Semigroups

Version 2.7.2

J. D. Mitchell
Manuel Delgado
James East
Attila Egri-Nagy
Julius Jonušas
Markus Pfeiffer
Ben Steinberg
Jhevon Smith
Michael Torpey
Wilf 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, principally of transformations, partial permutations, bipartitions, subsemigroups of regular Rees 0-matrix semigroups, free inverse semigroups, free bands, and semigroups of matrices over finite fields.

Semigroups contains more efficient methods than those available in the GAP library (and in many cases more efficient than any other software) for creating semigroups, 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 many more. It is also 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 congruences of certain types of semigroups, the normalizer of a semigroup in a permutation group, the maximal subsemigroups of a finite semigroup, and 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 bipartitions.

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.

Manuel Delgado and Attila Egri-Nagy contributed to the functions `Splash` (4.8.1) and `DotDClasses` (4.8.2).

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.

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

Yann Peresse and Yanhui Wang contributed to the function `MunnSemigroup` (2.5.10).

Jhevon Smith and Ben Steinberg contributed the function `CharacterTableOfInverseSemigroup` (4.7.16).

Michael Torpey contributed the part of the package relating to congruences of Rees (0-)matrix semigroups.

Wilf 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.

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

We would also like to acknowledge the support of the Centre of Algebra at the University of Lisbon, and of EPSRC grant number GR/S/56085/01.

Contents

1 The Semigroups package **6**
1.1 Introduction . . . 6
1.2 Installing the Semigroups package . . . 7
1.3 Compiling the documentation . . . 8
1.4 Testing the installation . . . 9
1.5 More information during a computation . . . 9
1.6 Reading and writing elements to a file . . . 10

2 Creating semigroups and monoids **12**
2.1 Random semigroups . . . 12
2.2 New semigroups from old . . . 14
2.3 Options when creating semigroups . . . 16
2.4 Changing the representation of a semigroup . . . 18
2.5 Standard examples . . . 21

3 Ideals **30**
3.1 Creating ideals . . . 30
3.2 Attributes of ideals . . . 31

4 Determining the structure of a semigroup **33**
4.1 Expressing semigroup elements as words in generators . . . 33
4.2 Creating Green's classes . . . 35
4.3 Iterators and enumerators of classes and representatives . . . 39
4.4 Attributes and properties directly related to Green's classes . . . 44
4.5 Further attributes of semigroups . . . 55
4.6 Further properties of semigroups . . . 71
4.7 Properties and attributes of inverse semigroups . . . 83
4.8 Visualising the structure of a semigroup . . . 94

5 Bipartitions and blocks **97**
5.1 The family and categories of bipartitions . . . 98
5.2 Creating bipartitions . . . 98
5.3 Changing the representation of a bipartition . . . 101
5.4 Operators for bipartitions . . . 104
5.5 Attributes for bipartitons . . . 106
5.6 Creating blocks and their attributes . . . 112
5.7 Actions on blocks . . . 113

5.8 Visualising blocks and bipartitions 115
5.9 Semigroups of bipartitions 116

6 Free inverse semigroups and free bands **119**
6.1 Free inverse semigroups 119
6.2 Displaying free inverse semigroup elements 121
6.3 Operators and operations for free inverse semigroup elements 121
6.4 Free bands 122
6.5 Operators and operations for free band elements 124

7 Matrix semigroups **125**
7.1 Creating matrix semigroups 125
7.2 Matrices in the Semigroups package 126
7.3 Matrix groups in the Semigroups package 130

8 Congruences **132**
8.1 Creating congruences 132
8.2 Congruence classes 133
8.3 Congruences on Rees matrix semigroups 134
8.4 Universal congruences 139

9 Homomorphisms **141**
9.1 Isomorphisms 141

10 Orbits **143**
10.1 Looking for something in an orbit 143
10.2 Strongly connected components of orbits 144

References **148**

Index **149**

Chapter 1

The Semigroups package

1.1 Introduction

This is the manual for the Semigroups package version 2.7.2. Semigroups 2.7.2 is a distant descendant of the Monoid package for GAP 3 by Goetz Pfeiffer, Steve A. Linton, Edmund F. Robertson, and Nik Ruskuc; and the Monoid package for GAP 4 by J. D. Mitchell.

Many of the operations, methods, properties, and functions described in this manual only apply to semigroups of transformations, partial permutations, bipartitions, subsemigroups of regular Rees 0-matrix semigroups over groups, semigroups of matrices over finite fields, free inverse semigroups, and free bands. For the sake of brevity, we have opted to say SEMIGROUP to describe the aforementioned classes of semigroups.

Semigroups 2.7.2 contains more efficient methods than those available in the GAP library (and in many cases more efficient than any other software) for creating semigroups and 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 many more; see Chapters 2, 3, and 4. There are also methods for testing if a semigroup satisfies a particular property, such as if it is regular, simple, inverse, completely regular, and a variety of further properties; see Chapter 4. The theory behind the main algorithms in Semigroups will be described in a forthcoming article.

It is harder for Semigroups to compute Green's $\mathscr{L}$- and $\mathscr{H}$-classes of a transformation semigroup. 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 semigroups of partial permutations, bipartitions, subsemigroups of a regular Rees 0-matrix semigroup over a group, or semigroups of matrices over a finite field.

The methods in Semigroups allow the computation of individual Green's classes without computing the entire data structure of the underlying semigroup; see `GreensRClassOfElementNC` (4.2.3). 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` (4.3.1), `RClassReps` (4.3.4), `IteratorOfRClassReps` (4.3.2), `IteratorOfRClasses` (4.3.3), or `NrRClasses` (4.4.6). This may be useful if you want to study a very large semigroup where computing all the elements of the semigroup is not feasible.

There are methods for finding: congruences of certain types of semigroups (based on Section 3.5

in [How95]), the normalizer of a semigroup in a permutation group (as given in [ABMN10]), the maximal subsemigroups of a finite semigroup (based on [GGR68]), 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, and for drawing bipartitions; see Sections 4.8 and 5.8.

Several standard examples of semigroups are provided see Section 2.5. Semigroups also provides functions to read and write collections of transformations, partial permutations, and bipartitions to a file; see `ReadGenerators` (1.6.2) and `WriteGenerators` (1.6.3).

Details of how to create and manipulate semigroups of bipartitions can be found in Chapter 5.

Details of how to create and manipulate semigroups of matrices over a finite field can be found in Chapter 7.

There are also functions in Semigroups to define and manipulate free inverse semigroups and their elements; this part of the package was written by Julius Jonušas; see Chapter 6 and Section 5.10 in [How95] for more details.

Semigroups contains functions synonymous to some of those defined in the GAP library but, for the sake of convenience, they have abbreviated names; further details can be found at the appropriate points in the later chapters of this manual.

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**).

The Semigroups package is written GAP code and requires the Orb and IO 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 transformations, partial permutations, and bipartitions to a file.

The Grape package must be loaded for the operation `SmallestMultiplicationTable` (9.1.2) to work, and it must be fully compiled for the following functions to work:

- `MunnSemigroup` (2.5.10)
- `MaximalSubsemigroups` (4.5.7)
- `IsIsomorphicSemigroup` (9.1.1)
- `IsomorphismSemigroups` (9.1.3).

If Grape is not available or is not compiled, then Semigroups can be used as normal with the exception that the functions above will not work.

The genss package is used in one version of the function `Normalizer` (4.5.23) but nowhere else in Semigroups. If genss is not available, then Semigroups can be used as normal with the exception that this function will not work.

Some further details about semigroups in GAP and Green's relations in particular, can be found in (**Reference: Semigroups**) and (**Reference: Green's Relations**).

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

1.2 Installing the Semigroups package

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 4.8.2 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 4.7.3 or higher is available. Orb and Semigroups both perform better if Orb is compiled.
- THIS STEP IS OPTIONAL: certain functions in Semigroups require the Grape package to be available and fully compiled; a full list of these functions can be found above. To use these functions make sure that the Grape package version 4.5 or higher is available. If Grape is not fully installed (i.e. compiled), then Semigroups can be used as normal with the exception that the functions listed above will not work.
- THIS STEP IS OPTIONAL: the non-deterministic version of the function `Normalizer` (4.5.23) requires the genss package to be loaded. If you want to use this function, then please ensure that the genss package version 1.5 or higher is available.
- download the package archive `semigroups-2.7.2.tar.gz` from the Semigroups package webpage.
- unzip and untar the file, this should create a directory called `semigroups-2.7.2`.
- locate the `pkg` directory of your GAP directory, which contains the directories `lib`, `doc` and so on. Move the directory `semigroups-2.7.2` into the `pkg` directory.
- start GAP in the usual way.
- type `LoadPackage("semigroups");`
- compile the documentation by using `SemigroupsMakeDoc` (1.3.1).

Presuming that the above steps can be completed successfully you will be running the Semigroups package!

If you want to check that the package is working correctly, you should run some of the tests described in Section 1.4.

1.3 Compiling the documentation

To compile the documentation use `SemigroupsMakeDoc` (1.3.1). If you want to use the help system, it is essential that you compile the documentation.

1.3.1 SemigroupsMakeDoc

▷ `SemigroupsMakeDoc()` (function)

Returns: Nothing.

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

1.4 Testing the installation

In this section we describe how to test that Semigroups is working as intended. To test that Semigroups is installed correctly use `SemigroupsTestInstall` (1.4.1) or for more extensive tests use `SemigroupsTestAll` (1.4.3). Please note that it will take a few seconds for `SemigroupsTestInstall` (1.4.1) to finish and it may take several minutes for `SemigroupsTestAll` (1.4.3) to finish.

If something goes wrong, then please review the instructions in Section 1.2 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.

1.4.1 SemigroupsTestInstall

▷ `SemigroupsTestInstall()` (function)

Returns: Nothing.

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 fraction of a second to complete. To more comprehensively test that Semigroups is installed correctly use `SemigroupsTestAll` (1.4.3).

1.4.2 SemigroupsTestManualExamples

▷ `SemigroupsTestManualExamples()` (function)

Returns: Nothing.

This function should be called with no argument to test the examples in the Semigroups manual. These tests should take no more than a few minutes to complete. To more comprehensively test that Semigroups is installed correctly use `SemigroupsTestAll` (1.4.3). See also `SemigroupsTestInstall` (1.4.1).

1.4.3 SemigroupsTestAll

▷ `SemigroupsTestAll()` (function)

Returns: Nothing.

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` (1.4.1).

1.5 More information during a computation

1.5.1 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. We recommend that you set the level to 1 so that basic 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: SetInfoLevel**).

1.6 Reading and writing elements to a file

The functions `ReadGenerators` (1.6.2) and `WriteGenerators` (1.6.3) can be used to read or write transformations, partial permutations, and bipartitions to a file.

1.6.1 SemigroupsDir

▷ `SemigroupsDir()` (function)

Returns: A string.

This function returns the absolute path to the Semigroups package directory as a string. The same result can be obtained typing:

Example
```
PackageInfo("semigroups")[1]!.InstallationPath;
```

at the GAP prompt.

1.6.2 ReadGenerators

▷ `ReadGenerators(`*`filename[, nr]`*`)` (function)

Returns: A list of lists of semigroup elements.

If *`filename`* is the name of a file created using `WriteGenerators` (1.6.3), then `ReadGenerators` returns the contents of this file as a list of lists of transformations, partial permutations, or bipartitions.

If the optional second argument *`nr`* is present, then `ReadGenerators` returns the elements stored in the *`nr`*th line of *`filename`*.

Example
```
gap> file:=Concatenation(SemigroupsDir(), "/tst/test.gz");;
gap> ReadGenerators(file, 1378);
[ Transformation( [ 1, 2, 2 ] ), IdentityTransformation,
  Transformation( [ 1, 2, 3, 4, 5, 7, 7 ] ),
  Transformation( [ 1, 3, 2, 4, 7, 6, 7 ] ),
  Transformation( [ 4, 2, 1, 1, 6, 5 ] ),
  Transformation( [ 4, 3, 2, 1, 6, 7, 7 ] ),
  Transformation( [ 4, 4, 5, 7, 6, 1, 1 ] ),
  Transformation( [ 7, 6, 6, 1, 2, 4, 4 ] ),
  Transformation( [ 7, 7, 5, 4, 3, 1, 1 ] ) ]
```

1.6.3 WriteGenerators

▷ `WriteGenerators(`*`filename, list[, append]`*`)` (function)

Returns: `true` or `fail`.

This function provides a method for writing transformations, partial permutations, and bipartitions to a file, that uses a relatively small amount of disk space. 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 types of elements and semigroups supported are: transformations, partial permutations, and bipartitions.

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.

If the optional third argument *append* is given and equals "w", then the previous content of the file is deleted. If the optional third argument is "a" or is not present, then `list` is appended to the file. This function returns `true` if everything went well or `fail` 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*.

The first character of the appended line indicates which type of element is contained in that line, the second character `m` is the number of characters in the degree of the elements to be written, the next `m` characters are the degree `n` of the elements to be written, and the internal representation of the elements themselves are written in blocks of `m*n` in the remainder of the line. For example, the transformations:

Example
```
[ Transformation( [ 2, 6, 7, 2, 6, 9, 9, 1, 1, 5 ] ),
  Transformation( [ 3, 8, 1, 9, 9, 4, 10, 5, 10, 6 ] )]
```

are written as:

Example
```
t210 2 2 6 7 2 6 9 9 1 1 5 3 8 1 9 9 410 510 6
```

The file *filename* can be read using `ReadGenerators` (1.6.2).

1.6.4 IteratorFromGeneratorsFile

▷ `IteratorFromGeneratorsFile(`*filename*`)` (function)

Returns: An iterator.

If *filename* is a string containing the name of a file created using `WriteGenerators` (1.6.3), then `IteratorFromGeneratorsFile` 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.

Chapter 2

Creating semigroups and monoids

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

Any semigroup created before Semigroups has been loaded must be recreated after Semigroups is loaded so that the options record (described in Section 2.3) is defined. Almost all of the functions and methods provided by Semigroups, including those methods for existing GAP library functions, will return an error when applied to a semigroup created before Semigroups is loaded.

2.1 Random semigroups

2.1.1 RandomInverseMonoid

▷ `RandomInverseMonoid(m, n)` (operation)
▷ `RandomInverseSemigroup(m, n)` (operation)

Returns: An inverse monoid or semigroup.

Returns a random inverse monoid or semigroup of partial permutations with degree at most *n* with *m* generators.

Example
```
gap> S := RandomInverseSemigroup(10, 10);
<inverse partial perm semigroup of rank 10 with 10 generators>
gap> S := RandomInverseMonoid(10, 10);
<inverse partial perm monoid of rank 10 with 10 generators>
```

2.1.2 RandomTransformationMonoid

▷ `RandomTransformationMonoid(m, n)` (operation)
▷ `RandomTransformationSemigroup(m, n)` (operation)

Returns: A transformation semigroup or monoid.

Returns a random transformation monoid or semigroup of at most degree *n* with *m* generators.

Example
```
gap> S := RandomTransformationMonoid(5, 5);
<transformation monoid of degree 5 with 5 generators>
gap> S := RandomTransformationSemigroup(5, 5);
<transformation semigroup of degree 5 with 5 generators>
```

2.1.3 RandomPartialPermMonoid

▷ RandomPartialPermMonoid(*m*, *n*) (operation)
▷ RandomPartialPermSemigroup(*m*, *n*) (operation)

Returns: A partial perm semigroup or monoid.

Returns a random partial perm monoid or semigroup of degree at most *n* with *m* generators.

Example
```
gap> S:=RandomPartialPermSemigroup(5, 5);
<partial perm semigroup of rank 4 with 5 generators>
gap> S:=RandomPartialPermMonoid(5, 5);
<partial perm monoid of degree 5 with 5 generators>
```

2.1.4 RandomBinaryRelationMonoid

▷ RandomBinaryRelationMonoid(*m*, *n*) (operation)
▷ RandomBinaryRelationSemigroup(*m*, *n*) (operation)

Returns: A semigroup or monoid of binary relations.

Returns a random monoid or semigroup of binary relations on *n* points with *m* generators.

Example
```
gap> RandomBinaryRelationSemigroup(5,5);
<semigroup with 5 generators>
gap> RandomBinaryRelationMonoid(5,5);
<monoid with 5 generators>
```

2.1.5 RandomBipartitionSemigroup

▷ RandomBipartitionSemigroup(*m*, *n*) (operation)
▷ RandomBipartitionMonoid(*m*, *n*) (operation)

Returns: A bipartition semigroup or monoid.

Returns a random monoid or semigroup of bipartition on *n* points with *m* generators.

Example
```
gap> RandomBipartitionMonoid(5, 5);
<bipartition monoid of degree 5 with 5 generators>
gap> RandomBipartitionSemigroup(5, 5);
<bipartition semigroup of degree 5 with 5 generators>
```

2.1.6 RandomMatrixSemigroup

▷ RandomMatrixSemigroup(*R*, *m*, *n*[, *ranks*]) (operation)
▷ RandomMatrixMonoid(*R*, *m*, *n*[, *ranks*]) (operation)

Returns: A matrix semigroup or monoid.

Returns a random semigroup or monoid of *n*-by-*n* matrices over the ring *R* with *m* generators.

The optional fourth argument *ranks* is expected to be a list of permissible ranks for the generators. For any generator the rank is chosen uniformly randomly from the list of permissible ranks. This allows for creating more interesting random matrix semigroups and monoids. Without the *ranks* argument there is a very high probability that the semigroups returned by this function are full matrix monoids over the base ring.

Example
```
gap> RandomMatrixSemigroup(GF(25),5,5);
<semigroup of 5x5 matrices over GF(5^2) with 5 generators>
gap> RandomMatrixSemigroup(GF(4),2,5,[1,2]);
<semigroup of 5x5 matrices over GF(2^2) with 2 generators>
```

2.2 New semigroups from old

2.2.1 ClosureInverseSemigroup

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

Returns: An inverse semigroup or monoid.

This function returns the inverse semigroup or monoid generated by the inverse semigroup `S` and the collection of elements `coll` after first removing duplicates and elements in `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`.

If present, the optional third argument `opts` should be a record containing the values of the options for the inverse semigroup being created; these options are described in Section 2.3.

Example
```
gap> S:=InverseMonoid(
> PartialPerm( [ 1, 2, 3, 5, 6, 7, 8 ], [ 5, 9, 10, 6, 3, 8, 4 ] ),
> PartialPerm( [ 1, 2, 4, 7, 8, 9 ], [ 10, 7, 8, 5, 9, 1 ] ) );;
gap> f:=PartialPerm(
> [ 1, 2, 3, 4, 5, 7, 8, 10, 11, 13, 18, 19, 20 ],
> [ 5, 1, 7, 3, 10, 2, 12, 14, 11, 16, 6, 9, 15 ]);;
gap> S:=ClosureInverseSemigroup(S, f);
<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 854 generators>
gap> S:=InverseSemigroup(SmallGeneratingSet(S));
<inverse partial perm semigroup of rank 19 with 14 generators>
```

2.2.2 ClosureSemigroup

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

Returns: A semigroup or monoid.

This function returns the semigroup or monoid generated by the semigroup `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`.

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 2.3.

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 i in [2..4] do S:=ClosureSemigroup(S, gens[i]); od;
gap> S;
<transformation monoid of degree 8 with 4 generators>
gap> Size(S);
233606
gap> gens:=
> [ NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,GF(25),2,
>     [ [ Z(5^2), Z(5^2)^13 ], [ 0*Z(5), Z(5^2)^14 ] ]),
>  NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,GF(25),2,
>     [ [ Z(5^2)^21, Z(5)^0 ], [ Z(5)^0, 0*Z(5) ] ]),
>  NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,GF(25),2,
>     [ [ Z(5^2)^23, Z(5^2)^5 ], [ Z(5^2)^20, Z(5^2)^20 ] ]) ];;
gap> S := Semigroup(gens[1]);
<semigroup of 2x2 matrices over GF(5^2) with 1 generator>
gap> Size(S);
24
gap> S := ClosureSemigroup(S, gens[2]);
<semigroup of 2x2 matrices over GF(5^2) with 2 generators>
gap> Size(S);
124800
gap> S := ClosureSemigroup(S, gens[3]);
<semigroup of 2x2 matrices over GF(5^2) with 3 generators>
gap> Size(S);
374400
```

2.2.3 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(f) return 1 ^ f <> 1 and
> ForAll([1..DegreeOfTransformation(f)], y-> y = 1 or y ^ f = y); end;
function( f ) ... 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>
```

2.2.4 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
```

2.3 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**),
- `ClosureInverseSemigroup` (2.2.1),
- `ClosureSemigroup` (2.2.2),
- `SemigroupIdeal` (3.1.1)

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* of transformations, partial permutations, Rees 0-matrix semigroup elements, or bipartitions; or *S* is an ideal of such a semigroup.

`acting`
: this component should be `true` or `false`. In order for a semigroup to use the methods in Semigroups it must satisfy `IsActingSemigroup`. By default any semigroup or monoid of transformations, partial permutations, Rees 0-matrix elements, or bipartitions satisfies `IsActingSemigroup`. From time to time, it might be preferable to use the exhaustive algorithm in the GAP library to compute with a semigroup. 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 methods in the Semigroups package 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: SetInfoLevel**).

`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` (2.2.2) or `ClosureInverseSemigroup` (2.2.1).

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 `SemigroupsOptionsRec` (2.3.1). 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> SemigroupsOptionsRec.small:=true;
true
```

If you want to change the default values of the options stored in `SemigroupsOptionsRec` (2.3.1) for all GAP sessions, then you can edit these values in the file `semigroups/gap/options.g`.

2.3.1 SemigroupsOptionsRec

▷ `SemigroupsOptionsRec` (global variable)

This global variable is a record whose components contain the default values of certain options for transformation semigroups created after Semigroups has been loaded. A description of these options is given above in Section 2.3.

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

Example
```
rec( acting := true, hashlen := rec( L := 25013, M := 6257, S :=
         251 ), regular := false, small := false )
```

2.4 Changing the representation of a semigroup

In addition, to the library functions

- `IsomorphismReesMatrixSemigroup` (**Reference: IsomorphismReesMatrixSemigroup**),
- `AntiIsomorphismTransformationSemigroup` (**Reference: AntiIsomorphismTransformationSemigroup**),
- `IsomorphismTransformationSemigroup` (**Reference: IsomorphismTransformationSemigroup**),
- `IsomorphismPartialPermSemigroup` (**Reference: IsomorphismPartialPermSemigroup**),

there are several methods for changing the representation of a semigroup in Semigroups. There are also methods for the operations given above for the types of semigroups defined in Semigroups which are not mentioned in the reference manual.

2.4.1 AsTransformationSemigroup

▷ `AsTransformationSemigroup(S)` (operation)
▷ `AsPartialPermSemigroup(S)` (operation)
▷ `AsBipartitionSemigroup(S)` (operation)
▷ `AsBlockBijectionSemigroup(S)` (operation)
▷ `AsMatrixSemigroup(S[, F])` (operation)

Returns: A semigroup.

`AsTransformationSemigroup(S)` is just shorthand for `Range(IsomorphismTransformationSemigroup(S))` when S is a semigroup; see `IsomorphismTransformationSemigroup` (**Reference: IsomorphismTransformationSemigroup**) for more details.

The operations:

- `AsPartialPermSemigroup`;
- `AsBipartitionSemigroup`;
- `AsBlockBijectionSemigroup`;

are analogous to `AsTransformationSemigroup`.

`AsMatrixSemigroup` returns the range of an isomorphism from S to a semigroup of matrices over `GF(2)`. If the optional argument F is present, then `AsMatrixSemigroup` returns an isomorphic semigroup over the finite field F.

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> AsTransformationSemigroup(S);
<transformation semigroup of degree 12 with 2 generators>
gap> AsMatrixSemigroup(S);
<semigroup of 12x12 matrices over GF(2) with 2 generators>
gap> T := Semigroup(Transformation([2, 2, 3]), Transformation([3, 1, 3]));
<transformation semigroup of degree 3 with 2 generators>
gap> S := AsMatrixSemigroup(T, GF(5));
<semigroup of 3x3 matrices over GF(5) with 2 generators>
gap> Size(S);
5
```

2.4.2 IsomorphismPermGroup

▷ `IsomorphismPermGroup(S)` (operation)

Returns: An isomorphism.

If the semigroup S is mathematically a group, so that it satisfies `IsGroupAsSemigroup` (4.6.6), then `IsomorphismPermGroup` returns an isomorphism to a permutation group.

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

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> IsomorphismPermGroup(S);
MappingByFunction( <transformation group of degree 8 with 2 generators
 >, Group([ (5,6,8), (2,3,8,
4) ]), <Attribute "PermutationOfImage">, function( x ) ... end )
gap> StructureDescription(Range(IsomorphismPermGroup(S)));
"S6"
gap> S := Range(IsomorphismPartialPermSemigroup(SymmetricGroup(4)));
<inverse partial perm semigroup of rank 4 with 2 generators>
gap> IsomorphismPermGroup(S);
MappingByFunction( <partial perm group of rank 4 with 2 generators>
, Group([ (1,2,3,4), (1,
2) ]), <Attribute "AsPermutation">, function( x ) ... end )
gap> G := GroupOfUnits(PartitionMonoid(4));
<bipartition group of degree 4 with 2 generators>
gap> StructureDescription(G);
"S4"
gap> iso := IsomorphismPermGroup(G);
MappingByFunction( <bipartition group of degree 4 with 2 generators>
, S4, <Attribute "AsPermutation">, function( x ) ... end )
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
```

2.4.3 IsomorphismBipartitionSemigroup

▷ IsomorphismBipartitionSemigroup(*S*) (attribute)

▷ IsomorphismBipartitionMonoid(*S*) (attribute)

Returns: An isomorphism.

If S is a semigroup, then IsomorphismBipartitionSemigroup returns an isomorphism from S to a bipartition semigroup. When S is a transformation semigroup, partial permutation semigroup, or a permutation group, on n points, IsomorphismBipartitionSemigroup returns the natural embedding of S into the partition monoid on n points. When S is a generic semigroup, this funciton returns the right regular representation of S acting on S with an identity adjoined.

See AsBipartition (5.3.1).

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> IsomorphismBipartitionSemigroup(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> ForAll(Generators(Range(last)), IsPartialPermBipartition);
true
```

2.4.4 IsomorphismBlockBijectionSemigroup

▷ `IsomorphismBlockBijectionSemigroup(S)` (attribute)
▷ `IsomorphismBlockBijectionMonoid(S)` (attribute)

Returns: An isomorphism.

If S is a partial perm semigroup on `n` points, then this function returns the embedding of S into a subsemigroup of the dual symmetric inverse monoid on `n+1` points given by the FitzGerald-Leech Theorem [FL98].

See `AsBlockBijection` (5.3.2) for more details.

Example
```
gap> S := SymmetricInverseMonoid(4);
<symmetric inverse monoid of degree 4>
gap> IsomorphismBlockBijectionSemigroup(S);
MappingByFunction( <symmetric inverse monoid of degree 4>,
<inverse bipartition monoid of degree 5 with 3 generators>
 , function( x ) ... end, function( x ) ... end )
gap> Size(Range(last));
209
gap> S:=Semigroup( PartialPerm( [ 1, 2 ], [ 3, 1 ] ),
> PartialPerm( [ 1, 2, 3 ], [ 1, 3, 4 ] ) );;
gap> IsomorphismBlockBijectionSemigroup(S);
MappingByFunction( <partial perm semigroup of rank 3 with 2
 generators>, <bipartition semigroup of degree 5 with 2 generators>
 , function( x ) ... end, function( x ) ... end )
```

2.4.5 IsomorphismMatrixSemigroup

▷ `IsomorphismMatrixSemigroup(S[, F])` (attribute)

Returns: An isomorphism to a matrix semigroup.

This attribute contains an isomorphism from the semigroup S to a matrix semigroup. Currently this is done by taking a standard basis of a vector space suitable dimension and acting on this basis over the field F if F is given, and over `GF(2)` if F is not given. This will not give an optimal matrix semigroup representation of S.

Example
```
gap> T := Semigroup(Transformation([2, 2, 3]), Transformation([3, 1, 3]));
<transformation semigroup of degree 3 with 2 generators>
gap> iso := IsomorphismMatrixSemigroup(T);
MappingByFunction( <transformation semigroup of degree 3 with 2
 generators>, <semigroup of 3x3 matrices over GF(2)
 with 2 generators>, function( x ) ... end, function( x ) ... end )
gap> Size(Range(iso));
5
```

2.5 Standard examples

In this section, we describe the operations in Semigroups that can be used to creating semigroups belonging to several standard classes of example. See Chapter 5 for more information about semigroups of bipartitions.

2.5.1 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
```

2.5.2 PartitionMonoid

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

Returns: A bipartition monoid.

If `n` is a positive integer, then this operation returns the partition monoid of degree `n` which 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).

Example
```
gap> S:=PartitionMonoid(5);
<regular bipartition monoid of degree 5 with 4 generators>
gap> Size(S);
115975
```

2.5.3 BrauerMonoid

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

Returns: A bipartition monoid.

If `n` is a positive integer, then this operation returns the Brauer monoid of degree `n`. The BRAUER MONOID is the subsemigroup of the partition monoid consisiting of those bipartitions where the size of every block is 2.

`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>
```

2.5.4 JonesMonoid

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

Returns: A bipartition monoid.

If `n` is a positive 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 with a planar diagram. 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> SingularJonesMonoid(8);
<regular bipartition semigroup ideal of degree 8 with 1 generator>
```

2.5.5 FactorisableDualSymmetricInverseSemigroup

▷ `FactorisableDualSymmetricInverseSemigroup(n)` (operation)
▷ `SingularFactorisableDualSymmetricInverseSemigroup(n)` (operation)

Returns: An inverse bipartition monoid.

If `n` is a positive integer, then this operation returns the largest factorisable inverse subsemigroup of the dual symmetric inverse monoid of degree `n`.

`SingularFactorisableDualSymmetricInverseSemigroup` returns the ideal of the factorisable dual symmetric inverse semigroup consisting of the non-invertible elements (i.e. those not in the group of units), when `n` is at least 2.

See `IsUniformBlockBijection` (5.5.14).

Example
```
gap> S:=DualSymmetricInverseMonoid(4);
<inverse bipartition monoid of degree 4 with 3 generators>
gap> IsFactorisableSemigroup(S);
false
gap> S:=FactorisableDualSymmetricInverseSemigroup(4);
<inverse bipartition monoid of degree 4 with 3 generators>
gap> IsFactorisableSemigroup(S);
true
gap> S:=Range(IsomorphismBipartitionSemigroup(SymmetricInverseMonoid(5)));
<inverse bipartition monoid of degree 5 with 3 generators>
gap> IsFactorisableSemigroup(S);
true
```

2.5.6 DualSymmetricInverseSemigroup

▷ `DualSymmetricInverseSemigroup(n)` (operation)
▷ `DualSymmetricInverseMonoid(n)` (operation)
▷ `SingularDualSymmetricInverseSemigroup(n)` (operation)

Returns: An inverse bipartition monoid.

If `n` is a positive integer, then these operations 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`.

`SingularDualSymmetricInverseSemigroup` 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.

See `IsBlockBijection` (5.5.13).

Example
```
gap> Number(PartitionMonoid(3), IsBlockBijection);
25
gap> S:=DualSymmetricInverseSemigroup(3);
<inverse bipartition monoid of degree 3 with 3 generators>
gap> Size(S);
25
```

2.5.7 PartialTransformationSemigroup

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

2.5.8 FullMatrixSemigroup

▷ `FullMatrixSemigroup(d, q)` (operation)
▷ `GeneralLinearSemigroup(d, q)` (operation)
▷ `GLS(d, q)` (operation)

Returns: A matrix semigroup.

`FullMatrixSemigroup`, `GeneralLinearSemigroup`, and `GLS` are synonyms for each other. They both return the full matrix semigroup, or if you prefer the general linear semigroup, of `d` by `d` matrices with entries over the field with `q` elements. This semigroup has `q ^ (d ^ 2)` elements.

Example
```
gap> S := FullMatrixSemigroup(3, 4);
<general linear monoid 3x3 over GF(2^2)>
gap> Size(S);
262144
```

2.5.9 SpecialLinearSemigroup

▷ `SpecialLinearSemigroup(d, q)` (operation)
▷ `SLS(d, q)` (operation)

Returns: A matrix semigroup.

`SpecialLinearSemigroup` and `SLS` are synonymous. The special linear semigroup of *d* by *d* matrices with entries over the field with *q* elements is generated by a generating set for the special linear group of *d* by *d* matrices over the field with *q* elements and a matrix of rank *d-1*.

Example

```
gap> S := SLS(3,4);
<special linear monoid 3x3 over GF(2^2)>
gap> Size(S);
141184
```

2.5.10 MunnSemigroup

▷ `MunnSemigroup(S)` (operation)

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 Peresse (St Andrews), Yanhui Wang (York).

PLEASE NOTE: the Grape package version 4.5 or higher must be available and compiled for this function to work.

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 := InverseSemigroup(Idempotents(S), rec(small := true));;
gap> M := MunnSemigroup(T);;
gap> NrIdempotents(M);
60
gap> NrIdempotents(S);
60
```

2.5.11 Monoids of order preserving functions

▷ `OrderEndomorphisms(n)` (operation)
▷ `POI(n)` (operation)
▷ `POPI(n)` (operation)

Returns: A semigroup of transformations or partial permutations 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:

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

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

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.

POPI(*n*)

POPI(*n*) returns the inverse monoid of partial permutation 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.

Example

```
gap> S:=POPI(10);
<inverse partial perm monoid of rank 10 with 2 generators>
gap> Size(S);
923781
gap> 1+5*Binomial(20, 10);
923781
gap> S:=POI(10);
<inverse partial perm monoid of rank 10 with 10 generators>
gap> Size(S);
184756
gap> Binomial(20,10);
184756
gap> IsSubsemigroup(POPI(10), POI(10));
true
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
```

2.5.12 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
```

2.5.13 RegularBinaryRelationSemigroup

▷ `RegularBinaryRelationSemigroup(n)` (operation)

Returns: A semigroup of binary relations.

`RegularBinaryRelationSemigroup` return the semigroup generated by the regular binary relations on the set $\{1,\dots,n\}$ for a positive integer `n`. `RegularBinaryRelationSemigroup(n)` is generated by the 4 binary relations:

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

This semigroup has nearly $2^{(n^2)}$ elements.

2.5.14 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 category 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`.

The semigroup `S` is generated by a single element, f. `S` consists of the elements $f, f^2, \dots, f^m, \dots, f^{m+r-1}$. The minimal ideal of `S` consists of the elements $f^m, \dots, f^{m+r-1}$ and is isomorphic to the cyclic group of order r.

See `IsMonogenicSemigroup` (4.6.10) 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);
<commutative simple transformation semigroup ideal of degree 8 with
  1 generator>
gap> IsGroupAsSemigroup(I);
```

```
true
gap> StructureDescription(I);
"C3"
gap> S := MonogenicSemigroup(IsBlockBijectionSemigroup, 9, 1);
<commutative non-regular bipartition semigroup of size 9, degree 10
 with 1 generator>
```

2.5.15 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, which is in the category given by the filter *filt*.

The optional argument *filt* may be one of the following:

- `IsTransformationSemigroup`,
- `IsBipartitionSemigroup`,
- `IsReesMatrixSemigroup` (the default, if *filt* is not specified).

See `IsRectangularBand` (4.6.13) for more information about rectangular bands.

Example
```
gap> S := RectangularBand(4, 8);
<Rees matrix semigroup 4x8 over Group(())>
gap> IsRectangularBand(S);
true
gap> IsCompletelySimpleSemigroup(S) and IsHTrivial(S);
true
gap> T := RectangularBand(IsTransformationSemigroup, 5, 6);
<transformation semigroup of size 30, degree 31 with 6 generators>
gap> IsRectangularBand(T);
true
```

2.5.16 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 category given by the filter *filt*.

The optional argument *filt* may be one of the following:

- `IsTransformationSemigroup`,
- `IsPartialPermSemigroup` (the default, if *filt* is not specified),
- `IsBipartitionSemigroup`,
- `IsBlockBijectionSemigroup`,
- `IsReesZeroMatrixSemigroup` (provided that $n > 1$).

See IsZeroSemigroup (4.6.23) for more information about zero semigroups.

Example
```
gap> S := ZeroSemigroup(15);
<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
gap> S := ZeroSemigroup(IsReesZeroMatrixSemigroup, 5);
<Rees 0-matrix semigroup 4x1 over Group(())>
gap> Matrix(S);
[ [ 0, 0, 0, 0 ] ]
gap> IsZeroSemigroup(S);
true
```

2.5.17 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 category given by the filter *filt*.

The optional argument *filt* may be one of the following:

- IsTransformationSemigroup (the default, if *filt* is not specified),
- IsBipartitionSemigroup,
- IsReesMatrixSemigroup.

See IsLeftZeroSemigroup (4.6.9) and IsRightZeroSemigroup (4.6.15) for more information about left and right zero semigroups.

Example
```
gap> S := LeftZeroSemigroup(20);
<transformation semigroup of size 20, degree 21 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);
<bipartition semigroup of size 5, degree 3 with 5 generators>
gap> IsRightZeroSemigroup(S);
true
```

Chapter 3

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 transformation, partial permutation, or bipartition semigroup, or an ideal of a subsemigroup of a Rees 0-matrix semigroup or semigroup of matrices over a finite field, that is created by the function `SemigroupIdeal` (3.1.1) 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.

3.1 Creating ideals

3.1.1 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>
```

3.2 Attributes of ideals

3.2.1 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);
<regular bipartition semigroup ideal of degree 4 with 1 generator>
gap> GeneratorsOfSemigroupIdeal(I);
[ <bipartition: [ 1, 2, 3, 4, -4 ], [ -1 ], [ -2 ], [ -3 ]> ]
gap> I=Semigroup(GeneratorsOfSemigroupIdeal(I));
false
```

3.2.2 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);
<non-regular bipartition semigroup ideal of degree 4 with 3 generators
 >
gap> MinimalIdealGeneratingSet(I);
[ <block bijection: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ], [ 4, -4 ]> ]
```

3.2.3 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` (3.1.1) or `SemigroupByGenerators`. This function returns the semigroup containing `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 := DClassNC(S, x);
<Green's D-class: Transformation( [ 2, 6, 7, 2, 6, 1, 1, 5 ] )>
gap> R := PrincipalFactor(D);
<Rees 0-matrix semigroup 1050x56 over Group([ (3,4), (2,8,7,4,3) ])>
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 4

Determining the structure of a semigroup

In this chapter we describe the functions in Semigroups for determining the structure of a semigroup, in particular for computing Green's classes and related properties of semigroups.

4.1 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.

4.1.1 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` returns the equivalent of:

Example
```
Product(List(w, i-> gens[i]));
```

see also `Factorization` (4.1.2).

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> f:=Transformation( [ 1, 4, 6, 1, 7, 2, 7, 6 ] );;
gap> Factorization(S, f);
[ 4, 2 ]
gap> EvaluateWord(gens, last);
Transformation( [ 1, 4, 6, 1, 7, 2, 7, 6 ] )
gap> S:=SymmetricInverseMonoid(10);;
gap> f:=PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] );
[3,7][8,1,2,6,9][10,5]
gap> Factorization(S, f);
[ -2, -2, -2, -2, -3, -4, -3, -2, -2, -2, -2, -3, -2, 2, 2, 2, 2, 4,
  4, 4, 4, 2, 2, 2, 2, 2, 3, 4, -3, -2, -3, -2, -3, -2, 2, 2, 2, 2,
  2, 3, 4, -3, -2, -3, -2, -3, -2, 2, 2, 2, 2, 2, 3, 4, -3, -2, -3,
  -2, -3, -2, 2, 2, 2, 2, 2, 3, 4, -3, -2, -3, -2, -3, -2, 2, 2, 2,
  2, 2, 3, 4, -3, -2, -3, -2, -3, -2, 3, 2, 2, 2, 2, 2, 3, 4, -3, -2,
  -3, -2, -3, -2, 2, 3, 2, 3, 2, 2, 2, 3, 2, 2, 2, 2, 2, 3, 2, 3, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), last);
[3,7][8,1,2,6,9][10,5]
```

4.1.2 Factorization

▷ `Factorization(S, f)` (operation)

Returns: A word in the generators.

for semigroups

When *S* is a semigroup and *f* belongs to *S*, `Factorization` return a word in the generators of *S* that is equal to *f*. In this case, a word is a list of positive integers where `i` corresponds to `GeneratorsOfSemigroups(S)[i]`. More specifically,

Example
```
EvaluateWord(GeneratorsOfSemigroup(S), Factorization(S, f))=f;
```

for inverse semigroups

When *S* is a inverse semigroup and *f* belongs to *S*, `Factorization` return a word in the generators of *S* that is equal to *f*. In this case, a word is a list of non-zero integers where `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, f))=f;
```

Note that `Factorization` does not return a word of minimum length.

See also `EvaluateWord` (4.1.1) 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> f:=Transformation( [ 1, 10, 2, 10, 1, 2, 7, 10, 2, 7 ] );;
gap> Factorization(S, f);
[ 2, 2, 1, 2 ]
gap> EvaluateWord(gens, last);
Transformation( [ 1, 10, 2, 10, 1, 2, 7, 10, 2, 7 ] )
gap> S:=SymmetricInverseMonoid(8);
<symmetric inverse monoid of degree 8>
gap> f:=PartialPerm( [ 1, 2, 3, 4, 5, 8 ], [ 7, 1, 4, 3, 2, 6 ] );
[5,2,1,7][8,6](3,4)
gap> Factorization(S, f);
[ -2, -2, -2, -2, -2, -2, -2, 2, 2, 4, 4, 2, 3, 2, 3, -2, -2, -2, 2,
  3, 2, 3, -2, -2, -2, 2, 3, 2, 3, -2, -2, -2, 3, 2, 3, 2, 3, -2, -2,
  -2, 3, 2, 3, 2, 3, -2, -2, -2, 2, 3, 2, 3, -2, -2, -2, 2, 3, 2, 3,
  -2, -2, -2, 2, 3, 2, 3, -2, -2, -2, 2, 3, 2, 3, -2, -2, -2, 3, 2,
  3, 2, 3, -2, -2, -2, 2, 3, 2, 3, -2, -2, -2, 2, 3, 2, 3, -2, -2,
  -2, 2, 3, 2, 3, -2, -2, -2, 2, 3, 2, 2, 3, 2, 2, 2, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), last);
[5,2,1,7][8,6](3,4)
gap> S:=DualSymmetricInverseMonoid(6);;
gap> f:=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> Factorization(S, f);
[ -2, -2, -2, -2, -2, 4, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), last);
<block bijection: [ 1, 6, -4 ], [ 2, -2, -3 ], [ 3, -5 ], [ 4, -6 ],
 [ 5, -1 ]>
```

4.2 Creating Green's classes

4.2.1 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

```
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` (4.2.3).

```
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([ 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 ]));
<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);
<Green's L-class: <identity partial perm on [ 1, 2, 5, 6, 7, 9 ]>>
gap> DClass(R) = DClass(L);
true
gap> DClass(H) = DClass(L);
true
```

4.2.2 GreensXClassOfElement

▷ `GreensDClassOfElement(X, f)` (operation)
▷ `DClass(X, f)` (function)
▷ `GreensHClassOfElement(X, f)` (operation)
▷ `GreensHClassOfElement(R, i, j)` (operation)
▷ `HClass(X, f)` (function)
▷ `HClass(R, i, j)` (function)
▷ `GreensLClassOfElement(X, f)` (operation)
▷ `LClass(X, f)` (function)
▷ `GreensRClassOfElement(X, f)` (operation)
▷ `RClass(X, f)` (function)

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.

4.2.3 GreensXClassOfElementNC

▷ `GreensDClassOfElementNC(X, f)` (operation)
▷ `DClassNC(X, f)` (function)
▷ `GreensHClassOfElementNC(X, f)` (operation)
▷ `HClassNC(X, f)` (function)
▷ `GreensLClassOfElementNC(X, f)` (operation)
▷ `LClassNC(X, f)` (function)
▷ `GreensRClassOfElementNC(X, f)` (operation)
▷ `RClassNC(X, f)` (function)

Returns: A Green's class.

These functions are essentially the same as `GreensDClassOfElement` (4.2.2) 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, 2
```

in the semigroup *X* of order-preserving mappings on 20 points, then

Example
```
GreensRClassOfElementNC(X, x);
```

returns an answer relatively quickly, whereas

```
GreensRClassOfElement(X, x)
```

can take a signficant amount of time to return a value.

See also GreensRClassOfElement (**Reference: GreensRClassOfElement**) and RClassOfHClass (4.2.1).

```
gap> S := RandomTransformationSemigroup(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([ 1, 2, 3, 4, 7, 8, 9, 10 ],
> [ 2, 3, 4, 5, 6, 8, 10, 11 ]);;
gap> L := LClass(POI(13), x);
<Green's L-class: [1,2,3,4,5,6,8,11][7,10]>
gap> Size(L);
1287
```

4.2.4 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**), IsRegularClass (4.4.4), and IsRegularSemigroup (4.6.14).

```
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( [ 1, 2, 3, 5 ], [ 2, 1, 6, 3 ] ),
> PartialPerm( [ 1, 2, 3, 6 ], [ 3, 5, 2, 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"
```

4.3 Iterators and enumerators of classes and representatives

4.3.1 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 (4.3.4), IteratorOfDClassReps (4.3.2), IteratorOfDClasses (4.3.3), and NrDClasses (4.4.6).

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( [ 1, 2, 3 ], [ 2, 4, 1 ] ),
> PartialPerm( [ 1, 3, 4 ], [ 3, 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 [ 2, 4 ]>>,
  <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 [ 2, 4 ]>>,
  <Green's L-class: [2,3][4,1]>, <Green's L-class: [4,2,1]>,
  <Green's L-class: [4,2,3]>, <Green's L-class: [2,4,3]>,
  <Green's L-class: [2,1](4)>,
  <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 [ 2, 4 ]>>
gap> GreensLClasses(D);
```

```
[ <Green's L-class: <identity partial perm on [ 2, 4 ]>>,
  <Green's L-class: [2,3][4,1]>, <Green's L-class: [4,2,1]>,
  <Green's L-class: [4,2,3]>, <Green's L-class: [2,4,3]>,
  <Green's L-class: [2,1](4)> ]
gap> GreensRClasses(D);
[ <Green's R-class: <identity partial perm on [ 2, 4 ]>>,
  <Green's R-class: [1,4][3,2]>, <Green's R-class: [1,2,4]>,
  <Green's R-class: [3,2,4]>, <Green's R-class: [3,4,2]>,
  <Green's R-class: [1,2](4)> ]
```

4.3.2 IteratorOfXClassReps

▷ `IteratorOfDClassReps(S)` (function)
▷ `IteratorOfHClassReps(S)` (function)
▷ `IteratorOfLClassReps(S)` (function)
▷ `IteratorOfRClassReps(S)` (function)

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` (4.3.1), and `IteratorOfRClasses` (4.3.3).

Example

```
gap> gens := [ 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> S := Semigroup(gens);;
gap> iter := IteratorOfRClassReps(S);
<iterator>
gap> NextIterator(iter);
Transformation( [ 3, 2, 1, 5, 4 ] )
gap> NextIterator(iter);
Transformation( [ 5, 5, 4, 5, 1 ] )
gap> iter;
<iterator>
gap> file := Concatenation(SemigroupsDir(), "/tst/test.gz");;
gap> S := InverseSemigroup(ReadGenerators(file, 1377));
<inverse partial perm semigroup of rank 983 with 2 generators>
gap> NrMovedPoints(S);
983
gap> iter := IteratorOfLClassReps(S);
<iterator>
gap> NextIterator(iter);
<partial perm on 634 pts with degree 1000, codegree 1000>
```

4.3.3 IteratorOfXClasses

▷ `IteratorOfDClasses(S)` (function)
▷ `IteratorOfHClasses(S)` (function)
▷ `IteratorOfLClasses(S)` (function)

▷ `IteratorOfRClasses(S)` (function)

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` (4.3.1), `GreensRClasses` (**Reference: GreensRClasses**), `NrRClasses` (4.4.6), and `IteratorOfRClassReps` (4.3.2).

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>
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>
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
```

4.3.4 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 (4.3.1), IteratorOfDClassReps (4.3.2), IteratorOfDClasses (4.3.3), and NrDClasses (4.4.6).

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> DClassReps(M);
[ <identity partial perm on [ 51 ]>,
  <identity partial perm on [ 27, 51 ]>,
  <identity partial perm on [ 15, 27, 50, 51 ]>,
  <identity partial perm on [ 8, 15, 26, 27, 49, 50, 51, 64 ]>,
  <identity partial perm on
    [ 4, 8, 14, 15, 25, 26, 27, 48, 49, 50, 51, 60, 61, 62, 63, 64 ]>,
  <identity partial perm on
    [ 2, 4, 7, 8, 13, 14, 15, 21, 25, 26, 27, 29, 34, 39, 44, 48, 49, \
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64 ]>,
  <identity partial perm on
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1\
9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,\
 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 5\
4, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64 ]> ]
gap> L := LClassNC(M, PartialPerm( [ 51, 63 ], [ 51, 47 ] ));;
gap> HClassReps(L);
[ <identity partial perm on [ 47, 51 ]>, [27,47](51), [50,47](51),
  [59,47](51), [63,47](51), [64,47](51) ]
```

4.4 Attributes and properties directly related to Green's classes

4.4.1 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 ], [ 1, 3, 4 ]) );
<Green's D-class: <identity partial perm on [ 1, 2, 3 ]>>
gap> B := GreensJClassOfElement(S, PartialPerm([ 1, 2 ], [ 3, 1 ]) );
<Green's D-class: <identity partial perm on [ 1, 2 ]>>
```

```
gap> A < B;
false
gap> B < A;
true
gap> IsGreensLessThanOrEqual(A, B);
false
gap> IsGreensLessThanOrEqual(B, A);
true
```

4.4.2 InjectionPrincipalFactor

▷ `InjectionPrincipalFactor(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 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` returns an error.

See also `PrincipalFactor` (4.4.3).

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> InjectionPrincipalFactor(d);;
gap> rms := Range(last);
<Rees 0-matrix semigroup 3x3 over Group(())>
gap> MatrixOfReesZeroMatrixSemigroup(rms);
[ [ (), 0, 0 ], [ 0, (), 0 ], [ 0, 0, () ] ]
gap> Size(rms);
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 := DClasses(S)[3];
<Green's D-class: <bipartition: [ 1, 5, -2, -4 ], [ 2, 3, 4, -1, -5 ]
   , [ -3 ]>>
gap> inj := InjectionPrincipalFactor(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( f ) ... end, function( x ) ... end )
```

4.4.3 PrincipalFactor

▷ `PrincipalFactor(D)` (attribute)

Returns: A Rees matrix semigroup.

`PrincipalFactor(D)` is just shorthand for `Range(InjectionPrincipalFactor(D))`, where *D* is a $\mathscr{D}$-class of semigroup; see `InjectionPrincipalFactor` (4.4.2) 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> PrincipalFactor(D);
<Rees matrix semigroup 1x5 over Group(())>
```

4.4.4 IsRegularClass

▷ `IsRegularClass(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` (4.2.4), `NrIdempotents` (4.5.4), `Idempotents` (4.5.3), 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> IsRegularClass(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> IsRegularClass(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> IsRegularClass(L);
false
gap> R := GreensRClassOfElement(S, f);;
gap> IsRegularClass(R);
false
gap> g := Transformation( [ 4, 4, 4, 4 ] );;
gap> IsRegularSemigroupElement(S, g);
true
gap> IsRegularClass(LClass(S, g));
true
gap> IsRegularClass(RClass(S, g));
true
gap> IsRegularDClass(DClass(S, g));
true
gap> DClass(S, g)=RClass(S, g);
true
```

4.4.5 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 IsRegularClass (4.4.4) 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> RegularDClasses(S);
[ <Green's D-class: Transformation( [ 1, 4, 1, 1, 4, 3 ] )>,
  <Green's D-class: Transformation( [ 1, 1, 1, 1, 1, 4 ] )>,
```

```
<Green's D-class: Transformation( [ 1, 1, 1, 1, 1, 1 ] )> ]
```

4.4.6 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 (4.3.1), GreensRClasses (**Reference: GreensRClasses**), IteratorOfRClasses (4.3.3), and IteratorOfRClassReps (4.3.2).

Example
```
gap> gens := [ 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> S := Semigroup(gens);;
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> gens := [ 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> S := Semigroup(gens);;
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
```

4.4.7 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` (4.3.1), `GreensDClasses` (**Reference: GreensDClasses**), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), and \< (4.4.1).

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
```

4.4.8 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.

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([ (1,9,8), (8,
   9) ]), Group([ (4,9) ]), Group(()), Group(()), Group(()),
  Group(()), Group(()), Group(()), Group(()), Group(()), Group(()),
  Group(()), Group([ (2,5)(3,7) ]), Group([ (1,7,5,6,9,3) ]),
  Group(()), Group(()), Group(()), Group(()), Group(()) ]
```

4.4.9 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` (4.4.7), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), `MinimalIdeal` (4.5.10) and `RepresentativeOfMinimalIdeal` (4.5.11).

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>>
```

4.4.10 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` (4.4.7), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), and `MinimalDClass` (4.4.9).

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)> ]
```

4.4.11 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" ]
```

4.4.12 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 bipartition 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" ]
```

4.4.13 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, 2, 3 ], [ 1, 5, 2 ] ),
> PartialPerm( [ 1, 3 ], [ 2, 4 ] ),
> PartialPerm( [ 1, 2, 3 ], [ 4, 1, 5 ] ),
> PartialPerm( [ 1, 3, 5 ], [ 1, 3, 4 ] ),
> PartialPerm( [ 1, 2, 4, 5 ], [ 1, 2, 3, 5 ] ),
> PartialPerm( [ 1, 2, 3, 5 ], [ 1, 3, 2, 5 ] ),
> PartialPerm( [ 1, 4, 5 ], [ 5, 4, 3 ] ) );;
gap> H := HClass(S, PartialPerm( [ 1, 2 ], [ 1, 2 ] ));;
gap> MultiplicativeNeutralElement(H);
<identity partial perm on [ 1, 2 ]>
gap> H := HClass(S, PartialPerm( [ 1, 2 ], [ 1, 4 ] ));;
gap> MultiplicativeNeutralElement(H);
fail
```

4.4.14 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.

4.4.15 IsTransformationSemigroupGreensClass

▷ `IsTransformationSemigroupGreensClass(`*class*`)` (property)

Returns: `true` or `false`.

A Green's class *class* of a semigroup S satisfies the property `IsTransformationSemigroupGreensClass` if and only if S is a semigroup of transformations.

4.4.16 IsBipartitionSemigroupGreensClass

▷ `IsBipartitionSemigroupGreensClass(`*class*`)` (property)

Returns: `true` or `false`.

A Green's class *class* of a semigroup S satisfies the property `IsBipartitionSemigroupGreensClass` if and only if S is a semigroup of bipartitions.

4.4.17 IsPartialPermSemigroupGreensClass

▷ `IsPartialPermSemigroupGreensClass(`*class*`)` (property)

Returns: `true` or `false`.

A Green's class *class* of a semigroup S satisfies the property `IsPartialPermSemigroupGreensClass` if and only if S is a semigroup of partial perms.

4.4.18 IsMatrixSemigroupGreensClass

▷ `IsMatrixSemigroupGreensClass(class)` (property)

Returns: `true` or `false`.

A Green's class *class* of a semigroup `S` satisfies the property `IsMatrixSemigroupGreensClass` if and only if `S` is belongs to the category `IsMatrixSemigroup`.

4.4.19 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"
```

4.4.20 IsGreensDLeq

▷ `IsGreensDLeq(S)` (attribute)

Returns: A function.

`IsGreensDLeq(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(DClassReps(S));
[ Transformation( [ 1, 3, 4, 1, 3 ] ),
  Transformation( [ 2, 4, 1, 5, 5 ] ),
  Transformation( [ 1, 4, 1, 1, 4 ] ),
  Transformation( [ 1, 1, 1, 1, 1 ] ) ]
gap> Sort(reps, IsGreensDLeq(S));
gap> reps;
[ Transformation( [ 2, 4, 1, 5, 5 ] ),
  Transformation( [ 1, 3, 4, 1, 3 ] ),
  Transformation( [ 1, 4, 1, 1, 4 ] ),
  Transformation( [ 1, 1, 1, 1, 1 ] ) ]
gap> IsGreensLessThanOrEqual(DClass(S, reps[2]), DClass(S, reps[1]));
true
gap> S := DualSymmetricInverseMonoid(4);;
gap> IsGreensDLeq(S)(S.1, S.3);
true
```

```
gap> IsGreensDLeq(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
```

4.5 Further attributes of semigroups

In this section we describe the attributes of a semigroup that can be found using the Semigroups package.

4.5.1 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` (3.2.1).

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 ] ) ]
```

4.5.2 GroupOfUnits

▷ `GroupOfUnits(S)` (attribute)

Returns: The group of units of a semigroup.

`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` (2.4.2) 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` (4.6.11), 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"
```

4.5.3 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**), `IsRegularClass` (4.4.4) `IsGroupHClass` (**Reference: IsGroupHClass**), `NrIdempotents` (4.5.4), and `GroupHClass` (4.2.4).

Example

```
gap> S := Semigroup([ Transformation( [ 2, 3, 4, 1 ] ),
> Transformation( [ 3, 3, 1, 1 ] ) ]);;
gap> Idempotents(S, 1);
[  ]
gap> Idempotents(S, 2);
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 3, 3, 1 ] ),
  Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> Idempotents(S);
[ IdentityTransformation, Transformation( [ 1, 1, 3, 3 ] ),
  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);
<Green's R-class: Transformation( [ 3, 3, 1, 1 ] )>
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> Idempotents(L);
[ Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> D := DClassOfLClass(L);
<Green's D-class: Transformation( [ 1, 1, 3, 3 ] )>
gap> Idempotents(D);
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 2, 2, 4, 4 ] ),
  Transformation( [ 1, 3, 3, 1 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> L := GreensLClassOfElement(S, Transformation( [ 3, 1, 1, 3 ] ));;
gap> 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( [ 1, 2, 3, 4, 5, 7 ], [ 10, 6, 3, 4, 9, 1 ] ),
> PartialPerm( [ 1, 2, 3, 4, 5, 6, 7, 8 ],
> [ 6, 10, 7, 4, 8, 2, 9, 1 ] ) ]);;
gap> Idempotents(S, 1);
[ <identity partial perm on [ 4 ]> ]
gap> Idempotents(S, 0);
[  ]
```

4.5.4 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` (4.5.3), `IsRegularDClass` (**Reference: IsRegularDClass**), `IsRegularClass` (4.4.4) `IsGroupHClass` (**Reference: IsGroupHClass**), and `GroupHClass` (4.2.4).

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
```

4.5.5 IdempotentGeneratedSubsemigroup

▷ `IdempotentGeneratedSubsemigroup(S)` (attribute)

Returns: A semigroup.

`IdempotentGeneratedSubsemigroup` returns the subsemigroup of the semigroup *S* generated by the idempotents of *S*.

See also `Idempotents` (4.5.3) and `SmallGeneratingSet` (4.5.14).

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);
<transformation monoid of degree 8 with 18 generators>
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 bipartition monoid of degree 5 with 3 generators>
gap> IdempotentGeneratedSubsemigroup(S);
<inverse bipartition monoid of degree 5 with 10 generators>
gap> IsSemilattice(last);
true
```

4.5.6 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(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 := RandomBipartitionSemigroup(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 ]> ]
```

4.5.7 MaximalSubsemigroups (for an acting semigroup)

▷ `MaximalSubsemigroups(S)` (attribute)

Returns: The maximal subsemigroups of S.

If S is a semigroup, then `MaximalSubsemigroups` returns a list of the maximal subsemigroups of S.

A *maximal subsemigroup* of S is a proper subsemigroup of S which is contained in no other proper subsemigroups of S.

The method for this function are based on [GGR68].

PLEASE NOTE: the Grape package version 4.5 or higher must be available and compiled for this function to work.

Example

```
gap> S := FullTransformationSemigroup(4);
<full transformation monoid of degree 4>
gap> MaximalSubsemigroups(S);
[ <transformation semigroup of degree 4 with 3 generators>,
  <transformation semigroup of degree 4 with 4 generators>,
  <transformation semigroup of degree 4 with 4 generators>,
  <transformation semigroup of degree 4 with 4 generators>,
  <transformation semigroup of degree 4 with 5 generators>,
  <transformation semigroup of degree 4 with 4 generators>,
  <transformation semigroup of degree 4 with 5 generators>,
  <transformation semigroup of degree 4 with 5 generators>,
  <transformation semigroup of degree 4 with 4 generators> ]
gap> D := DClass(S, Transformation([ 2, 2 ]));
<Green's D-class: Transformation( [ 2, 3, 1, 2 ] )>
gap> R := PrincipalFactor(D);
<Rees 0-matrix semigroup 6x4 over Group([ (1,2,3), (1,2) ])>
gap> MaximalSubsemigroups(R);
[ <Rees 0-matrix semigroup 6x3 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 6x3 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 6x3 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 6x3 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 5x4 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 5x4 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 5x4 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 5x4 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 5x4 over Group([ (1,2,3), (1,2) ])>,
  <Rees 0-matrix semigroup 5x4 over Group([ (1,2,3), (1,2) ])>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 23 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 23 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 21 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 23 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 21 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 21 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 23 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 21 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 21 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 21 generators> ]
```

4.5.8 MaximalSubsemigroups (for a Rees (0-)matrix semigroup, and a group)

▷ `MaximalSubsemigroups(R, H)` (operation)

Returns: The maximal subsemigroups of a Rees (0)-matrix semigroup corresponding to a maximal subgroup of the underlying group.

Suppose that R is a regular Rees (0-)matrix semigroup of the form $\mathscr{M}[G;I,J;P]$ where G is a group and P is a $|J|$ by $|I|$ matrix with entries in $G \cup \{0\}$. If H is a maximal subgroup of G, then this function returns the maximal subsemigroups of R which are isomorphic to $\mathscr{M}[H;I,J;P]$.

The method used in this function is based on Remark 1 of [GGR68].

PLEASE NOTE: the Grape package version 4.5 or higher must be available and compiled for this function to work, when the argument R is a Rees 0-matrix semigroup.

Example
```
gap> R := ReesZeroMatrixSemigroup(Group([ (1,2), (3,4) ]),
> [ [ (), (1,2) ], [ (), (1,2) ] ]);
<Rees 0-matrix semigroup 2x2 over Group([ (1,2), (3,4) ])>
gap> G := UnderlyingSemigroup(R);
Group([ (1,2), (3,4) ])
gap> H := Group((1,2));
Group([ (1,2) ])
gap> max := MaximalSubsemigroups(R, H);
[ <subsemigroup of 2x2 Rees 0-matrix semigroup with 6 generators> ]
gap> IsMaximalSubsemigroup(R, max[1]);
true
```

4.5.9 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 proper subsemigroup T of a semigroup S is a *maximal* if T is not contained in any other proper subsemigroups of S.

Example
```
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
```

4.5.10 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 (4.5.11), PartialOrderOfDClasses (4.4.7), IsGreensLessThanOrEqual (**Reference: IsGreensLessThanOrEqual**), and MinimalDClass (4.4.9).

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);
<Green's D-class: Transformation( [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ] )>
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>
```

4.5.11 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 (4.5.10) and MinimalDClass (4.4.9).

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 ] )>
```

4.5.12 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 degree 6 with 4 generators>
gap> MultiplicativeZero(S);
fail
gap> S := DualSymmetricInverseMonoid(6);
<inverse bipartition monoid of degree 6 with 3 generators>
gap> MultiplicativeZero(S);
<block bijection: [ 1, 2, 3, 4, 5, 6, -1, -2, -3, -4, -5, -6 ]>
```

4.5.13 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.

4.5.14 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` (4.5.1).

As neither irredundancy, nor minimal length are proven, these functions usually return an answer much more quickly than `IrredundantGeneratingSubset` (4.5.6). 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(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(G));;
gap> Append(mat, [1..1000]*0);
gap> mat := List([1..138], x-> List([1..57], x-> Random(mat)));;
gap> R := ReesZeroMatrixSemigroup(G, 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 := RandomBipartitionSemigroup(100,4);
<bipartition semigroup of degree 4 with 96 generators>
```

```
gap> Length(SmallGeneratingSet(S));
13
```

4.5.15 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 ]
```

4.5.16 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 ] ]
```

4.5.17 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);
[ [ 1, 11, 12, 5, 4, 6, 10, 7, 9 ] ]
```

4.5.18 IsTransitive (for a transformation semigroup and a set)

▷ `IsTransitive(S[, X])` (operation)

▷ `IsTransitive(S[, n])` (operation)

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> AsTransformationSemigroup(S);
<transformation semigroup of degree 12 with 2 generators>
gap> IsTransitive(last);
false
gap> IsTransitive(AsSemigroup(Group((1,2,3))));
true
```

4.5.19 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 ]
```

4.5.20 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 ] ]
```

4.5.21 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( [ 1, 2, 3, 4, 5, 8, 10 ], [ 3, 1, 4, 2, 5, 6, 7 ] );
[8,6][10,7](1,3,4,2)(5)
gap> CyclesOfPartialPerm(x);
[ [ 3, 4, 2, 1 ], [ 5 ] ]
```

4.5.22 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, 20, 2, 19, 3, 8, 11 ] ]
```

4.5.23 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` and the genss package is loaded, 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`.

If genss is not loaded, then this option is ignored.

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( [ 1, 2, 3, 4, 5 ], [ 2, 5, 6, 3, 8 ] ),
> PartialPerm( [ 1, 2, 4, 7, 8 ], [ 3, 6, 2, 5, 7 ] ) );;
gap> Normalizer(S, rec(random:=true, lambdastab:=false));
#I  Have 33389 points.
#I  Have 40136 points in new orbit.
Group(())
```

4.5.24 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 ] )
```

4.5.25 GeneratorsSmallest (for a transformation semigroup)

▷ GeneratorsSmallest(*S*) (attribute)

Returns: A generating set of transformations.

GeneratorsSmallest returns the lexicographically least collection X of transformations such that *S* is generated by X and each X[i] is not generated by X[1], X[2], ..., X[i-1].

Note that it can be difficult to find this set of generators, and that it might contain a substantial proportion of the elements of the semigroup.

The comparison of two transformation semigroups via the lexicographic comparison of their sets of elements is the same relation as the lexicographic comparison of their GeneratorsSmallest. However, due to the complexity of determining the GeneratorsSmallest, this is not the method used by the Semigroups package when comparing transformation 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 ] ) ]
```

4.5.26 UnderlyingSemigroupOfSemigroupWithAdjoinedZero

▷ UnderlyingSemigroupOfSemigroupWithAdjoinedZero(*S*) (attribute)

Returns: A semigroup, or fail.

If *S* is a semigroup for which the property IsSemigroupWithAdjoinedZero (4.6.17) is true, (i.e. *S* has a MultiplicativeZero (4.5.12) 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 (4.6.17) 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
```

4.6 Further properties of semigroups

In this section we describe the properties of a semigroup that can be determined using the Semigroups package.

4.6.1 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 (4.6.18).

Example

```
gap> gens := [ 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> S := Semigroup(gens);;
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 degree 4 with 4 generators>
gap> M := MinimalIdeal(S);
<simple bipartition semigroup ideal of degree 4 with 1 generator>
```

```
gap> IsBand(M);
true
```

4.6.2 IsBlockGroup

▷ `IsBlockGroup(S)` (property)
▷ `IsSemigroupWithCommutingIdempotents(S)` (property)

Returns: `true` or `false`.

`IsBlockGroup` and `IsSemigroupWithCommutingIdempotents` return `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 := Range(IsomorphismBlockBijectionSemigroup(S));
<bipartition semigroup of degree 6 with 5 generators>
gap> IsBlockGroup(T);
true
gap> IsBlockGroup(Range(IsomorphismBipartitionSemigroup(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
```

4.6.3 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> gens := [ Transformation( [ 2, 4, 5, 3, 7, 8, 6, 9, 1 ] ),
>  Transformation( [ 3, 5, 6, 7, 8, 1, 9, 2, 4 ] ) ];;
gap> S := Semigroup(gens);;
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
```

4.6.4 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 (4.7.1).

Example
```
gap> gens := [ 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> S := Semigroup(gens);;
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
```

4.6.5 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
```

4.6.6 IsGroupAsSemigroup

▷ `IsGroupAsSemigroup(S)` (property)

Returns: `true` or `false`.

`IsGroupAsSemigroup` returns `true` if and only if the semigroup S is mathematically a group.

Example
```
gap> gens := [ Transformation( [ 2, 4, 5, 3, 7, 8, 6, 9, 1 ] ),
>  Transformation( [ 3, 5, 6, 7, 8, 1, 9, 2, 4 ] ) ];;
gap> S := Semigroup(gens);;
gap> IsGroupAsSemigroup(S);
true
gap> G := SymmetricGroup(5);;
gap> S := Range(IsomorphismPartialPermSemigroup(G));
<inverse partial perm semigroup of rank 5 with 2 generators>
gap> IsGroupAsSemigroup(S);
true
gap> S := SymmetricGroup([1,2,10]);;
gap> T := Range(IsomorphismBlockBijectionSemigroup(
> Range(IsomorphismPartialPermSemigroup(S))));
<inverse bipartition semigroup of degree 11 with 2 generators>
gap> IsGroupAsSemigroup(T);
true
```

4.6.7 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` (4.5.3) and `IdempotentGeneratedSubsemigroup` (4.5.5).

An inverse semigroup is idempotent-generated if and only if it is a semilattice; see `IsSemilattice` (4.6.18).

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
```

4.6.8 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` (4.6.6).

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
```

4.6.9 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> gens := [ Transformation( [ 2, 1, 4, 3, 5 ] ),
>  Transformation( [ 3, 2, 3, 1, 1 ] ) ];;
gap> S := Semigroup(gens);;
gap> IsRightZeroSemigroup(S);
false
gap> gens := [Transformation( [ 1, 2, 3, 3, 1 ] ),
> Transformation( [ 1, 2, 3, 3, 3 ] ) ];;
gap> S := Semigroup(gens);;
gap> IsLeftZeroSemigroup(S);
true
```

4.6.10 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 `IsMonogenicInverseSemigroup` (4.7.7) and `IndexPeriodOfTransformation` (**Reference: IndexPeriodOfTransformation**).

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
```

4.6.11 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 := Range(IsomorphismBipartitionSemigroup(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
```

4.6.12 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` (4.6.14) and `IsRegularSemigroup` (**Reference: IsRegularSemigroup**).

Example
```
gap> gens := [ 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> S := Semigroup(gens);;
```

```
gap> IsOrthodoxSemigroup(S);
true
gap> S := Semigroup(GeneratorsOfSemigroup(DualSymmetricInverseMonoid(5)));;
gap> IsOrthodoxSemigroup(S);
true
```

4.6.13 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> gens := [ 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> S := Semigroup(gens);;
gap> IsRectangularBand(S);
true
gap> IsRectangularBand(MinimalIdeal(PartitionMonoid(4)));
true
```

4.6.14 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**), IsRegularClass (4.4.4), and IsRegularSemigroupElement (**Reference: IsRegularSemigroupElement**).

Example
```
gap> IsRegularSemigroup(FullTransformationSemigroup(5));
true
gap> IsRegularSemigroup(JonesMonoid(5));
true
```

4.6.15 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> gens := [ Transformation( [ 2, 1, 4, 3, 5 ] ),
>  Transformation( [ 3, 2, 3, 1, 1 ] ) ];;
gap> S := Semigroup(gens);;
gap> IsRightZeroSemigroup(S);
false
gap> gens := [Transformation( [ 1, 2, 3, 3, 1 ] ),
>  Transformation( [ 1, 2, 4, 4, 1 ] )];;
gap> S := Semigroup(gens);;
gap> IsRightZeroSemigroup(S);
true
```

4.6.16 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 (4.6.18).

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
```

4.6.17 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 (4.5.12) 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
```

4.6.18 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 := FactorisableDualSymmetricInverseSemigroup(5);;
gap> S := IdempotentGeneratedSubsemigroup(S);;
gap> IsSemilattice(S);
true
```

4.6.19 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(())>
```

4.6.20 IsSynchronizingSemigroup

▷ IsSynchronizingSemigroup(*S*[, *n*]) (operation)
▷ IsSynchronizingTransformationCollection(*coll*[, *n*]) (operation)

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*.

The operation IsSynchronizingTransformationCollection behaves in the same way as IsSynchronizingSemigroup but can be applied to any collection of transformations and not only semigroups.

Note that the semigroup consisting of the identity transformation has degree 0, and for 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 ] )
```

4.6.21 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
```

4.6.22 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` (4.6.24) and `IsHTrivial` (4.6.16). An inverse semigroup is a 0-rectangular band if and only if it is a 0-group; see `IsZeroGroup` (4.6.21).

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> IsZeroRectangularBand(Semigroup(Elements(GreensDClasses(S)[7])));
true
gap> IsZeroRectangularBand(Semigroup(Elements(GreensDClasses(S)[1])));
false
```

4.6.23 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 ] )
```

4.6.24 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 (4.5.12). An inverse semigroup is 0-simple if and only if it is a Brandt semigroup; see IsBrandtSemigroup (4.7.2).

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
```

4.7 Properties and attributes of inverse semigroups

In this section we describe properties and attributes specific to inverse semigroups that can be determined using Semigroups.

The functions

- IsJoinIrreducible (4.7.5)
- IsMajorantlyClosed (4.7.6)
- JoinIrreducibleDClasses (4.7.8)
- MajorantClosure (4.7.9)
- Minorants (4.7.10)
- RightCosetsOfInverseSemigroup (4.7.12)

- `SmallerDegreePartialPermRepresentation` (4.7.14)
- `VagnerPrestonRepresentation` (4.7.15)

were written by Wilf Wilson and Robert Hancock.

The function `CharacterTableOfInverseSemigroup` (4.7.16) was written by Jhevon Smith and Ben Steinberg.

4.7.1 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
```

4.7.2 IsBrandtSemigroup

▷ `IsBrandtSemigroup(S)` (property)

Returns: `true` or `false`.

`IsBrandtSemigroup` return `true` if the semigroup *S* is a 0-simple inverse semigroup, and `false` if it is not. See also `IsZeroSimpleSemigroup` (4.6.24) 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 := DClasses(S)[3];
<Green's D-class: <block bijection: [ 1, 2, 3, -1, -2, -3 ],
  [ 4, -4 ]>>
gap> R := InjectionPrincipalFactor(D);;
gap> S := Semigroup(PreImages(R, GeneratorsOfSemigroup(Range(R))));;
gap> IsBrandtSemigroup(S);
true
```

4.7.3 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

$$s \in S \text{ and } e \in E\colon es \in E \Rightarrow s \in E.$$

Equivalently, `S` is *E-unitary* if `E` is closed in the natural partial order (see Proposition 5.9.1 in [How95]):

$$\text{for } s \in S \text{ and } e \in E\colon e \leq s \Rightarrow s \in E.$$

This condition is equivalent to `E` being majorantly closed in `S`. See `IdempotentGeneratedSubsemigroup` (4.5.5) and `IsMajorantlyClosed` (4.7.6). 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"
```

4.7.4 IsFactorisableSemigroup

▷ IsFactorisableSemigroup(*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 (4.5.2) and Idempotents (4.5.3). 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> IsFactorisableSemigroup(S);
false
gap> IsFactorisableSemigroup(SymmetricInverseSemigroup(5));
true
gap> IsFactorisableSemigroup(DualSymmetricInverseMonoid(5));
false
gap> IsFactorisableSemigroup(FactorisableDualSymmetricInverseSemigroup(5));
true
```

4.7.5 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 bipartition 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
```

4.7.6 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` (4.7.9).

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 bipartition 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
```

4.7.7 IsMonogenicInverseSemigroup

▷ `IsMonogenicInverseSemigroup(S)` (property)

Returns: `true` or `false`.

`IsMonogenicInverseSemigroup` returns `true` if the semigroup S is an inverse monogenic 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` (4.6.10) and `IndexPeriodOfTransformation` (**Reference: IndexPeriodOfTransformation**).

Example
```
gap> f := PartialPerm( [ 1, 2, 3, 6, 8, 10 ], [ 2, 6, 7, 9, 1, 5 ] );;
gap> S := InverseSemigroup(f, f^2, f^3);;
gap> IsMonogenicSemigroup(S);
false
gap> IsMonogenicInverseSemigroup(S);
true
gap> x := Random(DualSymmetricInverseMonoid(100));;
gap> S := InverseSemigroup(x, x^2, x^20);;
gap> IsMonogenicInverseSemigroup(S);
true
```

4.7.8 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` (4.7.5). 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 [ 1 ]>> ]
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> 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 bipartition monoid of degree 3 with 3 generators>
gap> JoinIrreducibleDClasses(D);
[ <Green's D-class: <block bijection: [ 1, 2, -1, -2 ], [ 3, -3 ]>> ]
```

4.7.9 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` (4.7.6).

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
```

4.7.10 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> f := Elements(s)[13];
[1,3](2)
gap> Minorants(s,f);
[ <empty partial perm>, [1,3], <identity partial perm on [ 2 ]> ]
```

```
gap> f := PartialPerm([3,2,4,0]);
[1,3,4](2)
gap> S := InverseSemigroup(f);
<inverse partial perm semigroup of rank 4 with 1 generator>
gap> Minorants(S,f);
[ <identity partial perm on [ 2 ]>, [1,3](2), [3,4](2) ]
```

4.7.11 PrimitiveIdempotents

▷ `PrimitiveIdempotents(S)` (attribute)

Returns: A list of idempotent partial permutations.

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 of partial permutations 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> PrimitiveIdempotents(S);
[ <identity partial perm on [ 4 ]>, <identity partial perm on [ 1 ]>,
  <identity partial perm on [ 2 ]>, <identity partial perm on [ 3 ]> ]
gap> S := DualSymmetricInverseMonoid(4);
<inverse bipartition monoid of degree 4 with 3 generators>
gap> 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, -1 ], [ 2, 3, 4, -2, -3, -4 ]>,
  <block bijection: [ 1, 2, -1, -2 ], [ 3, 4, -3, -4 ]>,
  <block bijection: [ 1, 3, 4, -1, -3, -4 ], [ 2, -2 ]>,
  <block bijection: [ 1, 4, -1, -4 ], [ 2, 3, -2, -3 ]>,
  <block bijection: [ 1, 3, -1, -3 ], [ 2, 4, -2, -4 ]> ]
```

4.7.12 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` (4.7.6). 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` (4.7.9). 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) ] ]
```

4.7.13 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(GreensDClasses(S)[1]);
<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(GreensDClasses(T)[1]);
<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> SameMinorantsSubgroup(x);
[ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
```

4.7.14 SmallerDegreePartialPermRepresentation

▷ SmallerDegreePartialPermRepresentation(*S*) (attribute)

Returns: An isomorphism.

SmallerDegreePartialPermRepresentation attempts to find an isomorphism from the inverse semigroup *S* of partial permutations to another inverse semigroup of partial permutations with smaller degree. If 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]));
<commutative inverse partial perm semigroup 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>, <commutative inverse partial perm semigroup of rank 2
with 1 generator>, function( x ) ... end, function( x ) ... end )
gap> R := Range(T);
<commutative inverse partial perm semigroup 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 rank 6721 with 3 generators>
```

4.7.15 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 (4.7.14).

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 )
```

4.7.16 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 [ 1, 2 ]>, (1,2),
        <identity partial perm on [ 1 ]>, <empty partial perm> ] ]
```

4.8 Visualising the structure of a semigroup

In this section, we describe some functions for creating pictures of various structures related to a semigroup of transformations, partial permutations, or bipartitions; or a subsemigroup of a Rees 0-matrix semigroup.

Several of 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` (4.8.1).

4.8.1 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: `TikzBlocks` (5.8.2), `TikzBipartition` (5.8.1), `DotSemilatticeOfIdempotents` (4.8.3), and `DotDClasses` (4.8.2).

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(DotDClasses(FullTransformationMonoid(4)));
```

4.8.2 DotDClasses (for a semigroup)

▷ `DotDClasses(S)` (attribute)
▷ `DotDClasses(S, record)` (operation)

Returns: A string.

This function 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 `DotDClasses` 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 $\mathscr{D}$-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 red 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 *record* is present, it can be used to specify some options for output.

number
if *record*`.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 *record*`.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 `DotDClasses`. The default value for this option is `false`.

Example
```
gap> S:=FullTransformationSemigroup(3);
<full transformation semigroup of degree 3>
gap> DotDClasses(S);
```

```
"digraph  DClasses {\nnode [shape=plaintext]\nedge [color=red,arrowhe\
ad=none]\n1 [shape=box style=dotted label=<\n<TABLE BORDER=\"0\" CELL\
BORDER=\"1\" CELLPADDING=\"10\" CELLSPACING=\"0\" PORT=\"1\">\n<TR BO\
RDER=\"0\"><TD COLSPAN=\"1\" BORDER=\"0\" >1</TD></TR><TR><TD BGCOLOR\
=\"grey\">*</TD></TR>\n</TABLE>>];\n2 [shape=box style=dotted 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=\"grey\">*</TD><TD BGCOLOR=\"grey\">*</TD>\
<TD></TD></TR>\n<TR><TD BGCOLOR=\"grey\">*</TD><TD></TD><TD BGCOLOR=\
\"grey\">*</TD></TR>\n<TR><TD></TD><TD BGCOLOR=\"grey\">*</TD><TD BGC\
OLOR=\"grey\">*</TD></TR>\n</TABLE>>];\n3 [shape=box style=dotted lab\
el=<\n<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLPADDING=\"10\" CELLSPA\
CING=\"0\" PORT=\"3\">\n<TR BORDER=\"0\"><TD COLSPAN=\"1\" BORDER=\"0\
\" >3</TD></TR><TR><TD BGCOLOR=\"grey\">*</TD></TR>\n<TR><TD BGCOLOR=\
\"grey\">*</TD></TR>\n<TR><TD BGCOLOR=\"grey\">*</TD></TR>\n</TABLE>>\
];\n1 -> 2\n2 -> 3\n }"
gap> FileString(DotDClasses(S), "t3.dot");
fail
gap> FileString("t3.dot", DotDClasses(S));
966
```

4.8.3 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 bipartition monoid of degree 4 with 3 generators>
gap> DotSemilatticeOfIdempotents(S);
"graph graphname {\n  node [shape=point]\nranksep=2;\nsubgraph cluste\
r_1{\n15 \n}\nsubgraph cluster_2{\n5 11 14 8 12 13 \n}\nsubgraph clus\
ter_3{\n2 3 10 4 6 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\n\
12 -- 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 }"
```

Chapter 5

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.

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

The *partition monoid* of degree n is the set of all partitions of $\mathbf{n}\cup$-$\mathbf{n}$ 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 $\mathbf{n}\cup$-$\mathbf{n}$ whose union is $\mathbf{n}\cup$-$\mathbf{n}$; these subsets are called the *blocks* of x. A block containing elements of both $\mathbf{n}$ and -$\mathbf{n}$ is called a *transverse block*. If $i,j \in \mathbf{n}\cup$-$\mathbf{n}$ belong to the same block of a bipartition x, then we write $(i,j) \in x$.

Let x and y be bipartitions of equal degree. Then xy is the bipartition where $i,j \in \mathbf{n}\cup$-$\mathbf{n}$ belong to the same block of xy if there exist $k(1),k(2),\ldots,k(r) \in \mathbf{n}$; and one of the following holds:

- $r= 0$ and either $(i,j) \in x$ or $(-i,-j) \in y$;
- $r = 2s-1$ for some $s \geq 1$ and

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

- $r = 2s$ for some $s \geq 1$ and either:

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

  or

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

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

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 $i \in \mathbf{n}$. 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*.

5.1 The family and categories of bipartitions

5.1.1 IsBipartition

▷ `IsBipartition(obj)` (Category)

Returns: `true` or `false`.

Every bipartition in GAP belongs to the category `IsBipartition`. Basic operations for bipartitions are `RightBlocks` (5.5.4), `LeftBlocks` (5.5.5), `ExtRepOfBipartition` (5.5.3), `LeftProjection` (5.2.4), `RightProjection` (5.2.5), `StarOp` (5.2.6), `DegreeOfBipartition` (5.5.1), `RankOfBipartition` (5.5.2), multiplication of two bipartitions of equal degree is via `*`.

5.1.2 IsBipartitionCollection

▷ `IsBipartitionCollection(obj)` (Category)

Returns: `true` or `false`.

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

5.1.3 BipartitionFamily

▷ `BipartitionFamily` (family)

The family of all bipartitions is `BipartitionFamily`.

5.2 Creating bipartitions

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

5.2.1 Bipartition

▷ `Bipartition(blocks)` (function)

Returns: A bipartition.

`Bipartition` returns the bipartition `f` 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> f:=Bipartition( [ [ 1, -1 ],[ 2, 3, -3 ], [ -2 ] ] );
<bipartition: [ 1, -1 ], [ 2, 3, -3 ], [ -2 ]>
```

5.2.2 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

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

has internal representation

```
[ 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.

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

5.2.3 IdentityBipartition

▷ `IdentityBipartition(n)` (operation)

Returns: The identity bipartition.

Returns the identity bipartition with degree `n`.

```
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 ]>
```

5.2.4 LeftOne (for a bipartition)

▷ `LeftOne(f)` (attribute)

▷ `LeftProjection(f)` (attribute)

Returns: A bipartition.

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

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

```
gap> f:=Bipartition( [ [ 1, 4, -1, -2, -6 ], [ 2, 3, 5, -4 ],
> [ 6, -3 ], [ -5 ] ] );;
gap> LeftOne(f);
<block bijection: [ 1, 4, -1, -4 ], [ 2, 3, 5, -2, -3, -5 ],
```

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

5.2.5 RightOne (for a bipartition)

▷ `RightOne(f)` (attribute)
▷ `RightProjection(f)` (attribute)

Returns: A bipartition.

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

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

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

5.2.6 StarOp

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

Returns: A bipartition.

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

Example
```
gap> f:=Bipartition( [ [ 1, -4 ], [ 2, 3, 4 ], [ 5 ], [ -1 ],
> [ -2, -3 ], [ -5 ] ] );
<bipartition: [ 1, -4 ], [ 2, 3, 4 ], [ 5 ], [ -1 ], [ -2, -3 ],
 [ -5 ]>
gap> g:=Star(f);
<bipartition: [ 1 ], [ 2, 3 ], [ 4, -1 ], [ 5 ], [ -2, -3, -4 ],
 [ -5 ]>
gap> f*g*f=f;
```

```
true
gap> LeftBlocks(f)=RightBlocks(g);
true
gap> RightBlocks(f)=LeftBlocks(g);
true
```

5.2.7 RandomBipartition

▷ `RandomBipartition(n)` (operation)

Returns: A bipartition.

If *n* is a positive integer, then `RandomBipartition` returns a random bipartition of degree *n*.

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

5.3 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` (5.3.5), `AsPartialPerm` (5.3.4), `AsTransformation` (5.3.3) can be used to convert bipartitions into permutations, partial permutations, or transformations where appropriate.

5.3.1 AsBipartition

▷ `AsBipartition(f[, n])` (operation)

Returns: A bipartition.

`AsBipartition` returns the bipartition, permutation, transformation, or partial permutation *f*, as a bipartition of degree *n*. There are several possible arguments for `AsBipartition`:

permutations

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

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

transformations

If *f* is a transformation and *n* is a positive integer such that *f* 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 *f*.

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

partial permutations

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

bipartitions

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

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

Example

```
gap> f:=Transformation( [ 3, 5, 3, 4, 1, 2 ] );;
gap> AsBipartition(f, 5);
<bipartition: [ 1, 3, -3 ], [ 2, -5 ], [ 4, -4 ], [ 5, -1 ], [ -2 ]>
gap> AsBipartition(f);
<bipartition: [ 1, 3, -3 ], [ 2, -5 ], [ 4, -4 ], [ 5, -1 ],
 [ 6, -2 ], [ -6 ]>
gap> AsBipartition(f, 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> f:=PartialPerm( [ 1, 2, 3, 4, 5, 6 ], [ 6, 7, 1, 4, 3, 2 ] );;
gap> AsBipartition(f, 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(f);
<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> f:=Bipartition( [ [ 1, 2, -2 ], [ 3 ], [ 4, 5, 6, -1 ],
> [ -3, -4, -5, -6 ] ] );;
gap> AsBipartition(f, 0);
<empty bipartition>
gap> AsBipartition(f, 2);
<bipartition: [ 1, 2, -2 ], [ -1 ]>
gap> AsBipartition(f, 8);
<bipartition: [ 1, 2, -2 ], [ 3 ], [ 4, 5, 6, -1 ], [ 7 ], [ 8 ],
 [ -3, -4, -5, -6 ], [ -7 ], [ -8 ]>
```

5.3.2 AsBlockBijection

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

Returns: A block bijection or `fail`.

When the argument *f* is a partial perm and *n* is a positive integer which is greater than the maximum of the degree and codegree of *f*, this function returns a block bijection corresponding to *f*. This block bijection has the same non-singleton classes as `g:=AsBipartition(f, 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 *f* plus 1 is used by default. If the second argument *n* is not greater than this maximum, then `fail` is returned.

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

Example
```
gap> f:=PartialPerm( [ 1, 2, 3, 6, 7, 10 ], [ 9, 5, 6, 1, 7, 8 ] ) ;
[2,5][3,6,1,9][10,8](7)
gap> AsBipartition(f, 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(f, 10);
fail
gap> AsBlockBijection(f, 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 ]>
```

5.3.3 AsTransformation (for a bipartition)

▷ `AsTransformation(f)` (operation)

Returns: A transformation or `fail`.

When the argument *f* is a bipartition, that mathematically defines a transformation, this function returns that transformation. A bipartition *f* 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 *f*.

See `IsTransBipartition` (5.5.9).

Example
```
gap> f:=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(f);
Transformation( [ 3, 2, 7, 12, 7, 6, 6, 5, 11, 7, 10, 10 ] )
gap> IsTransBipartition(f);
true
gap> f:=Bipartition([[ 1, 5 ], [ 2, 4, 8, 10 ], [ 3, 6, 7, -1, -2 ],
> [ 9, -4, -6, -9 ], [ -3, -5 ], [ -7, -8 ], [ -10 ]]);;
gap> AsTransformation(f);
fail
```

5.3.4 AsPartialPerm (for a bipartition)

▷ `AsPartialPerm(f)` (operation)

Returns: A partial perm or `fail`.

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

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

See `IsPartialPermBipartition` (5.5.12).

Example
```
gap> f:=Bipartition( [ [ 1, -4 ], [ 2, -2 ], [ 3, -10 ], [ 4, -5 ],
> [ 5, -9 ], [ 6 ], [ 7 ], [ 8, -6 ], [ 9, -3 ], [ 10, -8 ],
> [ -1 ], [ -7 ] ] );;
gap> IsPartialPermBipartition(f);
true
gap> AsPartialPerm(f);
[1,4,5,9,3,10,8,6](2)
gap> f:=Bipartition([[ 1, -2, -4 ], [ 2, 3, 4, -3 ], [ -1 ]]);;
gap> IsPartialPermBipartition(f);
false
gap> AsPartialPerm(f);
fail
```

5.3.5 AsPermutation (for a bipartition)

▷ `AsPermutation(f)` (operation)

Returns: A permutation or `fail`.

When the argument `f` is a bipartition that mathematically defines a permutation, this function returns that permutation.

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

See `IsPermBipartition` (5.5.11).

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

5.4 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.

5.4.1 PartialPermLeqBipartition

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

Returns: `true` or `false`.

If *x* and *y* are partial perm bipartitions, i.e. they satisfy `IsPartialPermBipartition` (5.5.12), then this function returns `AsPartialPerm(x)<AsPartialPerm(y)`.

5.4.2 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.

5.4.3 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
```

5.4.4 PermLeftQuoBipartition

▷ `PermLeftQuoBipartition(f, g)` (operation)

Returns: A permutation.

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

`PermLeftQuoBipartition` verifies that `f` and `g` have equal left and right blocks, and returns an error if they do not. The value returned by `PermLeftQuoBipartition(f,g)` is the same as that returned by `PermRightBlocks(RightBlocks(f), Star(f)*g)`. See also `PermRightBlocks` (5.7.3) and `OnRightBlocksBipartitionByPerm` (5.4.5).

Example

```
gap> f:=Bipartition( [ [ 1, 4, 6, 7, 8, 10 ], [ 2, 5, -1, -2, -8 ],
> [ 3, -3, -6, -7, -9 ], [ 9, -4, -5 ], [ -10 ] ] );;
gap> g:=Bipartition( [ [ 1, 4, 6, 7, 8, 10 ], [ 2, 5, -3, -6, -7, -9 ],
> [ 3, -4, -5 ], [ 9, -1, -2, -8 ], [ -10 ] ] );;
gap> PermLeftQuoBipartition(f, g);
(1,2,3)
gap> Star(f)*g;
<bipartition: [ 1, 2, 8, -3, -6, -7, -9 ], [ 3, 6, 7, 9, -4, -5 ],
 [ 4, 5, -1, -2, -8 ], [ 10 ], [ -10 ]>
gap> PermRightBlocks(RightBlocks(f), last);
(1,2,3)
```

5.4.5 OnRightBlocksBipartitionByPerm

▷ `OnRightBlocksBipartitionByPerm(f, p)` (function)

Returns: A bipartition.

If `f` is a bipartition and `p` is a permutation of the indices of the right blocks of `f`, then `OnRightBlocksBipartitionByPerm` returns the bipartition obtained from `f` by rearranging the right blocks of `f` according to `p`.

Example

```
gap> f:=Bipartition( [ [ 1, 4, 6, 7, 8, 10 ], [ 2, 5, -1, -2, -8 ],
> [ 3, -3, -6, -7, -9 ], [ 9, -4, -5 ], [ -10 ] ] );;
gap> OnRightBlocksBipartitionByPerm(f, (1,2,3));
<bipartition: [ 1, 4, 6, 7, 8, 10 ], [ 2, 5, -3, -6, -7, -9 ],
 [ 3, -4, -5 ], [ 9, -1, -2, -8 ], [ -10 ]>
```

5.5 Attributes for bipartitons

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

5.5.1 DegreeOfBipartition

▷ `DegreeOfBipartition(f)` (attribute)

▷ `DegreeOfBipartitionCollection(f)` (attribute)

Returns: A positive integer.

The degree of a bipartition is, roughly speaking, the number of points where it is defined. More precisely, if `f` is a bipartition defined on `2*n` points, then the degree of `f` 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> f:=Bipartition( [ [ 1, 7, -3, -8 ], [ 2, 6 ], [ 3 ], [ 4, -7, -9 ],
> [ 5, 9, -2 ], [ 8, -1, -4, -6 ], [ -5 ] ] );;
gap> DegreeOfBipartition(f);
9
gap> s:=BrauerMonoid(5);
<regular bipartition monoid of degree 5 with 3 generators>
gap> IsBipartitionCollection(s);
true
gap> DegreeOfBipartitionCollection(s);
5
```

5.5.2 RankOfBipartition

▷ RankOfBipartition(*f*) (attribute)
▷ NrTransverseBlocks(*f*) (attribute)

Returns: The rank of a bipartition.

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

NrTransverseBlocks is just a synonym for RankOfBipartition.

Example
```
gap> f:=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(f);
4
```

5.5.3 ExtRepOfBipartition

▷ ExtRepOfBipartition(*f*) (attribute)

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

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

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

5.5.4 RightBlocks

▷ RightBlocks(*f*) (attribute)

Returns: The right blocks of a bipartition.

RightBlocks returns the right blocks of the bipartition *f*.

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

The right blocks of bipartition are GAP objects in their own right, and are not simply a list of blocks of *f*; see 5.6 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> f:=Bipartition( [ [ 1, 4, 7, 8, -4 ], [ 2, 3, 5, -2, -7 ],
> [ 6, -1 ], [ -3 ], [ -5, -6, -8 ] ] );;
gap> RightBlocks(f);
<blocks: [ 1 ], [ 2, 7 ], [ -3 ], [ 4 ], [ -5, -6, -8 ]>
gap> LeftBlocks(f);
<blocks: [ 1, 4, 7, 8 ], [ 2, 3, 5 ], [ 6 ]>
```

5.5.5 LeftBlocks

▷ LeftBlocks(*f*) (attribute)

Returns: The left blocks of a bipartition.

LeftBlocks returns the left blocks of the bipartition *f*.

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

The left blocks of bipartition are GAP objects in their own right, and are not simply a list of blocks of *f*; see 5.6 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> f:=Bipartition( [ [ 1, 4, 7, 8, -4 ], [ 2, 3, 5, -2, -7 ],
> [ 6, -1 ], [ -3 ], [ -5, -6, -8 ] ] );;
gap> RightBlocks(f);
<blocks: [ 1 ], [ 2, 7 ], [ -3 ], [ 4 ], [ -5, -6, -8 ]>
gap> LeftBlocks(f);
<blocks: [ 1, 4, 7, 8 ], [ 2, 3, 5 ], [ 6 ]>
```

5.5.6 NrLeftBlocks

▷ NrLeftBlocks(*f*) (attribute)

Returns: A non-negative integer.

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

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

5.5.7 NrRightBlocks

▷ NrRightBlocks(*f*) (attribute)

Returns: A non-negative integer.

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

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

5.5.8 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> f:=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(f);
4
```

5.5.9 IsTransBipartition

▷ IsTransBipartition(*f*) (property)

Returns: true or false.

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

A bipartition *f* 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> f:=Bipartition( [ [ 1, 4, -2 ], [ 2, 5, -6 ], [ 3, -7 ], [ 6, 7, -9 ],
> [ 8, 9, -1 ], [ 10, -5 ], [ -3 ], [ -4 ], [ -8 ], [ -10 ] ] );;
gap> IsTransBipartition(f);
true
gap> f:=Bipartition( [ [ 1, 4, -3, -6 ], [ 2, 5, -4, -5 ], [ 3, 6, -1 ],
> [ -2 ] ] );;
gap> IsTransBipartition(f);
false
```

```
gap> Number(PartitionMonoid(3), IsTransBipartition);
27
```

5.5.10 IsDualTransBipartition

▷ IsDualTransBipartition(*f*) (property)

Returns: true or false.

If the star of the bipartition *f* 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> f:=Bipartition( [ [ 1, -8, -9 ], [ 2, -1, -4 ], [ 3 ], [ 4 ],
> [ 5, -10 ], [ 6, -2, -5 ], [ 7, -3 ], [ 8 ], [ 9, -6, -7 ], [ 10 ] ] );;
gap> IsDualTransBipartition(f);
true
gap> f:=Bipartition( [ [ 1, 4, -3, -6 ], [ 2, 5, -4, -5 ], [ 3, 6, -1 ],
> [ -2 ] ] );;
gap> IsTransBipartition(f);
false
gap> Number(PartitionMonoid(3), IsDualTransBipartition);
27
```

5.5.11 IsPermBipartition

▷ IsPermBipartition(*f*) (property)

Returns: true or false.

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

5.5.12 IsPartialPermBipartition

▷ IsPartialPermBipartition(*f*) (property)

Returns: true or false.

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

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

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

5.5.13 IsBlockBijection

▷ `IsBlockBijection(f)` (property)

Returns: `true` or `false`.

If the bipartition *f* 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> f:=Bipartition( [ [ 1, 4, 5, -2 ], [ 2, 3, -1 ],
> [ 6, -5, -6 ], [ -3, -4 ] ] );;
gap> IsBlockBijection(f);
false
gap> f:=Bipartition( [ [ 1, 2, -3 ], [ 3, -1, -2 ], [ 4, -4 ],
> [ 5, -5 ] ] );;
gap> IsBlockBijection(f);
true
```

5.5.14 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
```

5.6 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 on the partition monoid; see `LeftBlocks` (5.5.5) and `RightBlocks` (5.5.4). 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.

5.6.1 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.

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

5.6.2 ExtRepOfBlocks

▷ `ExtRepOfBlocks(blocks)` (attribute)

Returns: A list of integers.

If `n` is the degree of a bipartition with left or right blocks *blocks*, then `ExtRepOfBlocks` 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> ExtRepOfBlocks(blocks);
[ [ 1, 6 ], [ 2, 3, 7 ], [ 4, 5 ], [ -8 ] ]
```

5.6.3 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
```

5.6.4 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
```

5.7 Actions on blocks

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

5.7.1 OnRightBlocks

▷ `OnRightBlocks(blocks, f)` (function)

Returns: The blocks of a bipartition.

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

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

5.7.2 OnLeftBlocks

▷ `OnLeftBlocks(blocks, f)` (function)

Returns: The blocks of a bipartition.

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

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

5.7.3 PermRightBlocks

▷ `PermRightBlocks(blocks, f)` (operation)
▷ `PermLeftBlocks(blocks, f)` (operation)

Returns: A permutation.

If *f* is a bipartition that stabilises *blocks*, i.e. `OnRightBlocks(blocks, f)=blocks`, then `PermRightBlocks` returns the permutation of the indices of the transverse blocks of *blocks* under the action of *f*.

`PermLeftBlocks` is the analogue of `PermRightBlocks` with respect to `OnLeftBlocks` (5.7.2).

Example
```
gap> f:=Bipartition( [ [ 1, 10 ], [ 2, -7, -9 ], [ 3, 4, 6, 8 ], [ 5, -5 ],
> [ 7, 9, -2 ], [ -1, -10 ], [ -3, -4, -6, -8 ] ] );;
gap> blocks:=BlocksNC([[ -1, -10 ], [ 2 ], [ -3, -4, -6, -8 ], [ 5 ],
> [ 7, 9 ]]);;
gap> OnRightBlocks(blocks, f)=blocks;
true
gap> PermRightBlocks(blocks, f);
(2,5)
```

5.7.4 InverseRightBlocks

▷ `InverseRightBlocks(blocks, f)` (function)

Returns: A bipartition.

If `OnRightBlocks(blocks, f)` has rank equal to the rank of *blocks*, then `InverseRightBlocks` returns a bipartition g such that `OnRightBlocks(blocks, f*g)=blocks` and where `PermRightBlocks(blocks, f*g)` is the identity permutation.

See `PermRightBlocks` (5.7.3) and `OnRightBlocks` (5.7.1).

Example
```
gap> f:=Bipartition( [ [ 1, 4, 7, 8, -4 ], [ 2, 3, 5, -2, -7 ],
> [ 6, -1 ], [ -3 ], [ -5, -6, -8 ] ] );;
gap> blocks:=BlocksNC([[ -1, -4, -5, -8 ], [ -2, -3, -7 ], [ 6 ]]);;
gap> RankOfBlocks(blocks);
1
gap> RankOfBlocks(OnRightBlocks(blocks, f));
1
gap> g:=InverseRightBlocks(blocks, f);
<bipartition: [ 1, -6 ], [ 2, 3, 4, 5, 6, 7, 8 ], [ -1, -4, -5, -8 ],
 [ -2, -3, -7 ]>
gap> blocks;
<blocks: [ -1, -4, -5, -8 ], [ -2, -3, -7 ], [ 6 ]>
gap> OnRightBlocks(blocks, f*g);
<blocks: [ -1, -4, -5, -8 ], [ -2, -3, -7 ], [ 6 ]>
gap> PermRightBlocks(blocks, f*g);
()
```

5.7.5 InverseLeftBlocks

▷ `InverseLeftBlocks(blocks, f)` (function)

Returns: A bipartition.

If `OnLeftBlocks(blocks, f)` has rank equal to the rank of `blocks`, then `InverseLeftBlocks` returns a bipartition `g` such that `OnLeftBlocks(blocks, g*f)=blocks` and where `PermLeftBlocks(blocks, g*f)` is the identity permutation.

See `PermLeftBlocks` (5.7.3) and `OnLeftBlocks` (5.7.2).

Example

```
gap> f:=Bipartition( [ [ 1, 4, 7, 8, -4 ], [ 2, 3, 5, -2, -7 ],
> [ 6, -1 ], [ -3 ], [ -5, -6, -8 ] ] );;
gap> blocks:=BlocksNC([[ -1, -2, -6 ], [ 3, 4, 5 ], [ -7, -8 ]]);;
gap> RankOfBlocks(OnLeftBlocks(blocks, f));
1
gap> g:=InverseLeftBlocks(blocks, f);
<bipartition: [ 1, 2, 6 ], [ 3, 4, 5, -1, -2, -3, -4, -5, -6, -7, -8 ]
 , [ 7, 8 ]>
gap> OnLeftBlocks(blocks, g*f);
<blocks: [ -1, -2, -6 ], [ 3, 4, 5 ], [ -7, -8 ]>
gap> PermLeftBlocks(blocks, g*f);
()
```

5.8 Visualising blocks and bipartitions

There are some functions in Semigroups for creating LaTeX pictures of bipartitions and blocks. Descriptions of these methods can be found in this section.

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` (4.8.1).

5.8.1 TikzBipartition

▷ `TikzBipartition(f[, opts])` (function)

Returns: A string.

This function produces a graphical representation of the bipartition *f* using the `tikz` package for LaTeX. More precisely, this function outputs a string containing a minimal LaTeX document which can be compiled using LaTeX to produce a picture of *f*.

If the optional second argument *opts* 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 *f* is less than 20.

Example

```
gap> f:=Bipartition( [ [ 1, 5 ], [ 2, 4, -3, -5 ], [ 3, -1, -2 ],
> [ -4 ] ] );;
gap> TikzBipartition(f);
"%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}"
```

5.8.2 TikzBlocks

▷ TikzBlocks(*blocks*) (function)

Returns: A string.

This function produces a graphical representation of the blocks *blocks* of a bipartition using the tikz package for LaTeX. More precisely, this function outputs a string containing a minimal LaTeX document which can be compiled using LaTeX to produce a picture of *blocks*.

Example

```
gap> f:=Bipartition( [ [ 1, 4, -2, -3 ], [ 2, 3, 5, -5 ], [ -1, -4 ] ] );;
gap> TikzBlocks(RightBlocks(f));
"%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}"
```

5.9 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**).

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 (5.5.13) and IsPartialPermBipartition (5.5.12).

5.9.1 IsBipartitionSemigroup

▷ IsBipartitionSemigroup(*S*) (property)

▷ IsBipartitionMonoid(*S*) (property)

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 (5.1.2).

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` (5.1.2).

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> f:=Bipartition( [ [ 1, 4, -2 ], [ 2, 5, -6 ], [ 3, -7 ],
> [ 6, 7, -9 ], [ 8, 9, -1 ], [ 10, -5 ], [ -3 ], [ -4 ],
> [ -8 ], [ -10 ] ] );;
gap> S:=Semigroup(f, One(f));
<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**).

5.9.2 IsBlockBijectionSemigroup

▷ `IsBlockBijectionSemigroup(S)` (property)

▷ `IsBlockBijectionMonoid(S)` (property)

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` (5.5.13).

5.9.3 IsPartialPermBipartitionSemigroup

▷ `IsPartialPermBipartitionSemigroup(S)` (property)

▷ `IsPartialPermBipartitionMonoid(S)` (property)

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` (5.5.12).

5.9.4 IsPermBipartitionGroup

▷ `IsPermBipartitionGroup(S)` (property)

Returns: `true` or `false`.

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

See `IsPermBipartition` (5.5.11).

5.9.5 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 6

Free inverse semigroups and free bands

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.

6.1 Free inverse semigroups

F is a *free inverse semigroup* on a non-empty set X if F is an inverse semigroup with 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 the 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 be 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` (6.3.2) 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` (6.3.1).

6.1.1 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
```

6.1.2 IsFreeInverseSemigroupCategory

▷ `IsFreeInverseSemigroupCategory(obj)` (Category)

Every free inverse semigroup in GAP created by `FreeInverseSemigroup` (6.1.1) 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` (6.1.4).

6.1.3 IsFreeInverseSemigroup

▷ `IsFreeInverseSemigroup(S)` (property)

Returns: true or false

Attempts to determine whether the given semigroup *S* is a free inverse semigroup.

6.1.4 IsFreeInverseSemigroupElement

▷ `IsFreeInverseSemigroupElement` (Category)

Every element of a free inverse semigroup belongs to the category `IsFreeInverseSemigroupElement`.

6.2 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
```

6.3 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.

6.3.1 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"
```

6.3.2 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"
```

6.4 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.

6.4.1 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> a := gens[1];; b := gens[2];;
gap> a * b;
ab
```

6.4.2 IsFreeBandCategory

▷ IsFreeBandCategory (Category)

IsFreeBandCategory is the category of semigroups created using FreeBand (6.4.1).

Example
```
gap> IsFreeBandCategory(FreeBand(3));
true
gap> IsFreeBand(SymmetricGroup(6));
false
```

6.4.3 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
```

6.4.4 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
```

6.4.5 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 := Generators(S)[1];
x1
gap> y := Generators(S)[2];
x2
gap> new := Semigroup([x*y, x]);
<semigroup with 2 generators>
gap> IsFreeBand(new);
false
gap> IsFreeBandSubsemigroup(new);
true
```

6.5 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.

6.5.1 GreensDClassOfElement (for a free band and a free band element)

▷ GreensDClassOfElement(*s*, *x*) (operation)

Returns: A Green's 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 := Generators(S)[1] * Generators(S)[2];
b1b2
gap> D := GreensDClassOfElement(S, x);
<Green's D-class: b1b2>
gap> IsGreensDClass(D);
true
```

Chapter 7

Matrix semigroups

This chapter describes the functions in Semigroups for dealing with matrix semigroups. This part of the manual and the functions described herein were written by Markus Pfeiffer.

A *matrix semigroup* for the purposes of this document is a subsemigroup of the full monoid of $n \times n$ matrices over a *finite field* $\mathbb{F}$.

More general matrix semigroups are planned, but not implemented yet.

GAP provides a way to define matrices which are in the filter `IsMatrix` (**Reference: IsMatrix**). For technical reasons, the matrix semigroup functions in Semigroups rely on a custom wrapper for matrices `IsMatrixOverFiniteField` (7.2.1).

Example
```
gap> x := Z(4) * [[1,0], [0,2]];
[ [ Z(2^2), 0*Z(2) ], [ 0*Z(2), 0*Z(2) ] ]
gap> IsMatrix(x);
true
gap> IsMatrixOverFiniteField(x);
false
gap> y := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 2, x);
<matrix over GF(2^2) of degree 2>
gap> IsMatrix(y);
false
gap> IsMatrixOverFiniteField(y);
true
```

In the following we will refer to matrices in `IsMatrix` (**Reference: IsMatrix**) by *GAP library matrices* and to matrices in `IsMatrixOverFiniteField` (7.2.1) by *matrices over finite fields*. We take precautions to hide this fact from the user of Semigroups and also provide conversion functions between the two representations.

7.1 Creating matrix semigroups

Random matrix semigroups can be created by using the functions `RandomMatrixSemigroup` (2.1.6) or `RandomMatrixMonoid` (2.1.6). While this is convenient for testing and playing around, creating semigroups from matrices can be a bit more work. We provide a couple of convenience functions to streamline the process.

7.1.1 IsMatrixSemigroup

▷ `IsMatrixSemigroup(S)` (property)

▷ `IsMatrixMonoid(S)` (property)

Returns: `true` or `false`.

A *matrix semigroup* is simply a semigroup consisting of matrices over a finite field. An object in GAP is a matrix semigroup if it satisfies `IsSemigroup` (**Reference: IsSemigroup**) and `IsMatrixOverFiniteFieldCollection` (7.2.2).

A *matrix monoid* is simply a monoid consisting of matrices over a finite field. An object in GAP is a matrix monoid if it satisfies `IsMonoid` (**Reference: IsMonoid**) and `IsMatrixOverFiniteFieldCollection` (7.2.2).

Note that it is possible for a matrix semigroup to have a multiplicative neutral element (i.e. an identity element) but not to satisfy `IsMatrixMonoid`.

7.1.2 MatrixSemigroup

▷ `MatrixSemigroup(list[, F])` (function)

Returns: A matrix semigroup.

This is a helper function to create matrix semigroups from GAP matrices. The argument *list* is a homogeneous list of GAP matrices over a finite field, and the optional argument *F* is a finite field.

The specification of the field *F* can be necessary to prevent GAP from trying to find a smaller common field for the entries in *list*.

Example

```
gap> S := MatrixSemigroup([Z(3) * [[1,0,0], [1,1,0], [0,1,0]],
>                          Z(3) * [[0,0,0], [0,0,1], [0,1,0]]], GF(9));
<semigroup of 3x3 matrices over GF(3^2) with 2 generators>
gap> S := MatrixSemigroup([Z(3) * [[1,0,0], [1,1,0], [0,1,0]],
>                          Z(3) * [[0,0,0], [0,0,1], [0,1,0]]]);
<semigroup of 3x3 matrices over GF(3) with 2 generators>
gap> S := MatrixSemigroup([Z(4) * [[1,0,0], [1,1,0], [0,1,0]],
>                          Z(4) * [[0,0,0], [0,0,1], [0,1,0]]]);
<semigroup of 3x3 matrices over GF(2^2) with 2 generators>
```

In addition to the above, `IsomorphismMatrixSemigroup` (2.4.5) and `AsMatrixSemigroup` (2.4.1) can be used to create a matrix semigroup isomorphic to an already known semigroup.

7.2 Matrices in the Semigroups package

The matrix functions in the Semigroups package use a wrapper object for matrices. In the following these objects are documented.

7.2.1 IsMatrixOverFiniteField

▷ `IsMatrixOverFiniteField(obj)` (Category)

Returns: `true` or `false`.

This category contains Semigroups matrix object wrapper. The introduction of this filter was necessary to get around GAP limitations with regards to matrices and associative objects.

The behaviour of this object type might be changed or removed completely from the package in the future.

Example
```
gap> x := Z(4) * [[1,0], [0,2]];
[ [ Z(2^2), 0*Z(2) ], [ 0*Z(2), 0*Z(2) ] ]
gap> IsMatrixOverFiniteField(x);
false
gap> y := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 2, x);
<matrix over GF(2^2) of degree 2>
gap> IsMatrixOverFiniteField(y);
true
```

7.2.2 IsMatrixOverFiniteFieldCollection

▷ `IsMatrixOverFiniteFieldCollection(obj)` (Category)

Returns: `true` or `false`.

Every collection of matrices in the category `IsMatrixOverFiniteField` (7.2.1) belongs to the category `IsMatrixOverFiniteFieldCollection`. For example, matrix semigroup belong to `IsMatrixOverFiniteFieldCollection`.

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

▷ `NewMatrixOverFiniteField(filt, F, n, 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), 2,
> Z(4)*[[1,0],[0,1]]);
<matrix over GF(2^2) of degree 2>
gap> y := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 0, []);
<matrix over GF(2^2) of degree 0>
```

7.2.4 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 over GF(2^2) of degree 2>
gap> y := NewZeroMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep,
>                                      GF(4), 2);
<matrix over GF(2^2) of degree 2>
```

7.2.5 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 := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> Z(4)^0*[[1,1,0], [0,1,1], [1,1,1]] );
<matrix over GF(2^2) of degree 3>
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 ] ]
```

7.2.6 RowRank (for a matrix over finite field)

▷ RowRank(*m*) (attribute)

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

Example

```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(5), 3,
> Z(5)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(5) of degree 3>
gap> RowRank(x);
2
```

7.2.7 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 := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> Z(4)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(2^2) of degree 3>
gap> LeftInverse(x);
<matrix over GF(2^2) of degree 3>
gap> Display(LeftInverse(x) * x);
<matrix over GF(2^2) of degree 3
[ [ 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 ] ]>
```

7.2.8 DegreeOfMatrixOverFiniteField (for a matrix over finite field)

▷ DegreeOfMatrixOverFiniteField(*m*) (attribute)

Returns: Number of rows and columns of the matrix *m*.

Example
```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(5), 3,
> Z(5)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(5) of degree 3>
gap> DegreeOfMatrixOverFiniteField(x);
3
```

7.2.9 BaseDomain (for a matrix over finite field)

▷ BaseDomain(*m*) (attribute)

Returns: The domain in which entries of *m* lie.

Example
```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(5), 3,
> Z(5)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(5) of degree 3>
gap> BaseDomain(x);
GF(5)
```

7.2.10 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 per default.

Example
```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(5), 3,
> Z(5)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(5) of degree 3>
gap> TransposedMatImmutable(x);
<matrix over GF(5) of degree 3>
```

7.2.11 AsMatrix (for a matrix over finite field)

▷ AsMatrix(*m*) (operation)

Returns: A matrix.

Turns a matrix over a finite field into a GAP matrix.

Example
```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(5), 3,
> Z(5)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(5) of degree 3>
gap> AsMatrix(x);
[ [ 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 ] ]
```

7.2.12 ConstructingFilter (for a matrix over finite field)

▷ ConstructingFilter(*m*) (operation)

Returns: A filter

Return the filter that was passed to NewMatrixOverFiniteField (7.2.3) when creating the matrix *m*. This is used to create new objects that lie in the same filter.

```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> Z(4)^0*[[1,1,0], [0,0,0], [1,1,1]] );
<matrix over GF(2^2) of degree 3>
gap> ConstructingFilter(x);
<Representation "IsPlistMatrixOverFiniteFieldRep">
```

7.3 Matrix groups in the Semigroups package

For interfacing the semigroups code with GAPs library code for matrix groups, the Semigroups package implements matrix groups that delegate to the GAP library.

7.3.1 IsMatrixOverFiniteFieldGroup

▷ IsMatrixOverFiniteFieldGroup(*G*) (property)

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 (7.2.2).

```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]]);
<matrix over GF(2^2) of degree 3>
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
```

7.3.2 \^ (for an 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 (7.3.1) and IsMatrixOverFiniteField (7.2.1). 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 := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> Z(4) ^ 0 * [[1, 1, 0], [0, 1, 0], [1, 1, 1]] );;
gap> y := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> 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>
```

7.3.3 IsomorphismMatrixGroup

▷ `IsomorphismMatrixGroup(G)` (attribute)

Returns: An isomorphism.

If *G* belongs to the category `IsMatrixOverFiniteFieldGroup` (7.3.1), then `IsomorphismMatrixGroup` returns an isomorphism from *G* into a group consisting of GAP library matrices.

Example

```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> 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 ] ] ])
```

7.3.4 AsMatrixGroup

▷ `AsMatrixGroup(G)` (attribute)

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

Returns the image of the isomorphism returned by 7.3.3.

Example

```
gap> x := NewMatrixOverFiniteField(IsPlistMatrixOverFiniteFieldRep, GF(4), 3,
> 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 8

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` (8.1.1) 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 will 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.

8.1 Creating congruences

8.1.1 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)>
```

8.2 Congruence classes

8.2.1 CongruenceClassOfElement

▷ `CongruenceClassOfElement(`*cong*, *elm*`)` (operation)

Returns: A congruence class.

This operation is a synonym of `EquivalenceClassOfElement` in the case that the argument *cong* is a congruence of a semigroup.

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);
{(1,(1,3,2),1)}
```

8.2.2 CongruenceClasses

▷ `CongruenceClasses(`*cong*`)` (attribute)

Returns: The classes of congruence.

When *cong* is a congruence of semigroup, this attribute is synonymous with `EquivalenceClasses`.

The return value is a list containing all the equivalence classes of the semigroup congruence *cong*.

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(),(1,3,2)],[(1,2),0]]);;
gap> cong := CongruencesOfSemigroup(S)[3];;
gap> classes := CongruenceClasses(cong);;
gap> Size(classes);
9
```

8.2.3 NrCongruenceClasses

▷ `NrCongruenceClasses(`*cong*`)` (attribute)

Returns: A positive integer.

This attribute describes the number of congruence classes in the semigroup congruence *cong*.

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(),(1,3,2)],[(1,2),0]]);;
```

```
gap> cong := CongruencesOfSemigroup(S)[3];;
gap> NrCongruenceClasses(cong);
9
```

8.2.4 CongruencesOfSemigroup

▷ CongruencesOfSemigroup(*S*) (attribute)

Returns: The congruences of a semigroup.

This attribute gives a list of the congruences of the semigroup *S*.

At present this only works for simple and 0-simple semigroups.

Example

```
gap> s := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(),(1,3,2)],[(1,2),0]]);;
gap> congs := CongruencesOfSemigroup(s);
[ <universal semigroup congruence over
    <Rees 0-matrix semigroup 2x2 over Sym( [ 1 .. 3 ] )>>,
  <semigroup congruence over <Rees 0-matrix semigroup 2x2 over
      Sym( [ 1 .. 3 ] )> with linked triple (1,2,2)>,
  <semigroup congruence over <Rees 0-matrix semigroup 2x2 over
      Sym( [ 1 .. 3 ] )> with linked triple (3,2,2)>,
  <semigroup congruence over <Rees 0-matrix semigroup 2x2 over
      Sym( [ 1 .. 3 ] )> with linked triple (S3,2,2)> ]
```

8.2.5 AsLookupTable

▷ AsLookupTable(*cong*) (attribute)

Returns: A list.

This attribute describes the semigroup congruence *cong* as a list of positive integers with length the size of the semigroup over which *cong* is defined.

Each position in the list corresponds to an element of the semigroup (in the order defined by SSortedList) and the integer at that position is a unique identifier for that element's congruence class under *cong*. Hence, two elements are congruent if and only if they have the same number at their two positions in the list.

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> AsLookupTable(cong);
[ 1, 2, 3, 4, 5, 6, 3, 2, 1, 6, 5, 1 ]
```

8.3 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 soley 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.

8.3.1 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` (8.1.1).

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
```

8.3.2 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)>
```

8.3.3 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);
{(2,(3,4),3)}
```

8.3.4 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
```

8.3.5 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)
```

8.3.6 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>
```

8.3.7 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)>
```

8.3.8 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*.

At present this only works for simple and 0-simple semigroups.

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(),(1,3,2)],[(1,2),0]]);;
gap> congs := CongruencesOfSemigroup(S);;
gap> MeetSemigroupCongruences(congs[2], congs[3]);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (1,2,2)>
```

8.3.9 JoinSemigroupCongruences

▷ `JoinSemigroupCongruences(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`.

At present this only works for simple and 0-simple semigroups.

Example
```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(),(1,3,2)],[(1,2),0]]);;
gap> congs := CongruencesOfSemigroup(S);;
gap> JoinSemigroupCongruences(congs[2], congs[3]);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (3,2,2)>
```

8.4 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.

8.4.1 IsUniversalSemigroupCongruence

▷ `IsUniversalSemigroupCongruence(obj)` (category)

Returns: `true` or `false`.

This category 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
```

8.4.2 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 9

Homomorphisms

In this chapter we describe the various ways to define a homomorphism from a semigroup to another semigroup.

Support for homomorphisms in Semigroups is currently rather limited but there are plans to improve this in the future.

9.1 Isomorphisms

9.1.1 IsIsomorphicSemigroup

▷ `IsIsomorphicSemigroup(S, T)` (operation)

Returns: `true` or `false`.

This operation attempts to determine if the semigroups *S* and *T* are isomorphic, it returns `true` if they are and `false` if they are not.

At present this only works for rather small semigroups, with approximately 50 elements or less.

PLEASE NOTE: the Grape package version 4.5 or higher must be available and compiled installed for this function to work.

Example
```
gap> S:=Semigroup( [ PartialPerm( [ 1, 2, 4 ], [ 3, 5, 1 ] ),
> PartialPerm( [ 1, 3, 5 ], [ 4, 3, 2 ] ) ] );;
gap> Size(S);
11
gap> T:=SemigroupByMultiplicationTable(SmallestMultiplicationTable(S));;
gap> IsIsomorphicSemigroup(S, T);
true
```

9.1.2 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.

PLEASE NOTE: the Grape package version 4.5 or higher must be loaded for this function to work.

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 ] ]
```

9.1.3 IsomorphismSemigroups

▷ `IsomorphismSemigroups(S, T)` (operation)

Returns: An isomorphism or `fail`.

This operation returns an isomorphism from the semigroup S and to the semigroup T if it exists, and it returns `fail` if it does not.

At present this only works for Rees matrix semigroups and Rees 0-matrix semigroups.

PLEASE NOTE: the Grape package version 4.5 or higher must be available and compiled for this function to work, when the argument R is a Rees 0-matrix semigroup.

Example
```
gap> S:=PrincipalFactor(DClasses(FullTransformationMonoid(5))[2]);
<Rees 0-matrix semigroup 10x5 over Group([ (1,2,3,4), (1,2) ])>
gap> T:=PrincipalFactor(DClasses(PartitionMonoid(5))[2]);
<Rees 0-matrix semigroup 15x15 over Group([ (2,3,4,5), (4,5) ])>
gap> IsomorphismSemigroups(S, T);
fail
gap> I:=SemigroupIdeal(FullTransformationMonoid(5),
> Transformation([1,1,2,3,4]));
<regular transformation semigroup ideal of degree 5 with 1 generator>
gap> T:=PrincipalFactor(DClass(I, I.1));
<Rees 0-matrix semigroup 10x5 over Group([ (2,3,4,5), (2,5) ])>
gap> IsomorphismSemigroups(S, T);
(( 2, 4, 3, 7, 9,10, 6, 5)
(11,13,14,15), GroupHomomorphismByImages( Group( [ (1,2,3,4), (1,2)
 ] ), Group( [ (2,3,4,5), (2,5) ] ), [ (1,2,3,4), (1,2) ],
[ (2,4,5,3), (2,3) ] ), fail)
```

Chapter 10

Orbits

10.1 Looking for something in an orbit

The functions described in this section supplement the Orb package by providing methods for finding something in an orbit, with the possibility of continuing the orbit enumeration at some later point.

10.1.1 EnumeratePosition

▷ `EnumeratePosition(`*o*`, `*val*`[, `*onlynew*`])` (function)

Returns: A positive integer or `fail`.

This function returns the position of the value *val* in the orbit *o*. If *o* is closed, then this is equivalent to doing `Position(`*o*`, `*val*`)`. However, if *o* is open, then the orbit is enumerated until *val* is found, in which case the position of *val* is returned, or the enumeration ends, in which case `fail` is returned.

If the optional argument *onlynew* is present, it should be `true` or `false`. If *onlynew* is `true`, then *val* will only be checked against new points in *o*. Otherwise, every point in the *o*, not only the new ones, is considered.

10.1.2 LookForInOrb

▷ `LookForInOrb(`*o*`, `*func*`, `*start*`)` (function)

Returns: `false` or a positive integer.

The arguments of this function should be an orbit *o*, a function *func* which gets the orbit object and a point in the orbit as arguments, and a positive integer *start*. The function *func* will be called for every point in *o* starting from *start* (inclusive) and the orbit will be enumerated until *func* returns `true` or the enumeration ends. In the former case, the position of the first point in *o* for which *func* returns `true` is returned, and in the latter `false` is returned.

Example
```
gap> o:=Orb(SymmetricGroup(100), 1, OnPoints);
<open Int-orbit, 1 points>
gap> func:=function(o, x) return x=42; end;
function( o, x ) ... end
gap> LookForInOrb(o, func, 1);
42
gap> o;
<open Int-orbit, 42 points>
```

10.2 Strongly connected components of orbits

The functions described in this section supplement the Orb package by providing methods for operations related to strongly connected components of orbits.

If any of the functions is applied to an open orbit, then the orbit is completely enumerated before any further calculation is performed. It is not possible to calculate the strongly connected components of an orbit of a semigroup acting on a set until the entire orbit has been found.

10.2.1 OrbSCC

▷ `OrbSCC(o)` (function)

Returns: The strongly connected components of an orbit.

If *o* is an orbit created by the Orb package with the option `orbitgraph=true`, then `OrbSCC` returns a set of sets of positions in *o* corresponding to its strongly connected components.

See also `OrbSCCLookup` (10.2.2) and `OrbSCCTruthTable` (10.2.3).

Example

```
gap> S:=FullTransformationSemigroup(4);;
gap> o:=LambdaOrb(S);
<open orbit, 1 points with Schreier tree with log>
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3, 4, 5, 6 ], [ 7, 8, 9, 10, 11, 12 ],
  [ 13, 14, 15, 16 ] ]
```

10.2.2 OrbSCCLookup

▷ `OrbSCCLookup(o)` (function)

Returns: A lookup table for the strongly connected components of an orbit.

If *o* is an orbit created by the Orb package with the option `orbitgraph=true`, then `OrbSCCLookup` returns a lookup table for its strongly connected components. More precisely, `OrbSCCLookup(o)[i]` equals the index of the strongly connected component containing `o[i]`.

See also `OrbSCC` (10.2.1) and `OrbSCCTruthTable` (10.2.3).

Example

```
gap> S:=FullTransformationSemigroup(4);;
gap> o:=LambdaOrb(S);;
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3, 4, 5, 6 ], [ 7, 8, 9, 10, 11, 12 ],
  [ 13, 14, 15, 16 ] ]
gap> OrbSCCLookup(o);
[ 1, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5 ]
gap> OrbSCCLookup(o)[1]; OrbSCCLookup(o)[4]; OrbSCCLookup(o)[7];
1
3
4
```

10.2.3 OrbSCCTruthTable

▷ `OrbSCCTruthTable(o)` (function)

Returns: Truth tables for strongly connected components of an orbit.

If *o* is an orbit created by the Orb package with the option `orbitgraph=true`, then `OrbSCCTruthTable` returns a list of boolean lists such that `OrbSCCTruthTable(o)[i][j]` is `true` if `j` belongs to `OrbSCC(o)[i]`.

See also `OrbSCC` (10.2.1) and `OrbSCCLookup` (10.2.2).

Example
```
gap> S:=FullTransformationSemigroup(4);;
gap> o:=LambdaOrb(S);;
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3, 4, 5, 6 ], [ 7, 8, 9, 10, 11, 12 ],
  [ 13, 14, 15, 16 ] ]
gap> OrbSCCTruthTable(o);
[ [ true, false, false, false, false, false, false, false, false,
      false, false, false, false, false, false, false ],
  [ false, true, false, false, false, false, false, false, false,
      false, false, false, false, false, false, false ],
  [ false, false, true, true, true, true, false, false, false, false,
      false, false, false, false, false, false ],
  [ false, false, false, false, false, false, true, true, true, true,
      true, true, false, false, false, false ],
  [ false, false, false, false, false, false, false, false, false,
      false, false, false, true, true, true, true ] ]
```

10.2.4 ReverseSchreierTreeOfSCC

▷ `ReverseSchreierTreeOfSCC(o, i)` (function)

Returns: The reverse Schreier tree corresponding to the `i`th strongly connected component of an orbit.

If *o* is an orbit created by the Orb package with the option `orbitgraph=true` and action `act`, and *i* is a positive integer, then `ReverseSchreierTreeOfSCC(o, i)` returns a pair `[ gen, pos ]` of lists with `Length(o)` entries such that

Example
```
act(o[j], o!.gens[gen[j]])=o[pos[j]].
```

The pair `[ gen, pos ]` corresponds to a tree with root `OrbSCC(o)[i][1]` and a path from every element of `OrbSCC(o)[i]` to the root.

See also `OrbSCC` (10.2.1), `TraceSchreierTreeOfSCCBack` (10.2.6), `SchreierTreeOfSCC` (10.2.5), and `TraceSchreierTreeOfSCCForward` (10.2.7).

Example
```
gap> S:=Semigroup(Transformation( [ 2, 2, 1, 4, 4 ] ),
> Transformation( [ 3, 3, 3, 4, 5 ] ),
> Transformation( [ 5, 1, 4, 5, 5 ] ) );;
gap> o:=Orb(S, [1..4], OnSets, rec(orbitgraph:=true, schreier:=true));;
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3, 5, 6, 7, 11 ], [ 4 ], [ 8 ], [ 9 ], [ 10, 12 ] ]
gap> ReverseSchreierTreeOfSCC(o, 3);
[ [ ,, fail,, 2, 1, 2,,,, 1 ], [ ,, fail,, 3, 5, 3,,,, 7 ] ]
gap> ReverseSchreierTreeOfSCC(o, 7);
[ [ ,,,,,,,,, fail,, 3 ], [ ,,,,,,,,, fail,, 10 ] ]
gap> OnSets(o[11], Generators(S)[1]);
[ 1, 4 ]
```

```
gap> Position(o, last);
7
```

10.2.5 SchreierTreeOfSCC

▷ SchreierTreeOfSCC(*o*, *i*) (function)

Returns: The Schreier tree corresponding to the *i*th strongly connected component of an orbit.

If *o* is an orbit created by the Orb package with the option orbitgraph=true and action act, and *i* is a positive integer, then SchreierTreeOfSCC(*o*, *i*) returns a pair [gen, pos] of lists with Length(o) entries such that

Example
```
act(o[pos[j]], o!.gens[gen[j]])=o[j].
```

The pair [gen, pos] corresponds to a tree with root OrbSCC(o)[i][1] and a path from the root to every element of OrbSCC(o)[i].

See also OrbSCC (10.2.1), TraceSchreierTreeOfSCCBack (10.2.6), ReverseSchreierTreeOfSCC (10.2.4), and TraceSchreierTreeOfSCCForward (10.2.7).

Example
```
gap> S:=Semigroup(Transformation( [ 2, 2, 1, 4, 4 ] ),
> Transformation( [ 3, 3, 3, 4, 5 ] ),
> Transformation( [ 5, 1, 4, 5, 5 ] ) );;
gap> o:=Orb(S, [1..4], OnSets, rec(orbitgraph:=true, schreier:=true));;
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3, 5, 6, 7, 11 ], [ 4 ], [ 8 ], [ 9 ], [ 10, 12 ] ]
gap> SchreierTreeOfSCC(o, 3);
[ [ ,, fail,, 1, 3, 1,,,, 2 ], [ ,, fail,, 7, 5, 3,,,, 6 ] ]
gap> SchreierTreeOfSCC(o, 7);
[ [ ,,,,,,,,, fail,, 1 ], [ ,,,,,,,,, fail,, 10 ] ]
gap> OnSets(o[6], Generators(S)[2]);
[ 3, 5 ]
gap> Position(o, last);
11
```

10.2.6 TraceSchreierTreeOfSCCBack

▷ TraceSchreierTreeOfSCCBack(*orb*, *m*, *nr*) (operation)

Returns: A word in the generators.

orb must be an orbit object with a Schreier tree and orbit graph, that is, the options schreier and orbitgraph must have been set to true during the creation of the orbit, *m* must be the number of a strongly connected component of *orb*, and nr must be the number of a point in the *m*th strongly connect component of *orb*.

This operation traces the result of ReverseSchreierTreeOfSCC (10.2.4) and with arguments *orb* and *m* and returns a word in the generators that maps the point with number *nr* to the first point in the *m*th strongly connected component of *orb*. Here, a word is a list of integers, where positive integers are numbers of generators. See also OrbSCC (10.2.1), ReverseSchreierTreeOfSCC (10.2.4), SchreierTreeOfSCC (10.2.5), and TraceSchreierTreeOfSCCForward (10.2.7).

Example
```
gap> S:=Semigroup(Transformation( [ 1, 3, 4, 1 ] ),
> Transformation( [ 2, 4, 1, 2 ] ),
```

```
> Transformation( [ 3, 1, 1, 3 ] ),
> Transformation( [ 3, 3, 4, 1 ] ) );;
gap> o:=Orb(S, [1..4], OnSets, rec(orbitgraph:=true, schreier:=true));;
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3 ], [ 4, 5, 6, 7, 8 ], [ 9, 10, 11, 12 ] ]
gap> ReverseSchreierTreeOfSCC(o, 4);
[ [ ,,, fail, 4, 1, 1, 3 ], [ ,,, fail, 4, 4, 4, 4 ] ]
gap> TraceSchreierTreeOfSCCBack(o, 4, 7);
[ 1 ]
gap> TraceSchreierTreeOfSCCBack(o, 4, 8);
[ 3 ]
```

10.2.7 TraceSchreierTreeOfSCCForward

▷ `TraceSchreierTreeOfSCCForward(`*orb*`, `*m*`, `*nr*`)` (operation)

Returns: A word in the generators.

orb must be an orbit object with a Schreier tree and orbit graph, that is, the options `schreier` and `orbitgraph` must have been set to `true` during the creation of the orbit, *m* must be the number of a strongly connected component of *orb*, and `nr` must be the number of a point in the *m*th strongly connect component of *orb*.

This operation traces the result of `SchreierTreeOfSCC` (10.2.5) and with arguments *orb* and *m* and returns a word in the generators that maps the first point in the *m*th strongly connected component of *orb* to the point with number *nr*. Here, a word is a list of integers, where positive integers are numbers of generators. See also `OrbSCC` (10.2.1), `ReverseSchreierTreeOfSCC` (10.2.4), `SchreierTreeOfSCC` (10.2.5), and `TraceSchreierTreeOfSCCBack` (10.2.6).

Example

```
gap> S:=Semigroup(Transformation( [ 1, 3, 4, 1 ] ),
> Transformation( [ 2, 4, 1, 2 ] ),
> Transformation( [ 3, 1, 1, 3 ] ),
> Transformation( [ 3, 3, 4, 1 ] ) );;
gap> o:=Orb(S, [1..4], OnSets, rec(orbitgraph:=true, schreier:=true));;
gap> OrbSCC(o);
[ [ 1 ], [ 2 ], [ 3 ], [ 4, 5, 6, 7, 8 ], [ 9, 10, 11, 12 ] ]
gap> SchreierTreeOfSCC(o, 4);
[ [ ,,, fail, 1, 2, 2, 4 ], [ ,,, fail, 4, 4, 6, 4 ] ]
gap> TraceSchreierTreeOfSCCForward(o, 4, 8);
[ 4 ]
gap> TraceSchreierTreeOfSCCForward(o, 4, 7);
[ 2, 2 ]
```

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, 68

[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. 22

[FL98] D.G. Fitzgerald and J. Leech. Dual symmetric inverse monoids and representation theory. *J. Austral. Math. Soc. A*, 64:345–67, 1998. 21, 103

[GGR68] N. Graham, R. Graham, and J. Rhodes. Maximal subsemigroups of finite semigroups. *J. Combinatorial Theory*, 4:203–209, 1968. 7, 61, 62

[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. 7, 74, 85, 119, 122, 134, 135

[Sch92] Boris M. Schein. The minimal degree of a finite inverse semigroup. *Trans. Amer. Math. Soc.*, 333(2):877–888, 1992. 7, 92

Index

* (for bipartitions), 104
\<
    for Green's classes, 44
< (for bipartitions), 104
= (for bipartitions), 105
\^
    for an matrix over finite field group and matrix over finite field, 130

AsBipartition, 101
AsBipartitionSemigroup, 19
AsBlockBijection, 103
AsBlockBijectionSemigroup, 19
AsLookupTable, 134
AsMatrix
    for a matrix over finite field, 129
AsMatrixGroup, 131
AsMatrixSemigroup, 19
AsPartialPerm
    for a bipartition, 104
AsPartialPermSemigroup, 19
AsPermutation
    for a bipartition, 104
AsRMSCongruenceByLinkedTriple, 138
AsRZMSCongruenceByLinkedTriple, 138
AsSemigroupCongruenceByGeneratingPairs, 138
AsTransformation
    for a bipartition, 103
AsTransformationSemigroup, 19

BaseDomain
    for a matrix over finite field, 129
Bipartition, 98
BipartitionByIntRep, 99
BipartitionFamily, 98
BlocksNC, 112
BrauerMonoid, 22

CanonicalForm
    for a free inverse semigroup element, 121
CanonicalRepresentative, 137
CharacterTableOfInverseSemigroup, 93
ClosureInverseSemigroup, 14
ClosureSemigroup, 14
ComponentRepsOfPartialPermSemigroup, 67
ComponentRepsOfTransformationSemigroup, 66
ComponentsOfPartialPermSemigroup, 67
ComponentsOfTransformationSemigroup, 66
CongruenceClasses, 133
CongruenceClassOfElement, 133
CongruencesOfSemigroup, 134
ConstructingFilter
    for a matrix over finite field, 130
CyclesOfPartialPerm, 68
CyclesOfPartialPermSemigroup, 68
CyclesOfTransformationSemigroup, 66

DClass, 36
DClasses, 39
DClassNC, 37
DClassOfHClass, 35
DClassOfLClass, 35
DClassOfRClass, 35
DClassReps, 43
DegreeOfBipartition, 106
DegreeOfBipartitionCollection, 106
DegreeOfBipartitionSemigroup, 118
DegreeOfBlocks, 113
DegreeOfMatrixOverFiniteField
    for a matrix over finite field, 129
DotDClasses, 95
    for a semigroup, 95
DotSemilatticeOfIdempotents, 96
DualSymmetricInverseMonoid, 24
DualSymmetricInverseSemigroup, 24

EndomorphismsPartition, 22

EnumeratePosition, 143
EvaluateWord, 33
ExtRepOfBipartition, 107
ExtRepOfBlocks, 112

FactorisableDualSymmetricInverseSemigroup, 23
Factorization, 34
FreeBand
  for a given rank, 122
  for a list of names, 122
  for various names, 122
FreeInverseSemigroup
  for a given rank, 120
  for a list of names, 120
  for various names, 120
FullMatrixSemigroup, 24

GeneralLinearSemigroup, 24
Generators, 55
GeneratorsOfSemigroupIdeal, 31
GeneratorsSmallest
  for a transformation semigroup, 70
GLS, 24
GreensDClasses, 39
GreensDClassOfElement, 36
  for a free band and a free band element, 124
GreensDClassOfElementNC, 37
GreensHClasses, 39
GreensHClassOfElement, 36
  for a Rees matrix semigroup, 36
GreensHClassOfElementNC, 37
GreensJClasses, 39
GreensLClasses, 39
GreensLClassOfElement, 36
GreensLClassOfElementNC, 37
GreensRClasses, 39
GreensRClassOfElement, 36
GreensRClassOfElementNC, 37
GroupHClass, 38
GroupOfUnits, 56

HClass, 36
  for a Rees matrix semigroup, 36
HClasses, 39
HClassNC, 37
HClassReps, 43

IdempotentGeneratedSubsemigroup, 59
Idempotents, 57
IdentityBipartition, 99
InfoSemigroups, 9
InjectionPrincipalFactor, 45
InverseLeftBlocks, 114
InverseRightBlocks, 114
InverseSubsemigroupByProperty, 16
IrredundantGeneratingSubset, 59
IsAperiodicSemigroup, 79
IsBand, 71
IsBipartition, 98
IsBipartitionCollection, 98
IsBipartitionMonoid, 116
IsBipartitionSemigroup, 116
IsBipartitionSemigroupGreensClass, 53
IsBlockBijection, 111
IsBlockBijectionMonoid, 117
IsBlockBijectionSemigroup, 117
IsBlockGroup, 72
IsBrandtSemigroup, 84
IsCliffordSemigroup, 84
IsCombinatorialSemigroup, 79
IsCommutativeSemigroup, 72
IsCompletelyRegularSemigroup, 73
IsCompletelySimpleSemigroup, 80
IsCongruenceFreeSemigroup, 74
IsDTrivial, 79
IsDualTransBipartition, 110
IsEUnitaryInverseSemigroup, 85
IsFactorisableSemigroup, 86
IsFreeBand
  for a given semigroup, 123
IsFreeBandCategory, 123
IsFreeBandElement, 123
IsFreeBandSubsemigroup, 123
IsFreeInverseSemigroup, 120
IsFreeInverseSemigroupCategory, 120
IsFreeInverseSemigroupElement, 120
IsGreensClassNC, 53
IsGreensDLeq, 54
IsGroupAsSemigroup, 74
IsHTrivial, 79
IsIdempotentGenerated, 74
IsIsomorphicSemigroup, 141
IsJoinIrreducible, 86
IsLeftSimple, 75

IsLeftZeroSemigroup, 76
IsLinkedTriple, 137
IsLTrivial, 79
IsMajorantlyClosed, 87
IsMatrixMonoid, 126
IsMatrixOverFiniteField, 126
IsMatrixOverFiniteFieldCollection, 127
IsMatrixOverFiniteFieldGroup, 130
IsMatrixSemigroup, 126
IsMatrixSemigroupGreensClass, 54
IsMaximalSubsemigroup, 62
IsMonogenicInverseSemigroup, 87
IsMonogenicSemigroup, 76
IsMonoidAsSemigroup, 77
IsomorphismBipartitionMonoid, 20
IsomorphismBipartitionSemigroup, 20
IsomorphismBlockBijectionMonoid, 21
IsomorphismBlockBijectionSemigroup, 21
IsomorphismMatrixGroup, 131
IsomorphismMatrixSemigroup, 21
IsomorphismPermGroup, 19
IsomorphismReesMatrixSemigroup, 45
IsomorphismSemigroups, 142
IsOrthodoxSemigroup, 77
IsPartialPermBipartition, 110
IsPartialPermBipartitionMonoid, 117
IsPartialPermBipartitionSemigroup, 117
IsPartialPermSemigroupGreensClass, 53
IsPermBipartition, 110
IsPermBipartitionGroup, 118
IsRectangularBand, 78
IsRegularClass, 46
IsRegularSemigroup, 78
IsRightSimple, 75
IsRightZeroSemigroup, 78
IsRMSCongruenceByLinkedTriple, 135
IsRTrivial, 79
IsRZMSCongruenceByLinkedTriple, 135
IsSemiBand, 74
IsSemigroupWithAdjoinedZero, 79
IsSemigroupWithCommutingIdempotents, 72
IsSemilattice, 80
IsSimpleSemigroup, 80
IsSynchronizingSemigroup, 81
IsSynchronizingTransformationCollection, 81
IsTransBipartition, 109
IsTransformationSemigroupGreensClass, 53
IsTransitive
    for a transformation semigroup and a pos int, 66
    for a transformation semigroup and a set, 66
IsUniformBlockBijection, 111
IsUniversalSemigroupCongruence, 139
IsZeroGroup, 81
IsZeroRectangularBand, 82
IsZeroSemigroup, 82
IsZeroSimpleSemigroup, 83
IteratorFromGeneratorsFile, 11
IteratorOfDClasses, 41
IteratorOfDClassReps, 41
IteratorOfHClasses, 41
IteratorOfHClassReps, 41
IteratorOfLClasses, 41
IteratorOfLClassReps, 41
IteratorOfRClasses, 42
IteratorOfRClassReps, 41

JClasses, 39
JoinIrreducibleDClasses, 88
JoinSemigroupCongruences, 139
JonesMonoid, 23

LargestElementSemigroup, 69
LClass, 36
LClasses, 39
LClassNC, 37
LClassOfHClass, 35
LClassReps, 43
LeftBlocks, 108
LeftInverse
    for a matrix over finite field, 128
LeftOne
    for a bipartition, 99
LeftProjection, 99
LeftZeroSemigroup, 29
LookForInOrb, 143

MajorantClosure, 88
MatrixSemigroup, 126
MaximalDClasses, 51
MaximalSubsemigroups
    for a Rees (0-)matrix semigroup, and a group, 61

for an acting semigroup, 60
MeetSemigroupCongruences, 138
MinimalDClass, 51
MinimalIdeal, 62
MinimalIdealGeneratingSet, 31
MinimalWord
for free inverse semigroup element, 122
Minorants, 89
MonogenicSemigroup, 27
MultiplicativeNeutralElement
for an H-class, 52
MultiplicativeZero, 64
MunnSemigroup, 25

NaturalLeqBlockBijection, 105
NaturalLeqPartialPermBipartition, 105
NewIdentityMatrixOverFiniteField, 127
NewMatrixOverFiniteField
for a filter, a field, an integer, and a list, 127
NewZeroMatrixOverFiniteField, 127
Normalizer
for a perm group, semigroup, record, 68
for a semigroup, record, 68
NrBlocks
for a bipartition, 109
for blocks, 109
NrCongruenceClasses, 133
NrDClasses, 48
NrHClasses, 48
NrIdempotents, 58
NrLClasses, 48
NrLeftBlocks, 108
NrRClasses, 48
NrRegularDClasses, 47
NrRightBlocks, 109
NrTransverseBlocks
for a bipartition, 107
for blocks, 112

OnLeftBlocks, 113
OnRightBlocks, 113
OnRightBlocksBipartitionByPerm, 106
OrbSCC, 144
OrbSCCLookup, 144
OrbSCCTruthTable, 144
OrderEndomorphisms
monoid of order preserving transformations, 25

PartialOrderOfDClasses, 49
PartialPermLeqBipartition, 105
PartialTransformationSemigroup, 24
PartitionMonoid, 22
PermLeftBlocks, 114
PermLeftQuoBipartition, 106
PermRightBlocks, 114
POI
monoid of order preserving partial perms, 25
POPI
monoid of orientation preserving partial perms, 25
PrimitiveIdempotents, 90
PrincipalFactor, 46

Random
for a semigroup, 64
RandomBinaryRelationMonoid, 13
RandomBinaryRelationSemigroup, 13
RandomBipartition, 101
RandomBipartitionMonoid, 13
RandomBipartitionSemigroup, 13
RandomInverseMonoid, 12
RandomInverseSemigroup, 12
RandomMatrixMonoid, 13
RandomMatrixSemigroup, 13
RandomPartialPermMonoid, 13
RandomPartialPermSemigroup, 13
RandomTransformationMonoid, 12
RandomTransformationSemigroup, 12
RankOfBipartition, 107
RankOfBlocks, 112
RClass, 36
RClasses, 39
RClassNC, 37
RClassOfHClass, 35
RClassReps, 43
ReadGenerators, 10
RectangularBand, 28
RegularBinaryRelationSemigroup, 27
RegularDClasses, 47
RepresentativeOfMinimalDClass, 63
RepresentativeOfMinimalIdeal, 63
ReverseSchreierTreeOfSCC, 145

RightBlocks, 107
RightCosetsOfInverseSemigroup, 90
RightInverse
    for a matrix over finite field, 128
RightOne
    for a bipartition, 100
RightProjection, 100
RightZeroSemigroup, 29
RMSCongruenceByLinkedTriple, 136
RMSCongruenceClassByLinkedTriple, 136
RowRank
    for a matrix over finite field, 128
RowSpaceBasis
    for a matrix over finite field, 128
RowSpaceTransformation
    for a matrix over finite field, 128
RowSpaceTransformationInv
    for a matrix over finite field, 128
RZMSCongruenceByLinkedTriple, 136
RZMSCongruenceClassByLinkedTriple, 136

SameMinorantsSubgroup, 91
SchreierTreeOfSCC, 146
SchutzenbergerGroup, 50
SemigroupCongruence, 132
SemigroupIdeal, 30
Semigroups package overview, 6
SemigroupsDir, 10
SemigroupsMakeDoc, 8
SemigroupsOptionsRec, 18
SemigroupsTestAll, 9
SemigroupsTestInstall, 9
SemigroupsTestManualExamples, 9
SingularBrauerMonoid, 22
SingularDualSymmetricInverseSemigroup, 24
SingularFactorisableDualSymmetric-InverseSemigroup, 23
SingularJonesMonoid, 23
SingularPartitionMonoid, 22
SingularTransformationMonoid, 26
SingularTransformationSemigroup, 26
SLS, 25
SmallerDegreePartialPerm-Representation, 91
SmallestElementSemigroup, 69
SmallestMultiplicationTable, 141
SmallGeneratingSet, 64
SmallInverseMonoidGeneratingSet, 65
SmallInverseSemigroupGeneratingSet, 65
SmallMonoidGeneratingSet, 64
SmallSemigroupGeneratingSet, 64
SpecialLinearSemigroup, 25
Splash, 94
Star, 100
StarOp, 100
StructureDescription
    for an H-class, 54
StructureDescriptionMaximalSubgroups, 52
StructureDescriptionSchutzenberger-Groups, 52
SubsemigroupByProperty
    for a semigroup and function, 15
    for a semigroup, function, and limit on the size of the subsemigroup, 15
SupersemigroupOfIdeal, 32

TemperleyLiebMonoid, 23
TikzBipartition, 115
TikzBlocks, 116
TraceSchreierTreeOfSCCBack, 146
TraceSchreierTreeOfSCCForward, 147
TransposedMatImmutable
    for a matrix over finite field, 129

UnderlyingSemigroupOfSemigroupWith-AdjoinedZero, 70
UniversalSemigroupCongruence, 140

VagnerPrestonRepresentation, 92

WriteGenerators, 10

ZeroSemigroup, 28