
6
```

16.4.5 LatticeOfCongruences (for a semigroup)

▷ `LatticeOfCongruences(S)` (attribute)
▷ `LatticeOfLeftCongruences(S)` (attribute)
▷ `LatticeOfRightCongruences(S)` (attribute)
▷ `LatticeOfCongruences(S, restriction)` (operation)
▷ `LatticeOfLeftCongruences(S, restriction)` (operation)
▷ `LatticeOfRightCongruences(S, restriction)` (operation)

Returns: A list of lists.

If S is a semigroup, then `LatticeOfCongruences` gives a list of lists showing how the congruences of S are contained in each other. The congruence numbered i is a subcongruence of the congruence numbered j if and only if i is in the jth list. The numbering is according to the order in `CongruencesOfPoset` (16.4.7).

`LatticeOfLeftCongruences` and `LatticeOfRightCongruences` do the same thing for left and right congruences respectively.

If *restriction* is specified and is a collection of elements from S, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See `CongruencesOfSemigroup` (16.4.1).

Example

```
gap> S := OrderEndomorphisms(2);;
gap> LatticeOfCongruences(S);
[ [  ], [ 1, 3 ], [ 1 ] ]
gap> LatticeOfLeftCongruences(S);
[ [  ], [ 1, 3 ], [ 1 ] ]
gap> LatticeOfRightCongruences(S);
[ [  ], [ 1 ], [ 1 ], [ 1 ], [ 1, 2, 3, 4 ] ]
gap> S := FullTransformationMonoid(4);;
gap> restriction := [Transformation([1, 1, 1, 1]),
>                    Transformation([1, 1, 1, 2]),
>                    Transformation([1, 1, 1, 3])];;
gap> latt := LatticeOfCongruences(S, restriction);
[ [  ], [ 1 ] ]
```

16.4.6 PosetOfPrincipalCongruences (for a semigroup)

▷ `PosetOfPrincipalCongruences(S)` (attribute)
▷ `PosetOfPrincipalLeftCongruences(S)` (attribute)
▷ `PosetOfPrincipalRightCongruences(S)` (attribute)
▷ `PosetOfPrincipalCongruences(S, restriction)` (operation)
▷ `PosetOfPrincipalLeftCongruences(S, restriction)` (operation)
▷ `PosetOfPrincipalRightCongruences(S, restriction)` (operation)

Returns: A congruence poset.

If *S* is a semigroup, then `PosetOfPrincipalCongruences` returns a congruence poset object which contains all the principal congruences of *S*, ordered by containment according to `IsSubrelation` (16.5.1). A congruence is *principal* if it can be defined by a single generating pair. `PosetOfPrincipalLeftCongruences` and `PosetOfPrincipalRightCongruences` do the same thing for left and right congruences respectively.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include principal congruences generated by pairs of elements from *restriction*. Otherwise, all principal congruences will be calculated.

See also `LatticeOfCongruences` (16.4.5) and `PrincipalCongruencesOfSemigroup` (16.4.3).

Example
```
gap> S := Semigroup([Transformation([1, 3, 1]),
>                    Transformation([2, 3, 3])]);;
gap> PosetOfPrincipalLeftCongruences(S);
[ [ 8, 11 ], [  ], [ 1, 2, 8, 11, 12 ], [ 2, 7, 10, 11, 12 ], [ 2 ],
  [ 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12 ], [ 10, 12 ], [ 11 ],
  [ 2, 11, 12 ], [ 12 ], [  ], [  ] ]
gap> PosetOfPrincipalCongruences(S);
[ [ 2, 3 ], [  ], [ 2 ] ]
gap> restriction := [Transformation([3, 2, 3]),
>                    Transformation([3, 1, 3]),
>                    Transformation([2, 2, 2])];;
gap> poset := PosetOfPrincipalRightCongruences(S, restriction);
[ [ 2, 3 ], [  ], [  ] ]
```

16.4.7 CongruencesOfPoset

▷ `CongruencesOfPoset(poset)` (attribute)

Returns: A list.

If *poset* is a congruence poset object, then this attribute returns a list of all the congruence objects in the poset (these may be left, right, or two-sided). The order of this list corresponds to the order of the entries in the poset.

See also `LatticeOfCongruences` (16.4.5) and `CongruencesOfSemigroup` (16.4.1).

Example
```
gap> S := OrderEndomorphisms(2);;
gap> latt := LatticeOfRightCongruences(S);
[ [  ], [ 1 ], [ 1 ], [ 1 ], [ 1, 2, 3, 4 ] ]
gap> CongruencesOfPoset(latt);
[ <right semigroup congruence over <regular transformation monoid
     of size 3, degree 2 with 2 generators> with 0 generating pairs>,
  <right semigroup congruence over <regular transformation monoid
```

```
    of size 3, degree 2 with 2 generators> with 1 generating pairs>,
 <right semigroup congruence over <regular transformation monoid
    of size 3, degree 2 with 2 generators> with 1 generating pairs>,
 <right semigroup congruence over <regular transformation monoid
    of size 3, degree 2 with 2 generators> with 1 generating pairs>,
 <right semigroup congruence over <regular transformation monoid
    of size 3, degree 2 with 2 generators> with 2 generating pairs> ]
```

16.4.8 UnderlyingSemigroupOfCongruencePoset

▷ `UnderlyingSemigroupOfCongruencePoset(`*poset*`)` (attribute)

Returns: A semigroup.

If *poset* is a congruence poset object, then this attribute returns the semigroup on which all its congruences are defined.

Example

```
gap> S := OrderEndomorphisms(2);
<regular transformation monoid of degree 2 with 2 generators>
gap> latt := LatticeOfRightCongruences(S);
[ [  ], [ 1 ], [ 1 ], [ 1 ], [ 1, 2, 3, 4 ] ]
gap> UnderlyingSemigroupOfCongruencePoset(latt) = S;
true
```

16.4.9 PosetOfCongruences

▷ `PosetOfCongruences(`*coll*`)` (operation)

Returns: A congruence poset.

If *coll* is a list or collection of semigroup congruences (which may be left, right, or two-sided) then this operation returns the congruence poset formed by these congruences partially ordered by containment.

This operation does not create any new congruences or take any joins. However, see `JoinSemilatticeOfCongruences` (16.4.10). See also `IsCongruencePoset` (16.4.4) and `LatticeOfCongruences` (16.4.5).

Example

```
gap> S := OrderEndomorphisms(2);;
gap> pair1 := [Transformation([1, 1]), IdentityTransformation];;
gap> pair2 := [IdentityTransformation, Transformation([2, 2])];;
gap> coll := [RightSemigroupCongruence(S, pair1),
>             RightSemigroupCongruence(S, pair2),
>             RightSemigroupCongruence(S, [])];;
gap> PosetOfCongruences(coll);
[ [ 3 ], [ 3 ], [  ] ]
```

16.4.10 JoinSemilatticeOfCongruences (for a list or collection and a function)

▷ `JoinSemilatticeOfCongruences(`*coll*`, `*join_func*`)` (operation)

▷ `JoinSemilatticeOfCongruences(`*poset*`, `*join_func*`)` (operation)

Returns: A congruence poset.

If *coll* is a list or collection of semigroup congruences (which may be left, right, or two-sided) and *join_func* is a function for taking the join of two of the congruences (such as

JoinSemigroupCongruences (16.5.4)) then this operation returns the congruence poset formed by these congruences partially ordered by containment, along with all their joins.

Alternatively, a congruence poset *poset* can be specified; in this case, the congruences contained in *poset* will be used in place of *coll*, and information already known about their containments will be used.

See also IsCongruencePoset (16.4.4) and PosetOfCongruences (16.4.9).

Example
```
gap> S := SymmetricInverseMonoid(2);;
gap> pair1 := [PartialPerm([1], [1]), PartialPerm([2], [1])];;
gap> pair2 := [PartialPerm([1], [1]), PartialPerm([1, 2], [1, 2])];;
gap> pair3 := [PartialPerm([1, 2], [1, 2]),
>             PartialPerm([1, 2], [2, 1])];;
gap> coll := [RightSemigroupCongruence(S, pair1),
>            RightSemigroupCongruence(S, pair2),
>            RightSemigroupCongruence(S, pair3)];;
gap> JoinSemilatticeOfCongruences(coll, JoinRightSemigroupCongruences);
[ [  ], [  ], [ 1 ], [ 1, 2, 3 ] ]
```

16.4.11 MinimalCongruences (for a list or collection)

▷ MinimalCongruences(*coll*) (attribute)

▷ MinimalCongruences(*poset*) (attribute)

Returns: A list.

If *coll* is a list or collection of semigroup congruences (which may be left, right, or two-sided) then this attribute returns a list of all the congruences from *coll* which do not contain any of the others as subrelations.

Alternatively, a congruence poset *poset* can be specified; in this case, the congruences contained in *poset* will be used in place of *coll*, and information already known about their containments will be used.

This function should not be confused with MinimalCongruencesOfSemigroup (16.4.2). See also IsCongruencePoset (16.4.4) and PosetOfCongruences (16.4.9).

Example
```
gap> S := SymmetricInverseMonoid(2);;
gap> pair1 := [PartialPerm([1], [1]), PartialPerm([2], [1])];;
gap> pair2 := [PartialPerm([1], [1]), PartialPerm([1, 2], [1, 2])];;
gap> pair3 := [PartialPerm([1, 2], [1, 2]),
>             PartialPerm([1, 2], [2, 1])];;
gap> coll := [RightSemigroupCongruence(S, pair1),
>            RightSemigroupCongruence(S, pair2),
>            RightSemigroupCongruence(S, pair3)];;
gap> MinimalCongruences(coll);
[ <right semigroup congruence over <symmetric inverse monoid of degree\
 2> with 1 generating pairs>,
  <right semigroup congruence over <symmetric inverse monoid of degree\
 2> with 1 generating pairs> ]
gap> poset := LatticeOfCongruences(S);
[ [  ], [ 1 ], [ 1, 2, 4 ], [ 1, 2 ] ]
gap> MinimalCongruences(poset);
[ <semigroup congruence over <symmetric inverse monoid of degree 2> wi\
th 0 generating pairs> ]
```

16.5 Comparing congruences

16.5.1 IsSubrelation

▷ IsSubrelation(*cong1, cong2*) (operation)

Returns: True or false.

If *cong1* and *cong2* are congruences over the same semigroup, then this operation returns whether *cong2* is a refinement of *cong1*, i.e. whether every pair in *cong2* is contained in *cong1*.

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                   RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> IsSubrelation(cong1, cong2);
true
gap> IsSubrelation(cong2, cong1);
false
```

16.5.2 IsSuperrelation

▷ IsSuperrelation(*cong1, cong2*) (operation)

Returns: True or false.

If *cong1* and *cong2* are congruences over the same semigroup, then this operation returns whether *cong1* is a refinement of *cong2*, i.e. whether every pair in *cong1* is contained in *cong2*.

See IsSubrelation (16.5.1).

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                   RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> IsSuperrelation(cong1, cong2);
false
gap> IsSuperrelation(cong2, cong1);
true
```

16.5.3 MeetSemigroupCongruences

▷ MeetSemigroupCongruences(*c1, c2*) (operation)

Returns: A semigroup congruence.

This operation returns the *meet* of the two semigroup congruences *c1* and *c2* – that is, the largest semigroup congruence contained in both *c1* and *c2*.

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                   RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> MeetSemigroupCongruences(cong1, cong2);
```

```
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (1,2,2)>
```

16.5.4 JoinSemigroupCongruences

▷ JoinSemigroupCongruences(*c1*, *c2*) (operation)
▷ JoinLeftSemigroupCongruences(*c1*, *c2*) (operation)
▷ JoinRightSemigroupCongruences(*c1*, *c2*) (operation)

Returns: A semigroup congruence.

This operation returns the *join* of the two semigroup congruences `c1` and `c2` – that is, the smallest semigroup congruence containing all the relations in both `c1` and `c2`.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                       RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> JoinSemigroupCongruences(cong1, cong2);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (3,2,2)>
```

16.6 Congruences on Rees matrix semigroups

This section describes the implementation of congruences of simple and 0-simple semigroups in the Semigroups package, and the functions associated with them. This code and this part of the manual were written by Michael Torpey. Most of the theorems used in this chapter are from Section 3.5 of [How95].

By the Rees Theorem, any 0-simple semigroup S is isomorphic to a *Rees 0-matrix semigroup* (see (**Reference: Rees Matrix Semigroups**)) over a group, with a regular sandwich matrix. That is,

$$S \cong \mathscr{M}^0[G;I,\Lambda;P],$$

where G is a group, Λ and I are non-empty sets, and P is regular in the sense that it has no rows or columns consisting solely of zeroes.

The congruences of a Rees 0-matrix semigroup are in 1-1 correspondence with the *linked triple*, which is a triple of the form `[N, S, T]` where:

- `N` is a normal subgroup of the underlying group `G`,
- `S` is an equivalence relation on the columns of `P`,
- `T` is an equivalence relation on the rows of `P`,

satisfying the following conditions:

- a pair of `S`-related columns must contain zeroes in precisely the same rows,
- a pair of `T`-related rows must contain zeroes in precisely the same columns,

- if i and j are S-related, k and l are T-related and the matrix entries $p_{k,i}, p_{k,j}, p_{l,i}, p_{l,j} \neq 0$, then $q_{k,l,i,j} \in N$, where
$$q_{k,l,i,j} = p_{k,i} p_{l,i}^{-1} p_{l,j} p_{k,j}^{-1}.$$

By Theorem 3.5.9 in [How95], for any finite 0-simple Rees 0-matrix semigroup, there is a bijection between its non-universal congruences and its linked triples. In this way, we can internally represent any congruence of such a semigroup by storing its associated linked triple instead of a set of generating pairs, and thus perform many calculations on it more efficiently.

If a congruence is defined by a linked triple (N, S, T), then a single class of that congruence can be defined by a triple (Nx, i / S, k / S), where Nx is a right coset of N, i / S is the equivalence class of i in S, and k / S is the equivalence class of k in T. Thus we can internally represent any class of such a congruence as a triple simply consisting of a right coset and two positive integers.

An analogous condition exists for finite simple Rees matrix semigroups without zero.

16.6.1 IsRMSCongruenceByLinkedTriple

▷ IsRMSCongruenceByLinkedTriple(*obj*) (category)
▷ IsRZMSCongruenceByLinkedTriple(*obj*) (category)

Returns: true or false.

These categories describe a type of semigroup congruence over a Rees matrix or 0-matrix semigroup. Externally, an object of this type may be used in the same way as any other object in the category IsSemigroupCongruence (**Reference: IsSemigroupCongruence**) but it is represented internally by its *linked triple*, and certain functions may take advantage of this information to reduce computation times.

An object of this type may be constructed with RMSCongruenceByLinkedTriple or RZMSCongruenceByLinkedTriple, or this representation may be selected automatically by SemigroupCongruence (16.2.1).

Example
```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);;
gap> IsRZMSCongruenceByLinkedTriple(cong);
true
```

16.6.2 RMSCongruenceByLinkedTriple

▷ RMSCongruenceByLinkedTriple(*S*, *N*, *colBlocks*, *rowBlocks*) (function)
▷ RZMSCongruenceByLinkedTriple(*S*, *N*, *colBlocks*, *rowBlocks*) (function)

Returns: A Rees matrix or 0-matrix semigroup congruence by linked triple.

This function returns a semigroup congruence over the Rees matrix or 0-matrix semigroup *S* corresponding to the linked triple (*N*, *colBlocks*, *rowBlocks*). The argument *N* should be a normal subgroup of the underlying semigroup of *S*; *colBlocks* should be a partition of the columns of the matrix of S; and *rowBlocks* should be a partition of the rows of the matrix of S. For example, if the matrix has 5 rows, then a possibility for *rowBlocks* might be `[[1, 3], [2, 5], [4]]`.

If the arguments describe a valid linked triple on S, then an object in the category `IsRZMSCongruenceByLinkedTriple` is returned. This object can be used like any other semigroup congruence in GAP.

If the arguments describe a triple which is not *linked* in the sense described above, then this function returns an error.

Example
```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);
<semigroup congruence over <Rees 0-matrix semigroup 6x3 over
  Group([ (1,4,5), (1,5,3,4) ])> with linked triple (2^2,4,3)>
```

16.6.3 IsRMSCongruenceClassByLinkedTriple

▷ `IsRMSCongruenceClassByLinkedTriple(obj)` (category)

▷ `IsRZMSCongruenceClassByLinkedTriple(obj)` (category)

Returns: `true` or `false`.

These categories contain the congruence classes of a semigroup congruence of the categories `IsRMSCongruenceByLinkedTriple` (16.6.1) and `IsRZMSCongruenceByLinkedTriple` (16.6.1) respectively.

An object of one of these types may be used in the same way as any other object in the category `IsCongruenceClass` (16.3.1), but the class is represented internally by information related to the congruence's *linked triple*, and certain functions may take advantage of this information to reduce computation times.

Example
```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);;
gap> classes := CongruenceClasses(cong);;
gap> IsRZMSCongruenceClassByLinkedTriple(classes[1]);
true
```

16.6.4 RMSCongruenceClassByLinkedTriple

▷ `RMSCongruenceClassByLinkedTriple(cong, nCoset, colClass, rowClass)` (operation)

▷ `RZMSCongruenceClassByLinkedTriple(cong, nCoset, colClass, rowClass)` (operation)

Returns: A Rees matrix or 0-matrix semigroup congruence class by linked triple.

This operation returns one congruence class of the congruence *cong*, as defined by the other three parameters.

The argument *cong* must be a Rees matrix or 0-matrix semigroup congruence by linked triple. If the linked triple consists of the three parameters `N`, `colBlocks` and `rowBlocks`, then *nCoset* must be a right coset of `N`, *colClass* must be a positive integer corresponding to a position in the list `colBlocks`, and *rowClass* must be a positive integer corresponding to a position in the list `rowBlocks`.

If the arguments are valid, an `IsRMSCongruenceClassByLinkedTriple` or `IsRZMSCongruenceClassByLinkedTriple` object is returned, which can be used like any other equivalence class in GAP. Otherwise, an error is returned.

Example
```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);;
gap> class := RZMSCongruenceClassByLinkedTriple(cong,
> RightCoset(N, (1, 5)), 2, 3);
<congruence class of (2,(3,4),3)>
```

16.6.5 IsLinkedTriple

▷ `IsLinkedTriple(S, N, colBlocks, rowBlocks)` (operation)

Returns: `true` or `false`.

This operation returns true if and only if the arguments (*N*, *colBlocks*, *rowBlocks*) describe a linked triple of the Rees matrix or 0-matrix semigroup *S*, as described above.

Example
```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> IsLinkedTriple(S, N, colBlocks, rowBlocks);
true
```

16.6.6 CanonicalRepresentative

▷ `CanonicalRepresentative(class)` (attribute)

Returns: A congruence class.

This attribute gives a canonical representative for the semigroup congruence class *class*. This representative can be used to identify a class uniquely.

At present this only works for simple and 0-simple semigroups.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong := CongruencesOfSemigroup(S)[3];;
gap> class := CongruenceClasses(cong)[3];;
gap> CanonicalRepresentative(class);
(1,(1,2,3),2)
```

16.6.7 AsSemigroupCongruenceByGeneratingPairs

▷ AsSemigroupCongruenceByGeneratingPairs(*cong*) (operation)

Returns: A semigroup congruence.

This operation takes *cong*, a semigroup congruence, and returns the same congruence relation, but described by GAP's default method of defining semigroup congruences: a set of generating pairs for the congruence.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
>                                 [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong := CongruencesOfSemigroup(S)[3];;
gap> AsSemigroupCongruenceByGeneratingPairs(cong);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with 3 generating pairs>
```

16.6.8 AsRMSCongruenceByLinkedTriple

▷ AsRMSCongruenceByLinkedTriple(*cong*) (operation)

▷ AsRZMSCongruenceByLinkedTriple(*cong*) (operation)

Returns: A Rees matrix or 0-matrix semigroup congruence by linked triple.

This operation takes a semigroup congruence *cong* over a finite simple or 0-simple Rees 0-matrix semigroup, and returns that congruence relation in a new form: as either a congruence by linked triple, or a universal congruence.

If the congruence is not defined over an appropriate type of semigroup, then this function returns an error.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> x := ReesZeroMatrixSemigroupElement(S, 1, (1, 3, 2), 1);;
gap> y := ReesZeroMatrixSemigroupElement(S, 1, (), 1);;
gap> cong := SemigroupCongruenceByGeneratingPairs(S, [[x, y]]);;
gap> AsRZMSCongruenceByLinkedTriple(cong);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (3,2,2)>
```

16.7 Congruences on inverse semigroups

This section describes the implementation of congruences of inverse semigroups in the Semigroups package, and the functions associated with them. This code and this part of the manual were written by Michael Torpey. Most of the theorems used in this chapter are from Section 5.3 of [How95].

The congruences of an inverse semigroup are in 1-1 correspondence with its *congruence pairs*. A congruence pair is a pair `(N, t)` such that:

- `N` is a normal subsemigroup of `S` – that is, a self-conjugate subsemigroup which contains all the idempotents of `S`,
- `t` is a normal congruence on `E`, the subsemigroup of all idempotents in `S` – that is, a congruence on `E` such that if (e, f) is a pair in `t`, then the pair $(a^{-1}ea, a^{-1}fa)$ is also in `t`,

satisfying the following conditions:

- If $ae \in N$ and $(e, a^{-1}a) \in t$, then $a \in N$,
- If $a \in N$, then $(aa^{-1}, a^{-1}a) \in t$.

By Theorem 5.3.3 in [How95], for any inverse semigroup, there is a bijection between its congruences and its congruence pairs. In this way, we can internally represent any congruence of such a semigroup by storing its associated congruence pair instead of a set of generating pairs, and thus perform many calculations on it more efficiently.

If we have a congruence `C` with congruence pair `(N, t)`, it turns out that `N` is its *kernel* (that is, the set of all elements congruent to an idempotent) and that `t` is its *trace* (that is, the restriction of `C` to the idempotents). Hence, we refer to a congruence stored in this format as a "congruence by kernel and trace".

See `cong_by_ker_trace_threshold` in Section 6.3 for details on when this method is used.

16.7.1 IsInverseSemigroupCongruenceByKernelTrace

▷ `IsInverseSemigroupCongruenceByKernelTrace(`*cong*`)` (Category)

Returns: `true` or `false`.

This category contains any inverse semigroup congruence *cong* which is represented internally by its kernel and trace. The `SemigroupCongruence` (16.2.1) function may create an object of this category if its first argument *S* is an inverse semigroup and has sufficiently large size. It can be treated like any other semigroup congruence object.

See [How95] Section 5.3 for more details. See also `InverseSemigroupCongruenceByKernelTrace` (16.7.2).

Example
```
gap> S := InverseSemigroup([
>  PartialPerm([3, 6, 7, 8, 0, 1, 2, 5]),
>  PartialPerm([5, 6, 7, 0, 9, 4, 0, 8]),
>  PartialPerm([5, 1, 8, 6, 9, 4, 0, 7])]);;
gap> cong := SemigroupCongruence(S, []);
<semigroup congruence over <inverse partial perm semigroup
 of size 117859, rank 9 with 3 generators> with congruence pair (228,
228)>
gap> IsInverseSemigroupCongruenceByKernelTrace(cong);
true
```

16.7.2 InverseSemigroupCongruenceByKernelTrace

▷ `InverseSemigroupCongruenceByKernelTrace(S, kernel, traceBlocks)` (function)

Returns: An inverse semigroup congruence by kernel and trace.

If *S* is an inverse semigroup, *kernel* is a subsemigroup of *S*, *traceBlocks* is a list of lists describing a congruence on the idempotents of *S*, and (*kernel*, *trace*) describes a valid congruence pair for *S* (see [How95] Section 5.3) then this function returns the semigroup congruence defined by that congruence pair.

See also `KernelOfSemigroupCongruence` (16.7.4) and `TraceOfSemigroupCongruence` (16.7.5).

Example

```
gap> S := InverseSemigroup([
>   PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> kernel := InverseSemigroup([
>   PartialPerm([1, 0, 3]), PartialPerm([0, 2, 3]),
>   PartialPerm([1, 2]), PartialPerm([3]),
>   PartialPerm([2])]);;
gap> trace := [
>  [PartialPerm([0, 2, 3])],
>  [PartialPerm([1, 2])],
>  [PartialPerm([1, 0, 3])],
>  [PartialPerm([0, 0, 3]), PartialPerm([0, 2]),
>   PartialPerm([1]), PartialPerm([], [])]];;
gap> cong := InverseSemigroupCongruenceByKernelTrace(S, kernel, trace);
<semigroup congruence over <inverse partial perm semigroup of rank 3
 with 2 generators> with congruence pair (13,4)>
```

16.7.3 AsInverseSemigroupCongruenceByKernelTrace

▷ `AsInverseSemigroupCongruenceByKernelTrace(cong)` (attribute)

Returns: An inverse semigroup congruence by kernel and trace.

If *cong* is a semigroup congruence over an inverse semigroup, then this attribute returns an object which describes the same congruence, but with an internal representation defined by that congruence's kernel and trace.

See [How95] section 5.3 for more details.

Example

```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> cong := SemigroupCongruenceByGeneratingPairs(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);
<semigroup congruence over <inverse partial perm semigroup of rank 3
 with 2 generators> with 2 generating pairs>
gap> cong2 := AsInverseSemigroupCongruenceByKernelTrace(cong);
<semigroup congruence over <inverse partial perm semigroup of rank 3
 with 2 generators> with congruence pair (19,1)>
```

16.7.4 KernelOfSemigroupCongruence

▷ `KernelOfSemigroupCongruence(cong)` (attribute)

Returns: An inverse semigroup.

If *cong* is a congruence over a semigroup with inverse op, then this attribute returns the *kernel* of that congruence; that is, the inverse subsemigroup consisting of all elements which are related to an idempotent by *cong*.

Example
```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> cong := SemigroupCongruence(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);
<semigroup congruence over <inverse partial perm semigroup
 of size 19, rank 3 with 2 generators> with 2 generating pairs>
gap> KernelOfSemigroupCongruence(cong);
<inverse partial perm semigroup of rank 3 with 5 generators>
```

16.7.5 TraceOfSemigroupCongruence

▷ `TraceOfSemigroupCongruence(cong)` (attribute)

Returns: A list of lists.

If *cong* is an inverse semigroup congruence by kernel and trace, then this attribute returns the restriction of *cong* to the idempotents of the semigroup. This is in block form: each idempotent will appear in precisely one list, and two idempotents will be in the same list if and only if they are related by *cong*.

Example
```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> cong := SemigroupCongruence(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);
<semigroup congruence over <inverse partial perm semigroup
 of size 19, rank 3 with 2 generators> with 2 generating pairs>
gap> TraceOfSemigroupCongruence(cong);
[ [ <empty partial perm>, <identity partial perm on [ 1 ]>,
      <identity partial perm on [ 2 ]>,
      <identity partial perm on [ 1, 2 ]>,
      <identity partial perm on [ 3 ]>,
      <identity partial perm on [ 2, 3 ]>,
      <identity partial perm on [ 1, 3 ]> ] ]
```

16.7.6 IsInverseSemigroupCongruenceClassByKernelTrace

▷ `IsInverseSemigroupCongruenceClassByKernelTrace(obj)` (Category)

Returns: `true` or `false`.

This category contains any congruence class which belongs to a congruence which is represented internally by its kernel and trace. See `InverseSemigroupCongruenceByKernelTrace` (16.7.2).

See [How95] Section 5.3 for more details.

Example
```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])],
> rec(cong_by_ker_trace_threshold := 0));;
gap> cong := SemigroupCongruence(I,
```

```
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);;
gap> class := CongruenceClassOfElement(cong,
>                                      PartialPerm([1, 2], [2, 3]));;
gap> IsInverseSemigroupCongruenceClassByKernelTrace(class);
true
```

16.7.7 MinimumGroupCongruence

▷ `MinimumGroupCongruence(S)` (attribute)

Returns: An inverse semigroup congruence by kernel and trace.

If *S* is an inverse semigroup, then this function returns the least congruence on *S* whose quotient is a group.

Example
```
gap> S := InverseSemigroup([
>   PartialPerm([5, 2, 0, 0, 1, 4]),
>   PartialPerm([1, 4, 6, 3, 5, 0, 2])]);;
gap> cong := MinimumGroupCongruence(S);
<semigroup congruence over <inverse partial perm semigroup of rank 7
 with 2 generators> with congruence pair (59,1)>
gap> IsGroupAsSemigroup(S / cong);
true
```

16.8 Rees congruences

A Rees congruence is defined by a semigroup ideal. It is a congruence on a semigroup `S` which has one congruence class equal to a semigroup ideal `I` of `S`, and every other congruence class being a singleton.

16.8.1 SemigroupIdealOfReesCongruence

▷ `SemigroupIdealOfReesCongruence(cong)` (attribute)

Returns: A semigroup ideal.

If *cong* is a rees congruence (see `IsReesCongruence` (**Reference: IsReesCongruence**)) then this attribute returns the two-sided ideal that was used to define it, i.e.~the ideal of elements in the only non-trivial congruence class of *cong*.

Example
```
gap> S := Semigroup([
> Transformation([2, 3, 4, 3, 1, 1]),
> Transformation([6, 4, 4, 4, 6, 1])]);;
gap> I := SemigroupIdeal(S,
> Transformation([4, 4, 4, 4, 4, 2]),
> Transformation([3, 3, 3, 3, 3, 2]));;
gap> cong := ReesCongruenceOfSemigroupIdeal(I);;
gap> SemigroupIdealOfReesCongruence(cong);
<non-regular transformation semigroup ideal of degree 6 with
  2 generators>
```

16.8.2 IsReesCongruenceClass

▷ `IsReesCongruenceClass(obj)` (category)

Returns: `true` or `false`.

This category describes a congruence class of a Rees congruence. A congruence class of a Rees congruence either contains all the elements of an ideal, or is a singleton (see `IsReesCongruence` (**Reference: IsReesCongruence**)).

An object of this type may be used in the same way as any other congruence class object.

Example
```
gap> S := Semigroup(
> Transformation([2, 3, 4, 3, 1, 1]),
> Transformation([6, 4, 4, 4, 6, 1]));;
gap> I := SemigroupIdeal(S,
> Transformation([4, 4, 4, 4, 4, 2]),
> Transformation([3, 3, 3, 3, 3, 2]));;
gap> cong := ReesCongruenceOfSemigroupIdeal(I);;
gap> classes := CongruenceClasses(cong);;
gap> IsReesCongruenceClass(classes[1]);
true
```

16.9 Universal congruences

The linked triples of a completely 0-simple Rees 0-matrix semigroup describe only its non-universal congruences. In any one of these, the zero element of the semigroup is related only to itself. However, for any semigroup S the universal relation $S \times S$ is a congruence; called the *universal congruence*. The universal congruence on a semigroup has its own unique representation.

Since many things we want to calculate about congruences are trivial in the case of the universal congruence, this package contains a category specifically designed for it, `IsUniversalSemigroupCongruence`. We also define `IsUniversalSemigroupCongruenceClass`, which represents the single congruence class of the universal congruence.

16.9.1 IsUniversalSemigroupCongruence

▷ `IsUniversalSemigroupCongruence(obj)` (property)

Returns: `true` or `false`.

This property describes a type of semigroup congruence, which must refer to the *universal semigroup congruence* $S \times S$. Externally, an object of this type may be used in the same way as any other object in the category `IsSemigroupCongruence` (**Reference: IsSemigroupCongruence**).

An object of this type may be constructed with `UniversalSemigroupCongruence` or this representation may be selected automatically as an alternative to an `IsRZMSCongruenceByLinkedTriple` object (since the universal congruence cannot be represented by a linked triple).

Example
```
gap> S := Semigroup([Transformation([3, 2, 3])]);;
gap> U := UniversalSemigroupCongruence(S);;
gap> IsUniversalSemigroupCongruence(U);
true
```

16.9.2 IsUniversalSemigroupCongruenceClass

▷ `IsUniversalSemigroupCongruenceClass(obj)` (category)

Returns: `true` or `false`.

This category describes a class of the universal semigroup congruence (see `IsUniversalSemigroupCongruence` (16.9.1)). A universal semigroup congruence by definition has precisely one congruence class, which contains all of the elements of the semigroup in question.

Example

```
gap> S := Semigroup([Transformation([3, 2, 3])]);;
gap> U := UniversalSemigroupCongruence(S);;
gap> classes := CongruenceClasses(U);;
gap> IsUniversalSemigroupCongruenceClass(classes[1]);
true
```

16.9.3 UniversalSemigroupCongruence

▷ `UniversalSemigroupCongruence(S)` (operation)

Returns: A universal semigroup congruence.

This operation returns the universal semigroup congruence for the semigroup `S`. It can be used in the same way as any other semigroup congruence object.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> UniversalSemigroupCongruence(S);
<universal semigroup congruence over
<Rees 0-matrix semigroup 2x2 over Sym( [ 1 .. 3 ] )>>
```

Chapter 17

Semigroup homomorphisms

In this chapter we describe the various ways to define a homomorphism from a semigroup to another semigroup.

17.1 Isomorphisms of arbitrary semigroups

17.1.1 IsIsomorphicSemigroup

▷ `IsIsomorphicSemigroup(S, T)` (operation)

Returns: `true` or `false`.

If S and T are semigroups, then this operation attempts to determine whether S and T are isomorphic semigroups by using the operation `IsomorphismSemigroups` (17.1.3). If `IsomorphismSemigroups(S, T)` returns an isomorphism, then `IsIsomorphicSemigroup(S, T)` returns `true`, while if `IsomorphismSemigroups(S, T)` returns `fail`, then `IsIsomorphicSemigroup(S, T)` returns `false`. Note that in some cases, at present, there is no method for determining whether S is isomorphic to T, even if it is obvious to the user whether or not S and T are isomorphic. There are plans to improve this in the future.

If the size of S and T is rather small — with approximately 50 or fewer elements — then it is possible to calculate whether S and T are isomorphic by using `SmallestMultiplicationTable` (17.1.2), but this is not currently done by `IsIsomorphicSemigroup`. In particular, S and T are isomorphic if and only if `SmallestMultiplicationTable(S) = SmallestMultiplicationTable(T)`.

Example

```
gap> S := Semigroup(PartialPerm([1, 2, 4], [1, 3, 5]),
>                   PartialPerm([1, 3, 5], [1, 2, 4]));;
gap> T := AsSemigroup(IsTransformationSemigroup, S);;
gap> IsIsomorphicSemigroup(S, T);
true
gap> IsIsomorphicSemigroup(FullTransformationMonoid(4),
> PartitionMonoid(4));
false
```

17.1.2 SmallestMultiplicationTable

▷ `SmallestMultiplicationTable(S)` (attribute)

Returns: The lex-least multiplication table of a semigroup.

This function returns the lex-least multiplication table of a semigroup isomorphic to the semigroup *S*. `SmallestMultiplicationTable` is an isomorphism invariant of semigroups, and so it can, for example, be used to check if two semigroups are isomorphic.

Due to the high complexity of computing the smallest multiplication table of a semigroup, this function only performs well for semigroups with at most approximately 50 elements.

`SmallestMultiplicationTable` is based on the function `IdSmallSemigroup` (**Smallsemi: IdSmallSemigroup**) by Andreas Distler.

Example

```
gap> S := Semigroup(
> Bipartition([[1, 2, 3, -1, -3], [-2]]),
> Bipartition([[1, 2, 3, -1], [-2], [-3]]),
> Bipartition([[1, 2, 3], [-1], [-2, -3]]),
> Bipartition([[1, 2, -1], [3, -2], [-3]]));;
gap> Size(S);
8
gap> SmallestMultiplicationTable(S);
[ [ 1, 1, 3, 4, 5, 6, 7, 8 ], [ 1, 1, 3, 4, 5, 6, 7, 8 ],
  [ 1, 1, 3, 4, 5, 6, 7, 8 ], [ 1, 3, 3, 4, 5, 6, 7, 8 ],
  [ 5, 5, 6, 7, 5, 6, 7, 8 ], [ 5, 5, 6, 7, 5, 6, 7, 8 ],
  [ 5, 6, 6, 7, 5, 6, 7, 8 ], [ 5, 6, 6, 7, 5, 6, 7, 8 ] ]
```

17.1.3 IsomorphismSemigroups

▷ `IsomorphismSemigroups(S, T)` (operation)

Returns: An isomorphism, or `fail`.

This operation attempts to find an isomorphism from the semigroup *S* to the semigroup *T*. If it finds one, then it is returned, and if not, then `fail` is returned.

For many types of semigroup, `IsomorphismSemigroups` is not able to determine whether or not *S* and *T* are isomorphic, and so this operation may result in an "Error, no method found". `IsomorphismSemigroups` may be able deduce that *S* and *T* are not isomorphic by finding that some of their semigroup-theoretic properties differ; however it is harder to construct an isomorphism for semigroups that are isomorphic.

At present, `IsomorphismSemigroups` is only able to return an isomorphism when *S* and *T* are finite simple, 0-simple, or monogenic semigroups, or when *S* = *T*. See `IsSimpleSemigroup` (14.1.21), `IsZeroSimpleSemigroup` (14.1.27), and `IsMonogenicSemigroup` (14.1.10) for more information about these types of semigroups.

Example

```
gap> S := RectangularBand(IsTransformationSemigroup, 4, 5);
<regular transformation semigroup of size 20, degree 9 with 5
 generators>
gap> T := RectangularBand(IsBipartitionSemigroup, 4, 5);
<regular bipartition semigroup of size 20, degree 3 with 5 generators>
gap> IsomorphismSemigroups(S, T) <> fail;
true
gap> D := DClass(FullTransformationMonoid(5),
>                Transformation([1, 2, 3, 4, 1]));;
gap> S := PrincipalFactor(D);;
gap> StructureDescription(UnderlyingSemigroup(S));
"S4"
gap> S;
```

```
<Rees 0-matrix semigroup 10x5 over S4>
gap> D := DClass(PartitionMonoid(5),
> Bipartition([[1], [2, -2], [3, -3], [4, -4], [5, -5], [-1]]));;
gap> T := PrincipalFactor(D);;
gap> StructureDescription(UnderlyingSemigroup(T));
"S4"
gap> T;
<Rees 0-matrix semigroup 15x15 over S4>
gap> IsomorphismSemigroups(S, T);
fail
gap> I := SemigroupIdeal(FullTransformationMonoid(5),
>                        Transformation([1, 1, 2, 3, 4]));;
gap> T := PrincipalFactor(DClass(I, I.1));;
gap> StructureDescription(UnderlyingSemigroup(T));
"S4"
gap> T;
<Rees 0-matrix semigroup 10x5 over S4>
gap> IsomorphismSemigroups(S, T) <> fail;
true
```

17.2 Isomorphisms of Rees (0-)matrix semigroups

An isomorphism between two regular finite Rees (0-)matrix semigroups whose underlying semigroups are groups can be described by a triple defined in terms of the matrices and underlying groups of the semigroups. For a full description of the theory involved, see Section 3.4 of [How95].

An isomorphism described in this way can be constructed using `RMSIsoByTriple` (17.2.2) or `RZMSIsoByTriple` (17.2.2), and will satisfy the filter `IsRMSIsoByTriple` (17.2.1) or `IsRZMSIsoByTriple` (17.2.1).

17.2.1 IsRMSIsoByTriple

▷ `IsRMSIsoByTriple` (Category)
▷ `IsRZMSIsoByTriple` (Category)

The isomorphisms between finite Rees matrix or 0-matrix semigroups `S` and `T` over groups `G` and `H`, respectively, specified by a triple consisting of:

1. an isomorphism of the underlying graph of `S` to the underlying graph of of `T`
2. an isomorphism from `G` to `H`
3. a function from `Rows(S)` union `Columns(S)` to `H`

belong to the categories `IsRMSIsoByTriple` and `IsRZMSIsoByTriple`. Basic operators for such isomorphism are given in 17.2.6, and basic operations are: `Range` (**Reference: range**), `Source` (**Reference: Source**), `ELM_LIST` (17.2.3), `CompositionMapping` (**Reference: CompositionMapping**), `ImagesElm` (17.2.5), `ImagesRepresentative` (17.2.5), `InverseGeneralMapping` (**Reference: InverseGeneralMapping**), `PreImagesRepresentative` (**Reference: PreImagesRepresentative**), `IsOne` (**Reference: IsOne**).

17.2.2 RMSIsoByTriple

▷ `RMSIsoByTriple(R1, R2, triple)` (operation)
▷ `RZMSIsoByTriple(R1, R2, triple)` (operation)

Returns: An isomorphism.

If *R1* and *R2* are isomorphic regular Rees 0-matrix semigroups whose underlying semigroups are groups then `RZMSIsoByTriple` returns the isomorphism between *R1* and *R2* defined by *triple*, which should be a list consisting of the following:

- *triple*`[1]` should be a permutation describing an isomorphism from the graph of *R1* to the graph of *R2*, i.e. it should satisfy `OnDigraphs(RZMSDigraph(R1), triple[1]) = RZMSDigraph(R2)`.
- *triple*`[2]` should be an isomorphism from the underlying group of *R1* to the underlying group of *R2* (see `UnderlyingSemigroup` (**Reference: UnderlyingSemigroup (for a Rees 0-matrix semigroup)**)).
- *triple*`[3]` should be a list of elements from the underlying group of *R2*. If the `Matrix` (**Reference: Matrix**) of *R1* has m columns and n rows, then the list should have length $m+n$, where the first m entries should correspond to the columns of *R1*'s matrix, and the last n entries should correspond to the rows. These column and row entries should correspond to the u_i and v_λ elements in Theorem 3.4.1 of [How95].

If *triple* describes a valid isomorphism from *R1* to *R2* then this will return an object in the category `IsRZMSIsoByTriple` (17.2.1); otherwise an error will be returned.

If *R1* and *R2* are instead Rees matrix semigroups (without zero) then `RMSIsoByTriple` should be used instead. This operation is used in the same way, but it should be noted that since an RMS's graph is a complete bipartite graph, *triple*`[1]` can be any permutation on `[1 .. m + n]`, so long as no point in `[1 .. m]` is mapped to a point in `[m + 1 .. m + n]`.

Example

```
gap> g := SymmetricGroup(3);;
gap> mat := [[0, 0, (1, 3)], [(1, 2, 3), (), (2, 3)], [0, 0, ()]];;
gap> R := ReesZeroMatrixSemigroup(g, mat);;
gap> id := IdentityMapping(g);;
gap> g_elms_list := [(), (), (), (), (), ()];;
gap> RZMSIsoByTriple(R, R, [(), id, g_elms_list]);
((), IdentityMapping( SymmetricGroup( [ 1 .. 3 ] ) ),
[ (), (), (), (), (), () ])
```

17.2.3 ELM_LIST (for IsRMSIsoByTriple)

▷ `ELM_LIST(map, pos)` (operation)

Returns: A component of an isomorphism of Rees (0-)matrix semigroups by triple.

`ELM_LIST(map, i)` returns the `i`th component of the Rees (0-)matrix semigroup isomorphism by triple *map* when `i = 1, 2, 3`.

The components of an isomorphism of Rees (0-)matrix semigroups by triple are:

1. An isomorphism of the underlying graphs of the source and range of *map*, respectively.
2. An isomorphism of the underlying groups of the source and range of *map*, respectively.

3. An function from the union of the rows and columns of the source of *map* to the underlying group of the range of *map*.

17.2.4 CompositionMapping2 (for IsRMSIsoByTriple)

▷ CompositionMapping2(*map1*, *map2*) (operation)
▷ CompositionMapping2(*map1*, *map2*) (operation)

Returns: A Rees (0-)matrix semigroup by triple.

If *map1* and *map2* are isomorphisms of Rees matrix or 0-matrix semigroups specified by triples and the range of *map2* is contained in the source of *map1*, then CompositionMapping2(*map1*, *map2*) returns the isomorphism from Source(*map2*) to Range(*map1*) specified by the triple with components:

1. *map1*[1] * *map2*[1]

2. *map1*[2] * *map2*[2]

3. the componentwise product of *map1*[1] * *map2*[3] and *map1*[3] * *map2*[2].

Example

```
gap> R := ReesZeroMatrixSemigroup(Group([(1, 2, 3, 4)]),
> [[(1, 3)(2, 4), (1, 4, 3, 2), (), (1, 2, 3, 4), (1, 3)(2, 4), 0],
>  [(1, 4, 3, 2), 0, (), (1, 4, 3, 2), (1, 2, 3, 4), (1, 2, 3, 4)],
>  [(), (), (1, 4, 3, 2), (1, 2, 3, 4), 0, (1, 2, 3, 4)],
>  [(1, 2, 3, 4), (1, 4, 3, 2), (1, 2, 3, 4), 0, (), (1, 2, 3, 4)],
>  [(1, 3)(2, 4), (1, 2, 3, 4), 0, (), (1, 4, 3, 2), (1, 2, 3, 4)],
>  [0, (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), ()]]);
<Rees 0-matrix semigroup 6x6 over Group([ (1,2,3,4) ])>
gap> G := AutomorphismGroup(R);
<automorphism group of <Rees 0-matrix semigroup 6x6 over Group([ (1,2,
3,4) ])> with 4 generators>
gap> G.2;
((), IdentityMapping( Group( [ (1,2,3,4) ] ) ),
[ (), (), (), (), (), (), (), (), (), (), (), () ])
gap> G.3;
(( 2, 4, 6, 3)( 7,11, 8,10), GroupHomomorphismByImages( Group(
[ (1,2,3,4) ] ), Group( [ (1,2,3,4) ] ), [ (1,2,3,4) ],
[ (1,2,3,4) ] ), [ (), (1,4,3,2), (1,4,3,2), (), (1,4,3,2),
  (1,3)(2,4), (), (1,3)(2,4), (), (1,2,3,4), (1,2,3,4), (1,4,3,2) ])
gap> CompositionMapping2(G.2, G.3);
(( 2, 4, 6, 3)( 7,11, 8,10), GroupHomomorphismByImages( Group(
[ (1,2,3,4) ] ), Group( [ (1,2,3,4) ] ), [ (1,2,3,4) ],
[ (1,2,3,4) ] ), [ (), (1,4,3,2), (1,4,3,2), (), (1,4,3,2),
  (1,3)(2,4), (), (1,3)(2,4), (), (1,2,3,4), (1,2,3,4), (1,4,3,2) ])
```

17.2.5 ImagesElm (for IsRMSIsoByTriple)

▷ ImagesElm(*map*, *pt*) (operation)
▷ ImagesRepresentative(*map*, *pt*) (operation)

Returns: An element of a Rees (0-)matrix semigroup or a list containing such an element.

If *map* is an isomorphism of Rees matrix or 0-matrix semigroups specified by a triple and *pt* is an element of the source of *map*, then `ImagesRepresentative(map, pt) = pt ^ map` returns the image of *pt* under *map*.

The image of *pt* under *map* of `Range(map)` is the element with components:

1. `pt[1] ^ map[1]`
2. `(pt[1] ^ map[3]) * (pt[2] ^ map[2]) * (pt[3] ^ map[3]) ^ -1`
3. `pt[3] ^ map[1]`.

`ImagesElm(map, pt)` simply returns `[ImagesRepresentative(map, pt)]`.

Example

```
gap> R := ReesZeroMatrixSemigroup(Group([(2, 8), (2, 8, 6)]),
> [[0, (2, 8), 0, 0, 0, (2, 8, 6)],
>  [(), 0, (2, 8, 6), (2, 6), (2, 6, 8), 0],
>  [(2, 8, 6), 0, (2, 6, 8), (2, 8), (), 0],
>  [(2, 8, 6), 0, (2, 6, 8), (2, 8), (), 0],
>  [0, (2, 8, 6), 0, 0, 0, (2, 8)],
>  [(2, 8, 6), 0, (2, 6, 8), (2, 8), (), 0]]);
<Rees 0-matrix semigroup 6x6 over Group([ (2,8), (2,8,6) ])>
gap> map := RZMSIsoByTriple(R, R,
> [(), IdentityMapping(Group([(2, 8), (2, 8, 6)])),
> [(), (2, 6, 8), (), (), (), (2, 8, 6),
>  (2, 8, 6), (), (), (), (2, 6, 8), ()]]);;
gap> ImagesElm(map, RMSElement(R, 1, (2, 8), 2));
[ (1,(2,8),2) ]
```

17.2.6 Operators for isomorphisms of Rees (0-)matrix semigroup by triples

`map[i]`

`map[i]` returns the `i`th component of the Rees (0-)matrix semigroup isomorphism by triple *map* when `i = 1, 2, 3`; see `ELM_LIST` (17.2.3).

`map1 * map2`

returns the composition of *map2* and *map1*; see `CompositionMapping2` (17.2.4).

`map1 < map2`

returns `true` if *map1* is lexicographically less than *map2*.

`map1 = map2`

returns `true` if the Rees (0-)matrix semigroup isomorphisms by triple *map1* and *map2* have equal source and range, and are equal as functions, and `false` otherwise.

It is possible for *map1* and *map2* to be equal but to have distinct components.

`pt ^ map`

returns the image of the element *pt* of the source of *map* under the isomorphism *map*; see `ImagesElm` (17.2.5).

Chapter 18

Visualising semigroups and elements

There are two operations `TikzString` (18.4.1) and `DotString` (18.2.1) in Semigroups for creating LaTeX and `dot` (also known as GraphViz) format pictures of the Green's class structure of a semigroup and for visualising certain types of elements of a semigroup. There is also a function `Splash` (18.1.1) for automatically processing the output of `TikzString` (18.4.1) and `DotString` (18.2.1) and opening the resulting pdf file.

18.1 Automatic viewing

In this section we describe the function `Splash` (18.1.1) for automatically processing and opening the strings output by `TikzString` (18.4.1) and `DotString` (18.2.1)

18.1.1 Splash

▷ `Splash(str[, opts])` (function)

Returns: Nothing.

This function attempts to convert the string *str* into a pdf document and open this document, i.e. to splash it all over your monitor.

The string *str* must correspond to a valid `dot` or `LaTeX` text file and you must have have `GraphViz` and `pdflatex` installed on your computer. For details about these file formats, see `http://www.latex-project.org` and `http://www.graphviz.org`.

This function is provided to allow convenient, immediate viewing of the pictures produced by the functions: `TikzString` (18.4.1), `TikzString` (18.4.1), `DotSemilatticeOfIdempotents` (18.2.2), and `DotString` (18.2.1).

The optional second argument *opts* should be a record with components corresponding to various options, given below.

path this should be a string representing the path to the directory where you want `Splash` to do its work. The default value of this option is `"~/"`.

directory
this should be a string representing the name of the directory in `path` where you want `Splash` to do its work. This function will create this directory if does not already exist.

The default value of this option is `"tmp.viz"` if the option `path` is present, and the result of `DirectoryTemporary` (**Reference: DirectoryTemporary**) is used otherwise.

filename
: this should be a string representing the name of the file where *str* will be written. The default value of this option is `"vizpicture"`.

viewer
: this should be a string representing the name of the program which should open the files produced by `GraphViz` or `pdflatex`.

type
: this option can be used to specify that the string *str* contains a LaTeX or `dot` document. You can specify this option in *str* directly by making the first line `"%latex"` or `"//dot"`. There is no default value for this option, this option must be specified in *str* or in *opt.type*.

filetype
: this should be a string representing the type of file which `Splash` should produce. For LaTeX files, this option is ignored and the default value `"pdf"` is used.

This function was written by Attila Egri-Nagy and Manuel Delgado with some minor changes by J. D. Mitchell.

Example

```
gap> Splash(DotString(FullTransformationMonoid(4)));
```

18.2 `dot` pictures

In this section, we describe the operations in Semigroups for creating pictures in `dot` format.

The operations described in this section return strings, which can be written to a file using the function `FileString` (**GAPDoc: FileString**) or viewed using `Splash` (18.1.1).

18.2.1 DotString

▷ `DotString(S[, options])` (operation)

Returns: A string.

If the argument *S* is a semigroup, and the optional second argument *options* is a record, then this operation produces a graphical representation of the partial order of the $\mathscr{D}$-classes of the semigroup *S* together with the eggbox diagram of each $\mathscr{D}$-class. The output is in `dot` format (also known as `GraphViz`) format. For details about this file format, and information about how to display or edit this format see `http://www.graphviz.org`.

The string returned by `DotString` can be written to a file using the command `FileString` (**GAPDoc: FileString**).

The $\mathscr{D}$-classes are shown as eggbox diagrams with $\mathscr{L}$-classes as rows and $\mathscr{R}$-classes as columns; group $\mathscr{H}$-classes are shaded gray and contain an asterisk. The $\mathscr{L}$-classes and $\mathscr{R}$-classes within a $\mathscr{D}$-class are arranged to correspond to the normalization of the principal factor given by `NormalizedPrincipalFactor` (12.4.8). The $\mathscr{D}$-classes are numbered according to their index in `GreensDClasses(S)`, so that an `i` appears next to the eggbox diagram of `GreensDClasses(S)[i]`. A line from one $\mathscr{D}$-class to another indicates that the higher $\mathscr{D}$-class is greater than the lower one in the $\mathscr{D}$-order on *S*.

If the optional second argument *options* is present, it can be used to specify some options for output.

number

if *options*.number is false, then the $\mathscr{D}$-classes in the diagram are not numbered according to their index in the list of $\mathscr{D}$-classes of *S*. The default value for this option is true.

maximal

if *options*.maximal is true, then the structure description of the group $\mathscr{H}$-classes is displayed; see StructureDescription (**Reference: StructureDescription**). Setting this attribute to true can adversely affect the performance of DotString. The default value for this option is false.

normal

if *options*.normal is false, then the $\mathscr{L}$- and $\mathscr{R}$-classes within each $\mathscr{D}$-class arranged to correspond to PrincipalFactor (12.4.8). If *options*.normal is true, they are instead arranged to correspond to NormalizedPrincipalFactor (12.4.8). Setting this attribute to false may improve the performance of DotString as it avoids the computation of InjectionNormalizedPrincipalFactor (12.4.7). The default value for this option is true.

Example

```
gap> S := FullTransformationMonoid(3);
<full transformation monoid of degree 3>
gap> DotString(S);
"//dot\ndigraph  DClasses {\nnode [shape=plaintext]\nedge [color=blac\
k,arrowhead=none]\n1 [shape=box style=invisible label=<\n<TABLE BORDE\
R=\"0\" CELLBORDER=\"1\" CELLPADDING=\"10\" CELLSPACING=\"0\" PORT=\"\
1\">\n<TR BORDER=\"0\"><TD COLSPAN=\"1\" BORDER = \"0\" > 1</TD></TR>\
<TR><TD BGCOLOR=\"gray\">*</TD></TR>\n</TABLE>>];\n2 [shape=box style\
=invisible label=<\n<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLPADDING=\
\"10\" CELLSPACING=\"0\" PORT=\"2\">\n<TR BORDER=\"0\"><TD COLSPAN=\"\
3\" BORDER = \"0\" > 2</TD></TR><TR><TD BGCOLOR=\"gray\">*</TD><TD BG\
COLOR=\"gray\">*</TD><TD BGCOLOR=\"white\"></TD></TR>\n<TR><TD BGCOLO\
R=\"gray\">*</TD><TD BGCOLOR=\"white\"></TD><TD BGCOLOR=\"gray\">*</T\
D></TR>\n<TR><TD BGCOLOR=\"white\"></TD><TD BGCOLOR=\"gray\">*</TD><T\
D BGCOLOR=\"gray\">*</TD></TR>\n</TABLE>>];\n3 [shape=box style=invis\
ible label=<\n<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLPADDING=\"10\"\
 CELLSPACING=\"0\" PORT=\"3\">\n<TR BORDER=\"0\"><TD COLSPAN=\"1\" BO\
RDER = \"0\" > 3</TD></TR><TR><TD BGCOLOR=\"gray\">*</TD></TR>\n<TR><\
TD BGCOLOR=\"gray\">*</TD></TR>\n<TR><TD BGCOLOR=\"gray\">*</TD></TR>\
\n</TABLE>>];\n1 -> 2\n2 -> 3\n }"
gap> FileString("t3.dot", DotString(S));
1040
```

18.2.2 DotSemilatticeOfIdempotents

▷ DotSemilatticeOfIdempotents(*S*) (attribute)

Returns: A string.

This function produces a graphical representation of the semilattice of the idempotents of an inverse semigroup *S* where the elements of *S* have a unique semigroup inverse accessible via Inverse (**Reference: Inverse**). The idempotents are grouped by the $\mathscr{D}$-class they belong to.

The output is in dot format (also known as GraphViz) format. For details about this file format, and information about how to display or edit this format see http://www.graphviz.org.

Example
```
gap> S := DualSymmetricInverseMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> DotSemilatticeOfIdempotents(S);
"//dot\ngraph graphname {\n  node [shape=point]\nranksep=2;\nsubgraph \
cluster_1{\n15 \n}\nsubgraph cluster_2{\n5 11 14 12 13 8 \n}\nsubgraph\
 cluster_3{\n2 10 6 3 4 9 7 \n}\nsubgraph cluster_4{\n1 \n}\n2 -- 1\n3\
 -- 1\n4 -- 1\n5 -- 2\n5 -- 3\n5 -- 4\n6 -- 1\n7 -- 1\n8 -- 2\n8 -- 6\
\n8 -- 7\n9 -- 1\n10 -- 1\n11 -- 2\n11 -- 9\n11 -- 10\n12 -- 3\n12 -- \
6\n12 -- 9\n13 -- 3\n13 -- 7\n13 -- 10\n14 -- 4\n14 -- 6\n14 -- 10\n15\
 -- 5\n15 -- 8\n15 -- 11\n15 -- 12\n15 -- 13\n15 -- 14\n }"
```

18.3 `tex` output

In this section, we describe the operations in Semigroups for creating LaTeX representations of GAP objects. For pictures of GAP objects please see Section 18.4.

18.3.1 TexString

▷ `TexString(f[, n])` (operation)

Returns: A string.

This function produces a string containing LaTeX code for the transformation *f*. If the optional paramater *n* is used, then this is taken to be the degree of the transformation *f*, if the parameter *n* is not given, then `DegreeOfTransformation` (**Reference: DegreeOfTransformation**) is used by default. If *n* is less than the degree of *f*, then an error is given.

Example
```
gap> TexString(Transformation([6, 2, 4, 3, 6, 4]));
"\\begin{pmatrix}\n  1 & 2 & 3 & 4 & 5 & 6 \\\\\n  6 & 2 & 4 & 3 & 6 &\
 4\n\\end{pmatrix}"
gap> TexString(Transformation([1, 2, 1, 3]), 5);
"\\begin{pmatrix}\n  1 & 2 & 3 & 4 & 5 \\\\\n  1 & 2 & 1 & 3 & 5\n\\en\
d{pmatrix}"
```

18.4 `tikz` pictures

In this section, we describe the operations in Semigroups for creating pictures in `tikz` format.

The functions described in this section return a string, which can be written to a file using the function `FileString` (**GAPDoc: FileString**) or viewed using `Splash` (18.1.1).

18.4.1 TikzString

▷ `TikzString(obj[, options])` (operation)

Returns: A string.

This function produces a graphical representation of the object *obj* using the `tikz` package for LaTeX. More precisely, this operation outputs a string containing a minimal LaTeX document which can be compiled using LaTeX to produce a picture of *obj*.

Currently the following types of objects are supported:

blocks

If *obj* is the left or right blocks of a bipartition, then `TikzString` returns a graphical representation of these blocks; see Section 3.6.

bipartitions

If *obj* is a bipartition, then `TikzString` returns a graphical representation of *obj*.

If the optional second argument *options* is a record with the component `colors` set to `true`, then the blocks of *f* will be colored using the standard `tikz` colors. Due to the limited number of colors available in `tikz` this option only works when the degree of *obj* is less than 20. See Chapter 3 for more details about bipartitions.

pbrs If *obj* is a PBR, then `TikzString` returns a graphical representation *obj*; see Section 3.6.

Example

```
gap> x := Bipartition([[1, 4, -2, -3], [2, 3, 5, -5], [-1, -4]]);;
gap> TikzString(RightBlocks(x));
"%tikz\n\\documentclass{minimal}\n\\usepackage{tikz}\n\\begin{documen\
t}\n\\begin{tikzpicture}\n  \\draw[ultra thick](5,2)circle(.115);\n  \
\\draw(1.8,5) node [top] {{$1$}};\n  \\fill(4,2)circle(.125);\n  \\dr\
aw(1.8,4) node [top] {{$2$}};\n  \\fill(3,2)circle(.125);\n  \\draw(1\
.8,3) node [top] {{$3$}};\n  \\draw[ultra thick](2,2)circle(.115);\n \
 \\draw(1.8,2) node [top] {{$4$}};\n  \\fill(1,2)circle(.125);\n  \\d\
raw(1.8,1) node [top] {{$5$}};\n\n  \\draw (5,2.125) .. controls (5,2\
.8) and (2,2.8) .. (2,2.125);\n  \\draw (4,2.125) .. controls (4,2.6)\
 and (3,2.6) .. (3,2.125);\n\\end{tikzpicture}\n\n\\end{document}"
gap> x := Bipartition([[1, 5], [2, 4, -3, -5], [3, -1, -2], [-4]]);;
gap> TikzString(x);
"%tikz\n\\documentclass{minimal}\n\\usepackage{tikz}\n\\begin{documen\
t}\n\\begin{tikzpicture}\n\n  %block #1\n  %vertices and labels\n  \\\
fill(1,2)circle(.125);\n  \\draw(0.95, 2.2) node [above] {{ $1$}};\n \
 \\fill(5,2)circle(.125);\n  \\draw(4.95, 2.2) node [above] {{ $5$}};\
\n\n  %lines\n  \\draw(1,1.875) .. controls (1,1.1) and (5,1.1) .. (5\
,1.875);\n\n  %block #2\n  %vertices and labels\n  \\fill(2,2)circle(\
.125);\n  \\draw(1.95, 2.2) node [above] {{ $2$}};\n  \\fill(4,2)circ\
le(.125);\n  \\draw(3.95, 2.2) node [above] {{ $4$}};\n  \\fill(3,0)c\
ircle(.125);\n  \\draw(3, -0.2) node [below] {{ $-3$}};\n  \\fill(5,0\
)circle(.125);\n  \\draw(5, -0.2) node [below] {{ $-5$}};\n\n  %lines\
\n  \\draw(2,1.875) .. controls (2,1.3) and (4,1.3) .. (4,1.875);\n  \
\\draw(3,0.125) .. controls (3,0.7) and (5,0.7) .. (5,0.125);\n  \\dr\
aw(2,2)--(3,0);\n\n  %block #3\n  %vertices and labels\n  \\fill(3,2)\
circle(.125);\n  \\draw(2.95, 2.2) node [above] {{ $3$}};\n  \\fill(1\
,0)circle(.125);\n  \\draw(1, -0.2) node [below] {{ $-1$}};\n  \\fill\
(2,0)circle(.125);\n  \\draw(2, -0.2) node [below] {{ $-2$}};\n\n  %l\
ines\n  \\draw(1,0.125) .. controls (1,0.6) and (2,0.6) .. (2,0.125);\
\n  \\draw(3,2)--(2,0);\n\n  %block #4\n  %vertices and labels\n  \\f\
ill(4,0)circle(.125);\n  \\draw(4, -0.2) node [below] {{ $-4$}};\n\n \
%lines\n\\end{tikzpicture}\n\n\\end{document}"
gap> TikzString(UniversalPBR(2));
"%latex\n\\documentclass{minimal}\n\\usepackage{tikz}\n\\begin{docume\
nt}\n\\usetikzlibrary{arrows}\n\\usetikzlibrary{arrows.meta}\n\\newco\
mmand{\\arc}{\\draw[semithick, -{>[width = 1.5mm, length = 2.5mm]}]}\
\n\\begin{tikzpicture}[\n  vertex/.style={circle, draw, fill=black, i\
nner sep =0.04cm},\n  ghost/.style={circle, draw = none, inner sep = \
```

```
0.14cm},\n  botloop/.style={min distance = 8mm, out = -70, in = -110}\
,\n  toploop/.style={min distance = 8mm, out = 70, in = 110}]\n\n  % \
vertices and labels\n  \\foreach \\i in {1,...,2} {\n    \\node [vert\
ex] at (\\i/1.5, 3) {};\n    \\node [ghost] (\\i) at (\\i/1.5, 3) {};\
\n  }\n\n  \\foreach \\i in {1,...,2} {\n    \\node [vertex] at (\\i/\
1.5, 0) {};\n    \\node [ghost] (-\\i) at (\\i/1.5, 0) {};\n  }\n\n  \
% arcs from vertex 1\n  \\arc (1) to (-2);\n  \\arc (1) to (-1);\n  \
\\arc (1) edge [toploop] (1);\n  \\arc (1) .. controls (1.06666666666\
66667, 2.125) and (0.9333333333333324, 2.125) .. (2);\n\n  % arcs fr\
om vertex -1\n  \\arc (-1) .. controls (1.0666666666666667, 0.875) an\
d (0.9333333333333324, 0.875) .. (-2);\n  \\arc (-1) edge [botloop] \
(-1);\n  \\arc (-1) to (1);\n  \\arc (-1) to (2);\n\n  % arcs from ve\
rtex 2\n  \\arc (2) to (-2);\n  \\arc (2) to (-1);\n  \\arc (2) .. co\
ntrols (0.9333333333333324, 2.125) and (1.0666666666666667, 2.125) .\
. (1);\n  \\arc (2) edge [toploop] (2);\n\n  % arcs from vertex -2\n \
 \\arc (-2) edge [botloop] (-2);\n  \\arc (-2) .. controls (0.9333333\
3333333324, 0.875) and (1.0666666666666667, 0.875) .. (-1);\n  \\arc \
(-2) to (1);\n  \\arc (-2) to (2);\n\n\\end{tikzpicture}\n\\end{docum\
ent}"
```

Chapter 19

IO

19.1 Reading and writing elements to a file

The functions `ReadGenerators` (19.1.1) and `WriteGenerators` (19.1.2) can be used to read or write, respectively, elements of a semigroup to a file.

19.1.1 ReadGenerators

▷ `ReadGenerators(filename[, nr])` (function)

▷ `ReadOldGenerators(filename[, nr])` (function)

Returns: A list of lists of semigroup elements.

If *filename* is the name of a file created using `WriteGenerators` (19.1.2), then `ReadGenerators` returns the contents of this file as a list of lists of elements of a semigroup.

If the optional second argument *nr* is present, then `ReadGenerators` returns the elements stored in the *nr*th line of *filename*.

If you want to read generators from a file written using `WriteGenerators` from a version of Semigroups before version 3.0.0, then you can use `ReadOldGenerators`.

Example

```
gap> file := Concatenation(SEMIGROUPS.PackageDir,
> "/data/tst/testdata");;
gap> ReadGenerators(file, 13);
[ <identity partial perm on [ 2, 3, 4, 5, 6 ]>,
  <identity partial perm on [ 2, 3, 5, 6 ]>, [1,2](5)(6) ]
```

19.1.2 WriteGenerators

▷ `WriteGenerators(filename, list[, append])` (function)

Returns: `IO_OK` or `IO_ERROR`.

This function provides a method for writing collections of elements of a semigroup to a file. The resulting file can be further compressed using `gzip` or `xz`.

The argument *list* should be a list of elements, a semigroup, or a list of lists of elements, or semigroups.

The argument *filename* should be a string containing the name of a file where the entries in *list* will be written or an IO package file object; see `IO_File` (IO_File???) and `IO_CompressedFile` (IO_CompressedFile???).

If the optional third argument *append* is given and equals "w", then the previous content of the file is deleted and overwritten. If the optional third argument is "a" or is not present, then `list` is appended to the file. This function returns `IO_OK` (IO_OK???) if everything went well or `IO_ERROR` (IO_ERROR???) if something went wrong.

`WriteGenerators` appends a line to the file *filename* for every entry in *list*. If any element of *list* is a semigroup, then the generators of that semigroup are written to *filename*. More specifically, the list returned by `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**) is written to the file.

The file *filename* can be read using `ReadGenerators` (19.1.1).

From Version 3.0.0 onwards the Semigroups package used the IO package pickling functionality; see (Pickling and unpickling???) for more details. This approach is used because it is more general and more robust than the methods used by earlier versions of Semigroups, although the performance is somewhat worse, and the resulting files are somewhat larger.

A file written in the old format can be read using `ReadOldGenerators` (19.1.1).

19.1.3 IteratorFromPickledFile

▷ `IteratorFromPickledFile(`*filename*`)` (function)

▷ `IteratorFromOldGeneratorsFile(`*filename*`)` (function)

Returns: An iterator.

If *filename* is a string containing the name of a file created using `WriteGenerators` (19.1.2), then `IteratorFromPickledFile` returns an iterator `iter` such that `NextIterator(iter)` returns the next collection of generators stored in the file *filename*.

This function is a convenient way of, for example, looping over a collection of generators in a file without loading every object in the file into memory. This might be useful if the file contains more information than there is available memory.

If you want to get an iterator for a file written using `WriteGenerators` from a version of Semigroups before version 3.0.0, then you can use `IteratorFromOldGeneratorsFile`.

References

[ABMN10] J. Araújo, P. V. Bünau, J. D. Mitchell, and M. Neunhöffer. Computing automorphisms of semigroups. *J. Symbolic Comput.*, 45(3):373–392, 2010. 7, 175

[ABMS14] J. Araújo, W. Bentz, J. D. Mitchell, and Csaba Schneider. The rank of the semigroup of transformations stabilising a partition of a finite set. in preparation, March 2014. 101

[Aui12] Karl Auinger. Krohn–rhodes complexity of brauer type semigroups. *Portugaliae Mathematica*, 69(4):341–360, 2012. 14

[BDF15] Andries E. Brouwer, Jan Draisma, and Bart J. Frenk. Lossy gossip and composition of metrics. *Discrete & Computational Geometry*, 53(4):890–913, 2015. 115

[BF92] Olsder G. J. Quadrat J.P. Baccelli F., Cohen G. *Synchronisation and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992. 71, 73

[Bur16] S. A. Burrell. The Order Problem for Natural and Tropical Matrix Semigroups. Master's thesis, University of St Andrews, United Kingdom, 2016. 73, 74

[DMW16] C. Donoven, J. D. Mitchell, and W. Wilson. Computing maximal subsemigroups of a finite semigroup. *submitted*, 06 2016. 7, 172, 173

[Eas16] James East. Presentations for rook partition monoids and algebras and their singular ideals. *arXiv:1606.00563*, 06 2016. 105

[EENMP15] J. East, A. Egri-Nagy, J. D. Mitchell, and Y. Péresse. Computing finite semigroups. *J. Symbolic Comput. (to appear)*, 10 2015. 77, 78, 148

[Far09] K.G. Farlow. Max-Plus Algebra. Master's thesis, Virginia Polytechnic Institute and State University, United States, 2009. 71, 73

[FL98] D.G. Fitzgerald and J. Leech. Dual symmetric inverse monoids and representation theory. *J. Austral. Math. Soc. A*, 64:345–67, 1998. 19

[FP97] Véronique Froidure and Jean-Eric Pin. Algorithms for computing finite semigroups. In *Foundations of computational mathematics (Rio de Janeiro, 1997)*, pages 112–126. Springer, Berlin, 1997. 47, 78

[Gau96] S. Gaubert. On the Burnside problem for Semigroups of Matrices over the (max, +) Algebra. *Semigroup Forum*, 5:271–292, 1996. 73, 74

[GGR68] N. Graham, R. Graham, and J. Rhodes. Maximal subsemigroups of finite semigroups. *J. Combinatorial Theory*, 4:203–209, 1968. 7, 171, 172, 173

[Gra68] R. Graham. On finite 0-simple semigroups and graph theory. *Mathematical systems theory*, 2(4):325–339, 1968. 92

[Gro06] Cheryl Grood. The rook partition algebra. *J. Combin. Theory Ser. A*, 113(2):325–351, 2006. 105

[How95] John M. Howie. *Fundamentals of semigroup theory*, volume 12 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 1995. Oxford Science Publications. 121, 124, 187, 206, 228, 229, 232, 233, 234, 235, 241, 242

[HR05] Tom Halverson and Arun Ram. Partition algebras. *European Journal of Combinatorics*, 26(6):869–921, 2005. 14, 105

[JK07] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007. 62

[KM11] Ganna Kudryavtseva and Victor Maltcev. Two generalisations of the symmetric inverse semigroups. *Publ. Math. Debrecen*, 78(2):253–282, 2011. 108

[KMU15] Ganna Kudryavtseva, Victor Maltcev, and Abdullahi Umar. Presentation for the partial dual symmetric inverse monoid. *Comm. Algebra*, 43(4):1621–1639, 2015. 108

[MM11] Paul Martin and Volodymyr Mazorchuk. Partitioned binary relations. *arXiv:1102.0862*, 2011. 35, 78

[MM16] Zachary Mesyan and J. D. Mitchell. The structure of a graph inverse semigroup. *Semigroup Forum*, 93(1):111–130, mar 2016. 128, 181

[Sch92] Boris M. Schein. The minimal degree of a finite inverse semigroup. *Trans. Amer. Math. Soc.*, 333(2):877–888, 1992. 7, 202

[Sim78] I. Simon. Limited subsets of a free monoid. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 143–150, Washington, DC, USA, 1978. IEEE Computer Society. 73

Index