# Semigroups

## A package for semigroups and monoids

5.7.1

11 November 2026

**James Mitchell**
**Marina Anagnostopoulou-Merkouri**
**Thomas Breuer**
**Stuart Burrell**
**Reinis Cirpons**
**Tom Conti-Leslie**
**Joseph Edwards**
**Attila Egri-Nagy**
**Jan Engelhardt**
**Luna Elliott**
**Fernando Flores Brito**
**Tillman Froehlich**
**Nick Ham**
**Robert Hancock**
**Max Horn**
**Christopher Jefferson**
**Julius Jonusas**
**William Kenyon**
**Alex Levine**
**Chinmaya Nagpal**

**Olexandr Konovalov**
**Artemis Konstantinidi**
**Hyeokjun Kwon**
**Dima V. Pasechnik**
**Yann Peresse**
**Markus Pfeiffer**
**Pramoth Ragavan**
**Joesph Daynger Ruiz**
**Christopher Russell**
**Jack Schmidt**
**Sergio Siccha**
**Finn Smith**
**Ben Spiers**
**James Sullivan**
**James Swent**
**James Sullivan**
**Nicolas Thiéry**
**Maria Tsalakou**
**Joe Ward**
**Chris Wensley**
**Murray Whyte**
**Wilf A. Wilson**
**Tianrun Yang**
**Michael Young**
**Fabian Zickgraf**

**James Mitchell** Email: jdm3@st-andrews.ac.uk
Homepage: https://jdbm.me
Address: Mathematical Institute, North Haugh, St Andrews, Fife, KY16 9SS, Scotland

**Marina Anagnostopoulou-Merkouri** Email: marina.anagnostopoulou-merkouri@bristol.ac.uk
Homepage: https://marinaanagno.github.io

**Thomas Breuer** Email: sam@math.rwth-aachen.de

Homepage: https://www.math.rwth-aachen.de/~Thomas.Breuer/

**Stuart Burrell** Email: stuartburrell1994@gmail.com
Homepage: https://stuartburrell.github.io

**Reinis Cirpons** Email: reinis.cirpons@inria.fr
Homepage: https://reinisc.id.lv/
Address: LS2N, UFR Sciences et Techniques, 2, rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, France

**Tom Conti-Leslie** Email: tom.contileslie@gmail.com
Homepage: https://tomcontileslie.com/

**Joseph Edwards** Email: jde1@st-andrews.ac.uk
Homepage: https://joseph-edwards.github.io/
Address: Mathematical Institute, North Haugh, St Andrews, Fife, KY16 9SS, Scotland

**Attila Egri-Nagy** Email: attila@egri-nagy.hu
Homepage: http://www.egri-nagy.hu

**Jan Engelhardt** Email: jengelh@inai.de

**Luna Elliott** Email: luna.elliott142857@gmail.com
Homepage: https://le27.github.io/L-Elliott/

**Fernando Flores Brito** Email: ffloresbrito@gmail.com

**Tillman Froehlich** Email: trf1@st-andrews.ac.uk

**Nick Ham** Email: nicholas.charles.ham@gmail.com
Homepage: https://n-ham.github.io

**Robert Hancock** Email: robert.hancock@maths.ox.ac.uk
Homepage: https://sites.google.com/view/robert-hancock/

**Max Horn** Email: mhorn@rptu.de
Homepage: https://www.quendi.de/math
Address: Fachbereich Mathematik, RPTU Kaiserslautern-Landau, Gottlieb-Daimler-Straße 48, 67663 Kaiserslautern, Germany

**Christopher Jefferson** Email: caj21@st-andrews.ac.uk
Homepage: https://heather.cafe/
Address: Jack Cole Building, North Haugh, St Andrews, Fife, KY16 9SX, Scotland

**Julius Jonusas** Email: j.jonusas@gmail.com
Homepage: http://julius.jonusas.work

**William Kenyon** Email: wdk3@st-andrews.ac.uk

**Alex Levine** Email: alexlevinemaths@gmail.com
Homepage: https://alex-a-levine.github.io

**Olexandr Konovalov** Email: obk1@st-andrews.ac.uk
Homepage: https://olexandr-konovalov.github.io/
Address: Jack Cole Building, North Haugh, St Andrews, Fife, KY16 9SX, Scotland

**Dima V. Pasechnik** Email: dmitrii.pasechnik@cs.ox.ac.uk
Homepage: https://pasechnik.info/dima
Address: Pembroke College, St. Aldates, Oxford OX1 1DW, England

**Yann Peresse** Email: y.peresse@herts.ac.uk

**Markus Pfeiffer** Email: markus.pfeiffer@morphism.de
Homepage: https://markusp.morphism.de/

**Pramoth Ragavan** Email: pramoth.ragavan@gmail.com

**Joesph Daynger Ruiz** Email: jdruiz@arizona.edu

**Jack Schmidt** Email: jack.schmidt@uky.edu
Homepage: https://www.ms.uky.edu/~jack/

**Sergio Siccha** Email: sergio.siccha@gmail.com

**Finn Smith** Email: fls3@st-andrews.ac.uk
Homepage: https://github.com/flsmith/
Address: Mathematical Institute, North Haugh, St Andrews, Fife, KY16 9SS, Scotland

**James Sullivan** Email: js521@st-andrews.ac.uk

**James Swent** Email: jake@osprey21.com

**Nicolas Thiéry** Email: nthiery@users.sf.net
Homepage: https://nicolas.thiery.name/

**Maria Tsalakou** Email: mt200@st-andrews.ac.uk
Homepage: https://mariatsalakou.github.io/

Address: Mathematical Institute, North Haugh, St Andrews, Fife, KY16 9SS, Scotland

**Joe Ward** Email: josephdward@hotmail.co.uk

**Chris Wensley** Email: cdwensley.maths@btinternet.com

**Murray Whyte** Email: mw231@st-andrews.ac.uk
Address: Mathematical Institute, North Haugh, St Andrews, Fife, KY16 9SS, Scotland

**Wilf A. Wilson** Email: gap@wilf-wilson.net
Homepage: https://wilf.me

**Michael Young** Email: mct25@st-andrews.ac.uk
Homepage: https://myoung.uk/work/
Address: Jack Cole Building, North Haugh, St Andrews, Fife, KY16 9SX, Scotland

**Fabian Zickgraf** Email: f.zickgraf@dashdos.com

# Abstract

The Semigroups package is a GAP package for semigroups, and monoids. There are particularly efficient methods for finitely presented semigroups and monoids, and for semigroups and monoids consisting of transformations, partial permutations, bipartitions, partitioned binary relations, subsemigroups of regular Rees 0-matrix semigroups, and matrices of various semirings including boolean matrices, matrices over finite fields, and certain tropical matrices. Semigroups contains efficient methods for creating semigroups, monoids, and inverse semigroups and monoids, calculating their Green's structure, ideals, size, elements, group of units, small generating sets, testing membership, finding the inverses of a regular element, factorizing elements over the generators, and so on. It is possible to test if a semigroup satisfies a particular property, such as if it is regular, simple, inverse, completely regular, and a large number of further properties. There are methods for finding presentations for a semigroup, the congruences of a semigroup, the maximal subsemigroups of a finite semigroup, smaller degree partial permutation representations, and the character tables of inverse semigroups. There are functions for producing pictures of the Green's structure of a semigroup, and for drawing graphical representations of certain types of elements.

# Copyright

© by J. D. Mitchell et al.

Semigroups is free software; you can redistribute it and/or modify it, under the terms of the GNU General Public License, version 3 of the License, or (at your option) any later, version.

# Acknowledgements

The authors of the Semigroups package would like to thank:

**Manuel Delgado**
who contributed to the function `DotString` (16.1.1).

**Casey Donoven and Rhiannon Dougall**
for their contribution to the development of the algorithms for maximal subsemigroups and smaller degree partial permutation representations.

**James East**
who contributed to the part of the package relating to bipartitions. We also thank the University of Western Sydney for their support of the development of this part of the package.

**Zak Mesyan**
who contributed to the code for graph inverse semigroups; see Section 7.10.

**Yann Péresse and Yanhui Wang**
who contributed to the attribute `MunnSemigroup` (7.2.1).

**Jhevon Smith and Ben Steinberg**
who contributed the function `CharacterTableOfInverseSemigroup` (11.15.10).

We would also like to acknowledge the support of: EPSRC grant number GR/S/56085/01; the Carnegie Trust for the Universities of Scotland for funding the PhD scholarships of Julius Jonušas and Wilf A. Wilson when

they worked on this project; the Engineering and Physical Sciences Research Council (EPSRC) for funding the PhD scholarships of F. Smith (EP/N509759/1) and M. Young (EP/M506631/1) when they worked on this project.

# Contents


**1 The Semigroups package** **8**
1.1 Introduction . . . . . 8
1.2 Overview . . . . . 8

**2 Installing Semigroups** **10**
2.1 For those in a hurry . . . . . 10
2.2 Compiling the kernel module . . . . . 11
2.3 Rebuilding the documentation . . . . . 11
2.4 Testing your installation . . . . . 12
2.5 More information during a computation . . . . . 13

**3 Bipartitions and blocks** **14**
3.1 The family and categories of bipartitions . . . . . 15
3.2 Creating bipartitions . . . . . 16
3.3 Changing the representation of a bipartition . . . . . 19
3.4 Operators for bipartitions . . . . . 23
3.5 Attributes for bipartitons . . . . . 24
3.6 Creating blocks and their attributes . . . . . 32
3.7 Actions on blocks . . . . . 34
3.8 Semigroups of bipartitions . . . . . 34

**4 Partitioned binary relations (PBRs)** **37**
4.1 The family and categories of PBRs . . . . . 37
4.2 Creating PBRs . . . . . 37
4.3 Changing the representation of a PBR . . . . . 39
4.4 Operators for PBRs . . . . . 42
4.5 Attributes for PBRs . . . . . 42
4.6 Semigroups of PBRs . . . . . 47

**5 Matrices over semirings** **49**
5.1 Creating matrices over semirings . . . . . 50
5.2 Operators for matrices over semirings . . . . . 58
5.3 Boolean matrices . . . . . 58
5.4 Matrices over finite fields . . . . . 68
5.5 Matrices over the integers . . . . . 69
5.6 Max-plus and min-plus matrices . . . . . 71

5.7 Matrix semigroups . . . . . 72

**6 Semigroups and monoids defined by generating sets 75**
6.1 Underlying algorithms . . . . . 75
6.2 Semigroups represented by generators . . . . . 78
6.3 Options when creating semigroups . . . . . 78
6.4 Subsemigroups and supersemigroups . . . . . 80
6.5 Changing the representation of a semigroup . . . . . 83
6.6 Random semigroups . . . . . 92

**7 Standard examples 95**
7.1 Transformation semigroups . . . . . 95
7.2 Semigroups of partial permutations . . . . . 98
7.3 Semigroups of bipartitions . . . . . 100
7.4 Standard PBR semigroups . . . . . 107
7.5 Semigroups of matrices over a finite field . . . . . 107
7.6 Semigroups of boolean matrices . . . . . 109
7.7 Semigroups of matrices over a semiring . . . . . 111
7.8 Examples in various representations . . . . . 112
7.9 Free bands . . . . . 118
7.10 Graph inverse semigroups . . . . . 121
7.11 Free inverse semigroups . . . . . 126

**8 Standard constructions 130**
8.1 Products of semigroups . . . . . 130
8.2 Dual semigroups . . . . . 131
8.3 Strong semilattices of semigroups . . . . . 133
8.4 McAlister triple semigroups . . . . . 136

**9 Ideals 141**
9.1 Creating ideals . . . . . 141
9.2 Attributes of ideals . . . . . 142

**10 Green's relations 145**
10.1 Creating Green's classes and representatives . . . . . 145
10.2 Iterators and enumerators of classes and representatives . . . . . 156
10.3 Properties of Green's classes . . . . . 157
10.4 Attributes of Green's classes . . . . . 159
10.5 Operations for Green's relations and classes . . . . . 165

**11 Attributes and operations for semigroups 167**
11.1 Accessing the elements of a semigroup . . . . . 167
11.2 Cayley graphs . . . . . 169
11.3 Random elements of a semigroup . . . . . 169
11.4 Properties of elements in a semigroup . . . . . 170
11.5 Operations for elements in a semigroup . . . . . 171
11.6 Expressing semigroup elements as words in generators . . . . . 172
11.7 Generating sets . . . . . 175

11.8 Minimal ideals and multiplicative zeros . . . 181
11.9 Group of units and identity elements . . . 184
11.10 Idempotents . . . 185
11.11 Maximal subsemigroups . . . 188
11.12 Attributes of transformations and transformation semigroups . . . 191
11.13 Attributes of partial perms and partial perm semigroups . . . 195
11.14 Attributes of Rees (0-)matrix semigroups . . . 197
11.15 Attributes of inverse semigroups . . . 198
11.16 Nambooripad partial order . . . 205
11.17 Monoid character table and Cartan matrix . . . 206

**12 Properties of semigroups** **212**
12.1 Arbitrary semigroups . . . 212
12.2 Inverse semigroups . . . 228

**13 Congruences** **234**
13.1 Semigroup congruence objects . . . 234
13.2 Creating congruences . . . 236
13.3 Congruence classes . . . 238
13.4 Finding the congruences of a semigroup . . . 242
13.5 Comparing congruences . . . 253
13.6 Congruences on Rees matrix semigroups . . . 254
13.7 Congruences on inverse semigroups . . . 258
13.8 Congruences on graph inverse semigroups . . . 261
13.9 Rees congruences . . . 264
13.10 Universal and trivial congruences . . . 265

**14 Semigroup homomorphisms** **267**
14.1 Homomorphisms of arbitrary semigroups . . . 267
14.2 Isomorphisms of arbitrary semigroups . . . 271
14.3 Isomorphisms of Rees (0-)matrix semigroups . . . 278

**15 Finitely presented semigroups and Tietze transformations** **282**
15.1 Changing representation for words and strings . . . 282
15.2 Helper functions . . . 284
15.3 Creating Tietze transformation objects . . . 286
15.4 Printing Tietze transformation objects . . . 289
15.5 Changing Tietze transformation objects . . . 291
15.6 Converting a Tietze transformation object into a fp semigroup . . . 294
15.7 Automatically simplifying a Tietze transformation object . . . 296
15.8 Automatically simplifying an fp semigroup . . . 298

**16 Visualising semigroups and elements** **302**
16.1 `dot` pictures . . . 302
16.2 `tex` output . . . 304
16.3 `tikz` pictures . . . 305

**17 IO** **308**
17.1 Reading and writing elements to a file . . . . . 308
17.2 Reading and writing multiplication tables to a file . . . . . 309

**18 Translations** **311**
18.1 Methods for translations . . . . . 312

**References** **320**

**Index** **321**

# Chapter 1

# The Semigroups package

## 1.1 Introduction

This is the manual for the Semigroups package for GAP version 5.7.1. Semigroups 5.7.1 is a distant descendant of the Monoid package for GAP 3 by Goetz Pfeiffer, Steve A. Linton, Edmund F. Robertson, and Nik Ruskuc.

From Version 3.0.0, Semigroups includes a copy of the libsemigroups C++ library which contains implementations of the Froidure-Pin, Todd-Coxeter, and Knuth-Bendix algorithms (among others) that Semigroups utilises.

If you find a bug or an issue with the package, please visit the issue tracker.

## 1.2 Overview

This manual is organised as follows:

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

**Part II: semigroups and monoids defined by generating sets**
functions and operations for creating semigroups and monoids defined by generating sets (of the type described in Part I) are described in Chapter 6.

**Part III: standard examples and constructions**
standard examples of semigroups, such as `FullBooleanMatMonoid` (7.6.1) or `UniformBlockBijectionMonoid` (7.3.8), are described in Chapter 7, and standard constructions, such as `DirectProduct` (8.1.1) are given in Chapter 8.

**Part IV: the structure of a semigroup or monoid**
the functionality for determining various structural properties of a given semigroup or monoid are described in Chapters 9, 10, 11, and 12.

**Part V: congruences, quotients, and homomorphisms**
methods for creating and manipulating congruences and homomorphisms are described by Chapters 13 and 14.

**Part VI: finitely presented semigroups and monoids**
methods for finitely presented semigroups and monoids, in particular, for Tietze transformations can be found in Chapters 15.

**Part VII: utilities and helper functions**
functions for reading and writing semigroups and their elements, and for visualising semigroups, and some of their elements, can be found in Chapters 16 and 17.

# Chapter 2

# Installing Semigroups

## 2.1 For those in a hurry

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

It is assumed that you have a working copy of GAP with version number 4.12.1 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 `https://www.gap-system.org`.

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

- ensure that the datastructures package version 0.3.0 or higher is available. datastructures must be compiled before Semigroups can be loaded.
- ensure that the Digraphs package version 1.6.2 or higher is available. Digraphs must be compiled before Semigroups can be loaded.
- ensure that the genss package version 1.6.5 or higher is available.
- ensure that the images package version 1.3.1 or higher is available.
- ensure that the IO package version 4.5.1 or higher is available. IO must be compiled before Semigroups can be loaded.
- ensure that the orb package version 4.8.2 or higher is available. orb and Semigroups both perform better if orb is compiled.
- download the package archive `semigroups-5.7.1.tar.gz` from the Semigroups package webpage.
- unzip and untar the file, this should create a directory called `semigroups-5.7.1`.
- locate your GAP directory, the one containing the directories `lib`, `doc` and so on. Move the directory `semigroups-5.7.1` into the `pkg` subdirectory of your GAP directory.
- from version 3.0.0, it is necessary to compile the Semigroups package. Semigroups uses the libsemigroups C++ library, which requires a compiler implementing the C++17 standard. Inside the `pkg/semigroups-5.7.1` directory, in your terminal type

```
./configure && make
```

Further information about this step can be found in Section 2.2.

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

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

If you want to check that the package is working correctly, you should run some of the tests described in Section 2.4.

## 2.2 Compiling the kernel module

As of version 3.0.0, the Semigroups package has a kernel module written in C++ and this must be compiled. The kernel module contains the interface to the C++ library libsemigroups . It is not possible to use the Semigroups package without compiling it.

To compile the kernel component inside the `pkg/semigroups-5.7.1` directory, type

```
./configure && make
```

If you are using GCC to compile Semigroups , then version 5.0 or higher is required. Trying to compile Semigroups with an earlier version of GCC will result in an error at compile time. Semigroups only supports GCC version 5.0 or higher, and clang version 5.0 or higher.

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

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

## 2.3 Rebuilding the documentation

The Semigroups package comes complete with pdf, html, and text versions of the documentation. However, you might find it necessary, at some point, to rebuild the documentation. To rebuild the documentation the GAPDoc and AutoDoc packages are required. To rebuild the documentation type:

```
gap makedoc.g
```

when you're inside the `pkg/semigroups-5.7.1` directory.

## 2.4 Testing your installation

In this section we describe how to test that Semigroups is working as intended. To quickly test that Semigroups is installed correctly use `SemigroupsTestInstall` (2.4.1). For more extensive tests use `SemigroupsTestStandard` (2.4.2). Finally, for lengthy benchmarking tests use `SemigroupsTestExtreme` (2.4.3).

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

### 2.4.1 SemigroupsTestInstall

▷ `SemigroupsTestInstall()` (function)

**Returns:** `true` or `false`.

This function should be called with no argument to test that your installation of Semigroups is working correctly. These tests should take no more than a few seconds to complete. To more comprehensively test that Semigroups is installed correctly use `SemigroupsTestStandard` (2.4.2).

### 2.4.2 SemigroupsTestStandard

▷ `SemigroupsTestStandard()` (function)

**Returns:** `true` or `false`.

This function should be called with no argument to comprehensively test that Semigroups is working correctly. The value `true` is returned if the tests all pass, and `false` is returned otherwise. These tests should take no more than a few minutes to complete. To quickly test that Semigroups is installed correctly use `SemigroupsTestInstall` (2.4.1).

Each test file is run twice, once when the methods for `IsActingSemigroup` (6.1.2) are enabled and once when they are disabled.

### 2.4.3 SemigroupsTestExtreme

▷ `SemigroupsTestExtreme()` (function)

**Returns:** `true` or `false`.

This function should be called with no argument to run some long-running tests, which could be used to benchmark Semigroups or test your hardware. The value `true` is returned if the tests all pass, and `false` is returned otherwise. These tests should take no more than around half an hour to complete. To quickly test that Semigroups is installed correctly use `SemigroupsTestInstall` (2.4.1), or to test all aspects of the package use `SemigroupsTestStandard` (2.4.2).

Each test file is run twice, once when the methods for semigroups satisfying `IsActingSemigroup` (6.1.2) are enabled and once when they are disabled.

### 2.4.4 SemigroupsTestAll

▷ `SemigroupsTestAll()` (function)

**Returns:** `true` or `false`.

This function should be called with no argument to compile the Semigroups package's documentation, run the standard suite of tests, and run all the examples from the documentation to ensure that

their output is correct. The value returned is `true` if all the tests succeed, and `false` otherwise. The whole process should take no more than a few minutes.

See `SemigroupsTestStandard` (2.4.2).

## 2.5 More information during a computation

### 2.5.1 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. To increase the amount of information displayed during a computation increase the info level to `2`, `3`, or `4`. To stop all info messages from being displayed, set the info level to `0`. See also (**Reference: Info Functions**) and `SetInfoLevel` (**Reference: InfoLevel**).

# Chapter 3

# Bipartitions and blocks

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

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

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

Equivalently, the product $x\ y$ is defined to be the bipartition where $i\ , j \in \mathbf{n} \cup$-$\mathbf{n}$ (we assume without loss of generality that $i \geq j$ ) belong to the same block of $x\ y$ if either:

- $i = j$ ,
- $i\ ,\ j \in \mathbf{n}$ and $(i\ , j\ ) \in x$ , or
- $i\ ,\ j \in$ -$\mathbf{n}$ and $(i\ , j\ ) \in y$ ;

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

- $r = 2s - 1$ for some $s \geq 1$ , $i \in \mathbf{n}$ , $j \in$ -$\mathbf{n}$ and

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

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

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

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

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

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

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

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

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

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

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

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

## 3.1 The family and categories of bipartitions

### 3.1.1 IsBipartition

▷ `IsBipartition(obj)` (category)

**Returns:** `true` or `false`.

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

### 3.1.2 IsBipartitionCollection

▷ `IsBipartitionCollection(obj)` (category)

▷ `IsBipartitionCollColl(obj)` (category)

**Returns:** `true` or `false`.

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

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

## 3.2 Creating bipartitions

There are several ways of creating bipartitions in GAP, which are described in this section. The maximum degree of a bipartition is set as $2^29 - 1$. In reality, it is unlikely to be possible to create bipartitions of degrees as small as $2^24$ because they require too much memory.

### 3.2.1 Bipartition

▷ `Bipartition(`*`blocks`*`)` (function)

**Returns:** A bipartition.

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

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

Example

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

### 3.2.2 BipartitionByIntRep

▷ `BipartitionByIntRep(`*`list`*`)` (operation)

**Returns:** A bipartition.

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

For example, the internal representation of the bipartition with blocks

Example

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

has internal representation

Example

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

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

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

See also `IntRepOfBipartition` (3.5.4).

Example

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

### 3.2.3 IdentityBipartition

▷ `IdentityBipartition(`*`n`*`)` (operation)

**Returns:** The identity bipartition.

Returns the identity bipartition with degree *`n`*.

Example

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

### 3.2.4 LeftOne (for a bipartition)

▷ `LeftOne(x)` (attribute)

▷ `LeftProjection(x)` (attribute)

**Returns:** A bipartition.

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

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

Example

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

### 3.2.5 RightOne (for a bipartition)

▷ `RightOne(x)` (attribute)

▷ `RightProjection(x)` (attribute)

**Returns:** A bipartition.

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

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

Example

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

### 3.2.6 StarOp (for a bipartition)

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

**Returns:** A bipartition.

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

Example

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

### 3.2.7 TensorBipartitions (for a pair of bipartitions)

▷ TensorBipartitions(x, y) (operation)
▷ TensorBipartitions(coll) (operation)

**Returns:** A bipartition.

This operation returns the tensor product of the bipartitions x and y, or of all of the bipartitions in coll. Roughly speaking, the *tensor product* of bipartitions x and y is obtained by writing the bipartitions next to each other and reindexing the second one y. More precisely, if x has degree m and y has degree n, then the tensor product is the partition of [-m - n .. -1, 1 .. m + n] given by:

- the blocks containing values in [-m .. -1, 1 .. m] are precisely the blocks of x;
- the blocks containing values in [-m - n .. -m - 1, m + 1 ... m + n] are obtained from the blocks of y by adding m to the positive, and subtracting m from the negative values.

If the argument is a list of bipartitions coll, then the tensor product of the bipartitions in the list (from left to right) is returned. An error is given if coll is empty.

See also: IrreducibleComponentsOfBipartition (3.5.7) and IsIrreducibleBipartition (3.5.20).

Example

```
gap> TensorBipartitions(
> Bipartition([[1, 2, -2], [-1]]),
> Bipartition([[1, 2, 3, -2], [-1], [-3]]));
<bipartition: [ 1, 2, -2 ], [ 3, 4, 5, -4 ], [ -1 ], [ -3 ], [ -5 ]>
```

### 3.2.8 RandomBipartition

▷ `RandomBipartition(`*`[rs, ]n`*`)` (operation)
▷ `RandomBlockBijection(`*`[rs, ]n`*`)` (operation)

**Returns:** A bipartition.

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

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

Note that neither of these functions has a uniform distribution.

Example

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

## 3.3 Changing the representation of a bipartition

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

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

### 3.3.1 AsBipartition

▷ `AsBipartition(`*`x[, n]`*`)` (operation)

**Returns:** A bipartition.

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

There are several possible arguments for `AsBipartition`:

**permutations**

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

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

**transformations**

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

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

**partial permutations**

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

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

**bipartitions**

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

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

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

Example

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

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

### 3.3.2 AsBlockBijection

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

**Returns:** A block bijection.

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

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

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

When the argument *x* is a partial perm bipartition (see `IsPartialPermBipartition` (3.5.16)) then this operation returns `AsBlockBijection(AsPartialPerm(`*x*`)[, `*n*`])`.

Example

```
gap> x := PartialPerm([1, 2, 3, 6, 7, 10], [9, 5, 6, 1, 7, 8]);
[2,5][3,6,1,9][10,8](7)
gap> AsBipartition(x, 11);
<bipartition: [ 1, -9 ], [ 2, -5 ], [ 3, -6 ], [ 4 ], [ 5 ],
 [ 6, -1 ], [ 7, -7 ], [ 8 ], [ 9 ], [ 10, -8 ], [ 11 ], [ -2 ],
 [ -3 ], [ -4 ], [ -10 ], [ -11 ]>
gap> AsBlockBijection(x, 10);
Error, the 2nd argument (a pos. int.) is less than or equal to the max\
imum of the degree and codegree of the 1st argument (a partial perm)
gap> AsBlockBijection(x, 11);
<block bijection: [ 1, -9 ], [ 2, -5 ], [ 3, -6 ],
 [ 4, 5, 8, 9, 11, -2, -3, -4, -10, -11 ], [ 6, -1 ], [ 7, -7 ],
 [ 10, -8 ]>
gap> x := Bipartition([[1, -3], [2], [3, -2], [-1]]);;
gap> IsPartialPermBipartition(x);
true
```

```
gap> AsBlockBijection(x);
<block bijection: [ 1, -3 ], [ 2, 4, -1, -4 ], [ 3, -2 ]>
```

### 3.3.3 AsTransformation (for a bipartition)

▷ AsTransformation(x) (attribute)

**Returns:** A transformation.

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

See IsTransBipartition (3.5.13).

Example

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

### 3.3.4 AsPartialPerm (for a bipartition)

▷ AsPartialPerm(x) (operation)

**Returns:** A partial perm.

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

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

See IsPartialPermBipartition (3.5.16).

Example

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

### 3.3.5 AsPermutation (for a bipartition)

▷ `AsPermutation(x)` (attribute)

**Returns:** A permutation.

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

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

See `IsPermBipartition` (3.5.15).

Example

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

## 3.4 Operators for bipartitions

`f * g`

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

`f < g`

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

`f = g`

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

### 3.4.1 PartialPermLeqBipartition

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

**Returns:** `true` or `false`.

If `x` and `y` are partial perm bipartitions, i.e. they satisfy `IsPartialPermBipartition` (3.5.16), then this function returns `AsPartialPerm(x) < AsPartialPerm(y)`.

### 3.4.2 NaturalLeqPartialPermBipartition

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

**Returns:** `true` or `false`.

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

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

### 3.4.3 NaturalLeqBlockBijection

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

**Returns:** true or false.

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

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

Example

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

### 3.4.4 PermLeftQuoBipartition

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

**Returns:** A permutation.

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

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

Example

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

## 3.5 Attributes for bipartitons

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

### 3.5.1 DegreeOfBipartition

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

**Returns:** A positive integer.

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

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

Example

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

### 3.5.2 RankOfBipartition

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

**Returns:** The rank of a bipartition.

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

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

Example

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

### 3.5.3 ExtRepOfObj (for a bipartition)

▷ ExtRepOfObj(x) (operation)

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

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

Example

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

### 3.5.4 IntRepOfBipartition

▷ IntRepOfBipartition(x) (attribute)

**Returns:** A list of positive integers.

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

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

See also BipartitionByIntRep (3.2.2).

Example

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

### 3.5.5 RightBlocks

▷ RightBlocks(x) (attribute)

**Returns:** The right blocks of a bipartition.

RightBlocks returns the right blocks of the bipartition x.

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

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

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

Example

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

### 3.5.6 LeftBlocks

▷ LeftBlocks(x) (attribute)

**Returns:** The left blocks of a bipartition.

LeftBlocks returns the left blocks of the bipartition x.

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

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

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

Example

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

### 3.5.7 IrreducibleComponentsOfBipartition

▷ `IrreducibleComponentsOfBipartition(x)` (operation)

**Returns:** A list of bipartitions.

The irreducible components of the bipartition $x$ are the maximal degree bipartitions `y1, ..., ym` such that `TensorBipartitions(y1, ... ym)` equals $x$. See also: `TensorBipartitions` (3.2.7) and `IsIrreducibleBipartition` (3.5.20).

Example

```
gap> x := TensorBipartitions(
> Bipartition([[1, 2, -2], [-1]]),
> Bipartition([[1, 2, 3, -2], [-1], [-3]]));
<bipartition: [ 1, 2, -2 ], [ 3, 4, 5, -4 ], [ -1 ], [ -3 ], [ -5 ]>
gap> coll := IrreducibleComponentsOfBipartition(x);
[ <bipartition: [ 1, 2, -2 ], [ -1 ]>,
<bipartition: [ 1, 2, 3, -2 ], [ -1 ], [ -3 ]> ]
gap> TensorBipartitions(coll) = x;
true
```

### 3.5.8 NrLeftBlocks

▷ `NrLeftBlocks(x)` (attribute)

**Returns:** A non-negative integer.

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

Example

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

### 3.5.9 NrRightBlocks

▷ `NrRightBlocks(x)` (attribute)

**Returns:** A non-negative integer.

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

Example

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

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

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

### 3.5.11 DomainOfBipartition

▷ DomainOfBipartition(x) (attribute)

**Returns:** A list of positive integers.

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

Example

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

### 3.5.12 CodomainOfBipartition

▷ CodomainOfBipartition(x) (attribute)

**Returns:** A list of positive integers.

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

Example

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

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

### 3.5.13 IsTransBipartition

▷ `IsTransBipartition(x)` (property)

**Returns:** `true` or `false`.

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

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

Example

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

### 3.5.14 IsDualTransBipartition

▷ `IsDualTransBipartition(x)` (property)

**Returns:** `true` or `false`.

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

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

Example

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

### 3.5.15 IsPermBipartition

▷ IsPermBipartition(x) (property)

**Returns:** true or false.

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

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

Example

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

### 3.5.16 IsPartialPermBipartition

▷ IsPartialPermBipartition(x) (property)

**Returns:** true or false.

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

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

Example

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

### 3.5.17 IsBlockBijection

▷ IsBlockBijection(x) (property)

**Returns:** true or false.

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

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

Example

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

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

### 3.5.18 IsUniformBlockBijection

▷ `IsUniformBlockBijection(x)` (property)

**Returns:** `true` or `false`.

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

Example

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

### 3.5.19 CanonicalBlocks

▷ `CanonicalBlocks(blocks)` (attribute)

**Returns:** Blocks of a bipartition.

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

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

Example

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

### 3.5.20 IsIrreducibleBipartition

▷ `IsIrreducibleBipartition(x)` (operation)

**Returns:** `true` or `false`.

A bipartition is *irreducible* if it is not the tensor product of bipartitions of strictly smaller degree. This function returns `true` if *x* is irreducible and `false` if it is reducible. See also: `TensorBipartitions` (3.2.7) and `IrreducibleComponentsOfBipartition` (3.5.7).

Example

```
gap> IsIrreducibleBipartition(TensorBipartitions(
> Bipartition([[1, 2, -2], [-1]]),
> Bipartition([[1, 2, 3, -2], [-1], [-3]])));
false
```

```
gap> IsIrreducibleBipartition(Bipartition([[1, 2, 3, -2], [-1], [-3]]));
true
```

## 3.6 Creating blocks and their attributes

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

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

### 3.6.1 IsBlocks

▷ `IsBlocks(`*obj*`)` (category)

**Returns:** `true` or `false`.

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

### 3.6.2 BLOCKS_NC

▷ `BLOCKS_NC(`*classes*`)` (function)

**Returns:** A blocks object.

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.

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

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

Example

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

### 3.6.3 ExtRepOfObj (for a blocks)

▷ `ExtRepOfObj(`*blocks*`)` (operation)

**Returns:** A list of integers.

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

Example

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

### 3.6.4 RankOfBlocks

▷ RankOfBlocks(*blocks*) (attribute)
▷ NrTransverseBlocks(*blocks*) (attribute)

**Returns:** A non-negative integer.

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

NrTransverseBlocks is a synonym of RankOfBlocks in this context.

Example

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

### 3.6.5 DegreeOfBlocks

▷ DegreeOfBlocks(*blocks*) (attribute)

**Returns:** A non-negative integer.

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

Example

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

### 3.6.6 ProjectionFromBlocks

▷ ProjectionFromBlocks(*blocks*) (attribute)

**Returns:** A bipartition.

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

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

Example

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

## 3.7 Actions on blocks

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

### 3.7.1 OnRightBlocks

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

**Returns:** The blocks of a bipartition.

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

Example

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

### 3.7.2 OnLeftBlocks

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

**Returns:** The blocks of a bipartition.

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

Example

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

## 3.8 Semigroups of bipartitions

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

It is possible to create inverse semigroups and monoids of bipartitions using `InverseSemigroup` (**Reference: InverseSemigroup**) and `InverseMonoid` (**Reference: InverseMonoid**) when the argument is a collection of block bijections or partial perm bipartions; see `IsBlockBijection` (3.5.17) and `IsPartialPermBipartition` (3.5.16). Note that every bipartition semigroup in Semigroups is finite.

### 3.8.1 IsBipartitionSemigroup

▷ `IsBipartitionSemigroup(`*S*`)` (filter)
▷ `IsBipartitionMonoid(`*S*`)` (filter)

**Returns:** `true` or `false`.

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

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

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

Example

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

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

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

### 3.8.2 IsBlockBijectionSemigroup

▷ `IsBlockBijectionSemigroup(`*S*`)` (property)
▷ `IsBlockBijectionMonoid(`*S*`)` (filter)

**Returns:** `true` or `false`.

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

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

See `IsBlockBijection` (3.5.17).

### 3.8.3 IsPartialPermBipartitionSemigroup

▷ `IsPartialPermBipartitionSemigroup(S)` (property)
▷ `IsPartialPermBipartitionMonoid(S)` (filter)

**Returns:** `true` or `false`.

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

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

See `IsPartialPermBipartition` (3.5.16).

### 3.8.4 IsPermBipartitionGroup

▷ `IsPermBipartitionGroup(S)` (property)

**Returns:** `true` or `false`.

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

See `IsPermBipartition` (3.5.15).

### 3.8.5 DegreeOfBipartitionSemigroup

▷ `DegreeOfBipartitionSemigroup(S)` (attribute)

**Returns:** A non-negative integer.

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

Example

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

# Chapter 4

# Partitioned binary relations (PBRs)

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

PBRs were introduced in the paper [MM13] as, roughly speaking, the maximum generalization of bipartitions and related objects. Although, mathematically, bipartitions are a special type of PBR, in Semigroups bipartitions and PBRs are currently distinct types of objects. It is possible to change the representation from bipartition to PBR, and from PBR to bipartition, when appropriate; see Section 4.3 for more details. The reason for this distinct is largely historical, bipartition appeared in the literature, and in the Semigroups package, before PBRs.

## 4.1 The family and categories of PBRs

### 4.1.1 IsPBR

▷ `IsPBR(`*obj*`)` (category)

**Returns:** `true` or `false`.

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

### 4.1.2 IsPBRCollection

▷ `IsPBRCollection(`*obj*`)` (category)

▷ `IsPBRCollColl(`*obj*`)` (category)

**Returns:** `true` or `false`.

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

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

## 4.2 Creating PBRs

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

### 4.2.1 PBR

▷ PBR(*left*, *right*) (operation)

**Returns:** A PBR.

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

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

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

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

Example

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

### 4.2.2 RandomPBR

▷ RandomPBR(*n*[, *p*]) (operation)

**Returns:** A PBR.

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

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

Example

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

### 4.2.3 EmptyPBR

▷ EmptyPBR(*n*) (operation)

**Returns:** A PBR.

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

Example

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

### 4.2.4 IdentityPBR

▷ IdentityPBR(*n*) (operation)

**Returns:** A PBR.

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

Example

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

### 4.2.5 UniversalPBR

▷ UniversalPBR(*n*) (operation)

**Returns:** A PBR.

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

Example

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

## 4.3 Changing the representation of a PBR

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

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

### 4.3.1 AsPBR

▷ AsPBR(*x*[, *n*]) (operation)

**Returns:** A PBR.

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

There are several possible arguments for AsPBR:

**bipartitions**

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

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

**boolean matrices**

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

**transformations, partial perms, permutations**

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

Example

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

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

### 4.3.2 AsTransformation (for a PBR)

▷ `AsTransformation(x)` (attribute)

**Returns:** A transformation.

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

Example

```
gap> x := PBR([[-3], [-3], [-2]], [[], [3], [1, 2]]);;
gap> IsTransformationPBR(x);
true
gap> AsTransformation(x);
Transformation( [ 3, 3, 2 ] )
gap> x := PBR([[1], [1, 2]], [[-2, -1], [-2, -1]]);;
gap> AsTransformation(x);
Error, the argument (a pbr) does not define a transformation
```

### 4.3.3 AsPartialPerm (for a PBR)

▷ `AsPartialPerm(x)` (operation)

**Returns:** A partial perm.

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

Example

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

### 4.3.4 AsPermutation (for a PBR)

▷ `AsPermutation(x)` (attribute)

**Returns:** A permutation.

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

Example

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

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

## 4.4 Operators for PBRs

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

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

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

## 4.5 Attributes for PBRs

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

### 4.5.1 StarOp (for a PBR)

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

**Returns:** A PBR.

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

Example

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

### 4.5.2 DegreeOfPBR

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

**Returns:** A positive integer.

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

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

Example

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

```
3
gap> S := FullPBRMonoid(2);
<pbr monoid of degree 2 with 10 generators>
gap> DegreeOfPBRCollection(S);
2
```

### 4.5.3 ExtRepOfObj (for a PBR)

▷ `ExtRepOfObj(x)` (operation)

**Returns:** A pair of lists of lists of integers.

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

Example

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

### 4.5.4 PBRNumber

▷ `PBRNumber(m, n)` (operation)

▷ `NumberPBR(mat)` (operation)

**Returns:** A PBR, or a positive integer.

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

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

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

Example

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

### 4.5.5 IsEmptyPBR

▷ `IsEmptyPBR(x)` (property)

**Returns:** `true` or `false`.

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

Example

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

```
gap> IsEmptyPBR(x);
false
```

### 4.5.6 IsIdentityPBR

▷ `IsIdentityPBR(x)` (property)

**Returns:** `true` or `false`.

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

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

Example

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

### 4.5.7 IsUniversalPBR

▷ `IsUniversalPBR(x)` (property)

**Returns:** `true` or `false`.

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

Example

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

### 4.5.8 IsBipartitionPBR

▷ `IsBipartitionPBR(x)` (property)

▷ `IsBlockBijectionPBR(x)` (property)

**Returns:** `true` or `false`.

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

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

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

Example

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

### 4.5.9 IsTransformationPBR

▷ IsTransformationPBR(x) (property)

**Returns:** true or false.

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

A PBR $x$ defines a transformation if and only if it satisfies IsBipartitionPBR (4.5.8) and when it is considered as a bipartition it satisfies IsTransBipartition (3.5.13).

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

Example

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

### 4.5.10 IsDualTransformationPBR

▷ IsDualTransformationPBR(x) (property)

**Returns:** true or false.

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

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

Example

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

### 4.5.11 IsPartialPermPBR

▷ IsPartialPermPBR(*x*) (property)

**Returns:** true or false.

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

A PBR *x* defines a partial perm if and only if it satisfies IsBipartitionPBR (4.5.8) and when it is considered as a bipartition it satisfies IsPartialPermBipartition (3.5.16).

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

Example

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

### 4.5.12 IsPermPBR

▷ IsPermPBR(*x*) (property)

**Returns:** true or false.

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

A PBR *x* defines a permutation if and only if it satisfies IsBipartitionPBR (4.5.8) and when it is considered as a bipartition it satisfies IsPermBipartition (3.5.15).

With this definition, AsPBR (4.3.1) and AsPermutation (4.3.4) define mutually inverse isomorphisms from the symmetric group of degree n to the subgroup of the full PBR monoid of degree n

consisting of all the elements satisfying IsPermPBR (i.e. the GroupOfUnits (11.9.1) of the full PBR monoid of degree n).

Example

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

## 4.6 Semigroups of PBRs

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

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

Note that every PBR semigroup in Semigroups is finite.

### 4.6.1 IsPBRSemigroup

▷ IsPBRSemigroup(*S*) (filter)

▷ IsPBRMonoid(*S*) (filter)

**Returns:** true or false.

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

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

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

Example

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

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

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

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

### 4.6.2 DegreeOfPBRSemigroup

▷ `DegreeOfPBRSemigroup(`*S*`)` (attribute)

**Returns:** A non-negative integer.

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

Example

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

# Chapter 5

# Matrices over semirings

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

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

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

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

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

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

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

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

**the integers**
the usual ring of integers;

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

With the exception of matrices of finite fields, semigroups of matrices in Semigroups are of the second type described in Section 6.1. In other words, a version of the Froidure-Pin Algorithm [FP97] is used to compute semigroups of these types, i.e it is possible that all of the elements of such a semigroup are enumerated and stored in the memory of your computer.

## 5.1 Creating matrices over semirings

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

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

### 5.1.1 IsMatrixOverSemiring

▷ `IsMatrixOverSemiring(`*obj*`)` (category)

**Returns:** `true` or `false`.

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

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

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

### 5.1.2 IsMatrixOverSemiringCollection

▷ `IsMatrixOverSemiringCollection(`*obj*`)` (category)

▷ `IsMatrixOverSemiringCollColl(`*obj*`)` (category)

**Returns:** `true` or `false`.

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

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

### 5.1.3 DimensionOfMatrixOverSemiring

▷ `DimensionOfMatrixOverSemiring(`*mat*`)` (attribute)

**Returns:** A positive integer.

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

Example

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

### 5.1.4 DimensionOfMatrixOverSemiringCollection

▷ DimensionOfMatrixOverSemiringCollection(*coll*) (attribute)

**Returns:** A positive integer.

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

Example

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

### 5.1.5 Matrix (for a filter and a matrix)

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

▷ Matrix(*semiring, mat*) (operation)

**Returns:** A matrix over semiring.

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

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

*filt*

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

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

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

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

*threshold*

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

*period*

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

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

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

Example

```
gap> Matrix(IsBooleanMat, [[1, 0, 0, 0],
>                          [0, 0, 0, 0],
>                          [1, 1, 1, 1],
>                          [1, 0, 1, 1]]);
Matrix(IsBooleanMat, [[1, 0, 0, 0], [0, 0, 0, 0], [1, 1, 1, 1],
  [1, 0, 1, 1]])
gap> Matrix(IsMaxPlusMatrix, [[4, 0, -2],
>                            [1, -3, 0],
>                            [5, -1, -4]]);
Matrix(IsMaxPlusMatrix, [[4, 0, -2], [1, -3, 0], [5, -1, -4]])
gap> Matrix(IsMinPlusMatrix, [[-1, infinity],
>                            [1, -1]]);
Matrix(IsMinPlusMatrix, [[-1, infinity], [1, -1]])
gap> Matrix(IsTropicalMaxPlusMatrix, [[3, 2, 4],
>                                     [3, 1, 1],
>                                     [-infinity, 1, 1]],
>           9);
Matrix(IsTropicalMaxPlusMatrix, [[3, 2, 4], [3, 1, 1],
  [-infinity, 1, 1]], 9)
gap> Matrix(IsTropicalMinPlusMatrix, [[1, 1, 1],
>                                     [0, 3, 0],
>                                     [1, 1, 3]],
>           9);
Matrix(IsTropicalMinPlusMatrix, [[1, 1, 1], [0, 3, 0], [1, 1, 3]], 9)
gap> Matrix(IsNTPMatrix, [[0, 0, 0],
>                         [2, 0, 1],
>                         [2, 2, 2]],
>           2, 1);
Matrix(IsNTPMatrix, [[0, 0, 0], [2, 0, 1], [2, 2, 2]], 2, 1)
gap> Matrix(Integers, [[-1, -2, 0],
>                      [0, 3, -1],
>                      [1, 0, -3]]);
<3x3 plist matrix over Integers>
gap> Matrix(Integers, [[-1, -2, 0],
>                      [0, 3, -1],
>                      [1, 0, -3]]);
<3x3 plist matrix over Integers>
```

### 5.1.6 AsMatrix (for a filter and a matrix)

▷ AsMatrix(*filt*, *mat*) (operation)

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

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

**Returns:** A matrix.

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

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

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

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

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

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

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

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

Example

```
gap> mat := Matrix(IsTropicalMinPlusMatrix, [[0, 1, 3],
>                                            [1, 1, 6],
>                                            [0, 4, 2]], 10);;
gap> AsMatrix(IsMinPlusMatrix, mat);
Matrix(IsMinPlusMatrix, [[0, 1, 3], [1, 1, 6], [0, 4, 2]])
gap> mat := Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
>                                            [0, 1, 3],
>                                            [4, 1, 0]], 10);;
gap> AsMatrix(IsMaxPlusMatrix, mat);
Matrix(IsMaxPlusMatrix, [[-infinity, -infinity, 3], [0, 1, 3],
  [4, 1, 0]])
gap> mat := Matrix(IsNTPMatrix, [[1, 2, 2],
>                                [0, 2, 0],
>                                [1, 3, 0]], 4, 5);;
gap> Matrix(Integers, mat);
<3x3 plist matrix over Integers>
gap> mat := Matrix(IsMinPlusMatrix, [[0, 1, 3], [1, 1, 6], [0, 4, 2]]);;
gap> mat := AsMatrix(IsTropicalMinPlusMatrix, mat, 2);
Matrix(IsTropicalMinPlusMatrix, [[0, 1, 2], [1, 1, 2], [0, 2, 2]], 2)
gap> mat := AsMatrix(IsTropicalMinPlusMatrix, mat, 1);
Matrix(IsTropicalMinPlusMatrix, [[0, 1, 1], [1, 1, 1], [0, 1, 1]], 1)
gap> mat := Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
>                                            [0, 1, 3],
>                                            [4, 1, 0]], 10);;
gap> AsMatrix(IsTropicalMaxPlusMatrix, mat, 4);
Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
  [0, 1, 3], [4, 1, 0]], 4)
gap> mat := Matrix(IsMaxPlusMatrix, [[-infinity, -infinity, 3],
>                                    [0, 1, 3],
>                                    [4, 1, 0]]);;
gap> AsMatrix(IsTropicalMaxPlusMatrix, mat, 10);
Matrix(IsTropicalMaxPlusMatrix, [[-infinity, -infinity, 3],
  [0, 1, 3], [4, 1, 0]], 10)
```

```
gap> mat := Matrix(IsNTPMatrix, [[0, 1, 0],
>                                [1, 3, 1],
>                                [1, 0, 1]], 10, 10);;
gap> mat := AsMatrix(IsNTPMatrix, mat, 5, 6);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 3, 1], [1, 0, 1]], 5, 6)
gap> mat := AsMatrix(IsNTPMatrix, mat, 2, 6);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 3, 1], [1, 0, 1]], 2, 6)
gap> mat := AsMatrix(IsNTPMatrix, mat, 2, 1);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 2, 1], [1, 0, 1]], 2, 1)
gap> mat := Matrix(Integers, mat);
<3x3 plist matrix over Integers>
gap> AsMatrix(IsNTPMatrix, mat, 1, 2);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 2, 1], [1, 0, 1]], 1, 2)
```

### 5.1.7 RandomMatrix (for a filter and a matrix)

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

▷ `RandomMatrix(`*semiring, dim*`)` (function)

**Returns:** A matrix over semiring.

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

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

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

*dim* This must be a positive integer.

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

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

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

Example

```
gap> RandomMatrix(IsBooleanMat, 3);
Matrix(IsBooleanMat, [[1, 0, 0], [1, 0, 1], [1, 0, 1]])
gap> RandomMatrix(IsMaxPlusMatrix, 2);
Matrix(IsMaxPlusMatrix, [[1, -infinity], [1, 0]])
gap> RandomMatrix(IsMinPlusMatrix, 3);
Matrix(IsMinPlusMatrix, [[infinity, 2, infinity], [4, 0, -2], [1, -3, 0]])
gap> RandomMatrix(IsTropicalMaxPlusMatrix, 3, 5);
```

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

### 5.1.8 Matrix filters

▷ IsBooleanMat(*obj*) (category)
▷ IsMaxPlusMatrix(*obj*) (category)
▷ IsMinPlusMatrix(*obj*) (category)
▷ IsTropicalMatrix(*obj*) (category)
▷ IsTropicalMaxPlusMatrix(*obj*) (category)
▷ IsTropicalMinPlusMatrix(*obj*) (category)
▷ IsNTPMatrix(*obj*) (category)

**Returns:** `true` or `false`.

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

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

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

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

### 5.1.9 Matrix collection filters

▷ IsBooleanMatCollection(*obj*) (category)
▷ IsBooleanMatCollColl(*obj*) (category)
▷ IsMatrixOverFiniteFieldCollection(*obj*) (category)
▷ IsMatrixOverFiniteFieldCollColl(*obj*) (category)
▷ IsMaxPlusMatrixCollection(*obj*) (category)
▷ IsMaxPlusMatrixCollColl(*obj*) (category)
▷ IsMinPlusMatrixCollection(*obj*) (category)
▷ IsMinPlusMatrixCollColl(*obj*) (category)
▷ IsTropicalMatrixCollection(*obj*) (category)

▷ IsTropicalMaxPlusMatrixCollection(*obj*) (category)
▷ IsTropicalMaxPlusMatrixCollColl(*obj*) (category)
▷ IsTropicalMinPlusMatrixCollection(*obj*) (category)
▷ IsTropicalMinPlusMatrixCollColl(*obj*) (category)
▷ IsNTPMatrixCollection(*obj*) (category)
▷ IsNTPMatrixCollColl(*obj*) (category)

**Returns:** true or false.

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

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

### 5.1.10 AsList

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

**Returns:** A list of lists.

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

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

Example

```
gap> mat := Matrix(IsNTPMatrix,
>                  [[0, 1, 0], [1, 3, 1], [1, 0, 1]], 5, 6);
Matrix(IsNTPMatrix, [[0, 1, 0], [1, 3, 1], [1, 0, 1]], 5, 6)
gap> list := AsList(mat);
[ [ 0, 1, 0 ], [ 1, 3, 1 ], [ 1, 0, 1 ] ]
gap> IsMutable(list);
false
gap> IsMutable(list[1]);
false
gap> list := AsMutableList(mat);
[ [ 0, 1, 0 ], [ 1, 3, 1 ], [ 1, 0, 1 ] ]
gap> IsMutable(list);
true
gap> IsMutable(list[1]);
true
gap> mat = Matrix(IsNTPMatrix, AsList(mat), 5, 6);
true
```

### 5.1.11 ThresholdTropicalMatrix

▷ ThresholdTropicalMatrix(*mat*) (attribute)

**Returns:** A positive integer.

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

Example

```
gap> mat := Matrix(IsTropicalMaxPlusMatrix,
> [[0, 3, 0, 2],
```

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

### 5.1.12 ThresholdNTPMatrix

▷ `ThresholdNTPMatrix(`*mat*`)` (attribute)

▷ `PeriodNTPMatrix(`*mat*`)` (attribute)

**Returns:** A positive integer.

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

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

Example

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

## 5.2 Operators for matrices over semirings

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

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

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

## 5.3 Boolean matrices

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

- `NumberBooleanMat` (5.3.6)
- `Successors` (5.3.5)
- `IsRowTrimBooleanMat` (5.3.9), `IsColTrimBooleanMat` (5.3.9), and `IsTrimBooleanMat` (5.3.9),
- `CanonicalBooleanMat` (5.3.8)
- `IsSymmetricBooleanMat` (5.3.10)
- `IsAntiSymmetricBooleanMat` (5.3.13)
- `IsTransitiveBooleanMat` (5.3.12)
- `IsReflexiveBooleanMat` (5.3.11)
- `IsTotalBooleanMat` (5.3.14)
- `IsOntoBooleanMat` (5.3.14)
- `IsPartialOrderBooleanMat` (5.3.15)
- `IsEquivalenceBooleanMat` (5.3.16)

### 5.3.1 BooleanMat

▷ `BooleanMat(`*arg*`)` (function)

**Returns:** A boolean matrix.

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

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

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

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

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

Example

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

### 5.3.2 AsBooleanMat

▷ `AsBooleanMat(x[, n])` (operation)

**Returns:** A boolean matrix.

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

There are several possible arguments for `AsBooleanMat`:

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

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

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

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

**partial permutations**
If *x* is a partial permutation and *n* is a positive integer such that `i ^ x <= n` for all `i` in `[1 ..`

*n*], then AsBooleanMat returns the boolean matrix mat of dimension *n* such that mat[i][j] = true if and only if j = i ^ x.

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

**bipartitions**

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

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

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

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

Example

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

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

```
1 1 1 1 0 1
```

### 5.3.3 \in

▷ \in(*mat1*, *mat2*) (operation)

**Returns:** true or false.

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

Example

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

### 5.3.4 OnBlist

▷ OnBlist(*blist*, *mat*) (function)

**Returns:** A boolean list.

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

Example

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

### 5.3.5 Successors

▷ Successors(*mat*) (attribute)

**Returns:** A list of lists of positive integers.

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

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

Example

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

### 5.3.6 BooleanMatNumber

▷ BooleanMatNumber(*m*, *n*) (operation)
▷ NumberBooleanMat(*mat*) (operation)

**Returns:** A boolean matrix, or a positive integer.

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

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

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

Example

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

### 5.3.7 BlistNumber

▷ BlistNumber(*m*, *n*) (function)
▷ NumberBlist(*blist*) (function)

**Returns:** A boolean list, or a positive integer.

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

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

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

Example

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

```
[ false, false, false, false, false, false, false, false, false, true
 ]
```

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

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

**Returns:** A boolean matrix.

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

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

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

Example

```
gap> mat := BooleanMat([[1, 1, 1, 0, 0, 0],
>                       [0, 0, 0, 1, 0, 1],
>                       [1, 0, 0, 1, 0, 1],
>                       [0, 0, 0, 0, 0, 0],
>                       [0, 1, 1, 1, 1, 1],
>                       [0, 1, 1, 0, 1, 0]]);
Matrix(IsBooleanMat, [[1, 1, 1, 0, 0, 0], [0, 0, 0, 1, 0, 1],
  [1, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 1, 1],
  [0, 1, 1, 0, 1, 0]])
gap> CanonicalBooleanMat(mat);
Matrix(IsBooleanMat, [[0, 0, 0, 0, 0, 0], [1, 1, 0, 0, 0, 0],
  [0, 0, 1, 1, 1, 0], [1, 1, 0, 0, 1, 0], [0, 0, 1, 1, 0, 1],
  [1, 1, 1, 1, 0, 1]])
gap> Display(CanonicalBooleanMat(mat));
0 0 0 0 0 0
1 1 0 0 0 0
0 0 1 1 1 0
1 1 0 0 1 0
0 0 1 1 0 1
1 1 1 1 0 1
gap> Display(CanonicalBooleanMat(Group((1, 3)), mat));
0 1 1 0 0 1
0 0 1 0 0 1
1 1 0 1 0 0
0 0 0 0 0 0
1 0 1 1 1 1
1 0 0 1 1 0
gap> Display(CanonicalBooleanMat(Group((1, 3)), Group(()), mat));
1 1 1 0 0 0
```

```
0 0 0 1 0 1
0 1 0 1 0 1
0 0 0 0 0 0
1 0 1 1 1 1
1 0 1 0 1 0
```

### 5.3.9 IsRowTrimBooleanMat

▷ IsRowTrimBooleanMat(*mat*) (property)
▷ IsColTrimBooleanMat(*mat*) (property)
▷ IsTrimBooleanMat(*mat*) (property)

**Returns:** true or false.

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

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

Example

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

### 5.3.10 IsSymmetricBooleanMat

▷ IsSymmetricBooleanMat(*mat*) (property)

**Returns:** true or false.

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

Example

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

```
> [1, 0, 1, 1],
> [1, 1, 0, 1],
> [0, 1, 1, 1]]);
Matrix(IsBooleanMat, [[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 1],
  [0, 1, 1, 1]])
gap> IsSymmetricBooleanMat(mat);
true
```

### 5.3.11 IsReflexiveBooleanMat

▷ `IsReflexiveBooleanMat(`*mat*`)` (property)

**Returns:** `true` or `false`.

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

Example

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

### 5.3.12 IsTransitiveBooleanMat

▷ `IsTransitiveBooleanMat(`*mat*`)` (property)

**Returns:** `true` or `false`.

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

Example

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

```
Matrix(IsBooleanMat, [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1],
  [1, 1, 1, 1]])
gap> IsTransitiveBooleanMat(x);
true
```

### 5.3.13 IsAntiSymmetricBooleanMat

▷ IsAntiSymmetricBooleanMat(*mat*) (property)

**Returns:** true or false.

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

Example

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

### 5.3.14 IsTotalBooleanMat

▷ IsTotalBooleanMat(*mat*) (property)

▷ IsOntoBooleanMat(*mat*) (property)

**Returns:** true or false.

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

Example

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

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

### 5.3.15 IsPartialOrderBooleanMat

▷ `IsPartialOrderBooleanMat(`*mat*`)` (property)

**Returns:** `true` or `false`.

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

Example

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

### 5.3.16 IsEquivalenceBooleanMat

▷ `IsEquivalenceBooleanMat(`*mat*`)` (property)

**Returns:** `true` or `false`.

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

Example

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

### 5.3.17 IsTransformationBooleanMat

▷ `IsTransformationBooleanMat(`*mat*`)` (property)

**Returns:** `true` or `false`.

A boolean matrix is a TRANSFORMATION if every row contains precisely one `true` value.

Example

```
gap> Number(FullBooleanMatMonoid(3), IsTransformationBooleanMat);
27
```

## 5.4 Matrices over finite fields

In this section we describe some operations in Semigroups for matrices over finite fields that are required for such matrices to form semigroups satisfying `IsActingSemigroup` (6.1.2).

From v5.0.0, Semigroups uses the GAP library implementation of matrices over finite fields belonging to the category `IsMatrixObj` (**Reference: IsMatrixObj**) rather than the previous implementation in the Semigroups package. This means that from v5.0.0, matrices over a finite field no longer belong to the category `IsMatrixOverSemiring` (5.1.1).

The following methods are implemented in Semigroups for matrix objects over finite fields.

### 5.4.1 RowSpaceBasis (for a matrix over finite field)

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

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

Example

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

### 5.4.2 RightInverse (for a matrix over finite field)

▷ RightInverse(*m*) (attribute)
▷ LeftInverse(*m*) (attribute)

**Returns:** A matrix over a finite field.

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

Example

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

## 5.5 Matrices over the integers

In this section we describe operations in Semigroups specifically for integer matrices.

From v5.0.0, Semigroups uses the GAP library implementation of matrices over the integers belonging to the category IsMatrixObj (**Reference: IsMatrixObj**) rather than the previous implementation in the Semigroups package. This means that from v5.0.0, matrices over the integers no longer belong to the category IsMatrixOverSemiring (5.1.1).

The following methods are implemented in Semigroups for matrix objects over the integers.

### 5.5.1 InverseOp (for an integer matrix)

▷ `InverseOp(`*mat*`)` (operation)

**Returns:** An integer matrix.

If *mat* is an integer matrix whose inverse (if it exists) is also an integer matrix (i.e. a matrix whose `BaseDomain` (**Reference: BaseDomain**) is `Integers` (**Reference: Integers**)), then `InverseOp` returns the inverse of *mat*.

An integer matrix has an integer matrix inverse if and only if it has determinant $\pm 1$.

Example

```
gap> mat := Matrix(Integers, [[0, 0, -1],
>                             [0, 1, 0],
>                             [1, 0, 0]]);
<3x3 plist matrix over Integers>
gap> InverseOp(mat);
<3x3 plist matrix over Integers>
gap> mat * InverseOp(mat) = One(mat);
true
```

### 5.5.2 IsTorsion (for an integer matrix)

▷ `IsTorsion(`*mat*`)` (attribute)

**Returns:** `true` or `false`

If *mat* is an integer matrix (i.e. a matrix whose `BaseDomain` (**Reference: BaseDomain**) is `Integers` (**Reference: Integers**)), then `IsTorsion` returns `true` if *mat* is torsion and `false` otherwise.

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

Example

```
gap> mat := Matrix(Integers, [[0, 0, -1],
>                             [0, 1, 0],
>                             [1, 0, 0]]);
<3x3 plist matrix over Integers>
gap> IsTorsion(mat);
true
gap> mat := Matrix(Integers, [[0, 0, -1, 0],
>                             [0, -1, 0, 0],
>                             [4, 4, 2, -1],
>                             [1, 1, 0, 3]]);
<4x4 plist matrix over Integers>
gap> IsTorsion(mat);
false
```

### 5.5.3 Order

▷ `Order(`*mat*`)` (attribute)

**Returns:** An integer or `infinity`.

If *mat* is an integer matrix, then `Order` returns the order of *mat*. The order of *mat* is the smallest integer power of *mat* equal to the identity. If no such integer exists, the order is equal to `infinity`.

Example

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

## 5.6 Max-plus and min-plus matrices

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

- InverseOp (5.6.1)
- RadialEigenvector (5.6.2)
- SpectralRadius (5.6.3)
- UnweightedPrecedenceDigraph (5.6.4)

### 5.6.1 InverseOp

▷ InverseOp(*mat*) (operation)

**Returns:** A max-plus matrix.

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

Example

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

### 5.6.2 RadialEigenvector

▷ RadialEigenvector(*mat*) (operation)

**Returns:** A vector.

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

Example

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

### 5.6.3 SpectralRadius

▷ SpectralRadius(*mat*) (operation)

**Returns:** A rational number.

If *mat* is a max-plus matrix (i.e. belongs to the category IsMaxPlusMatrix (5.1.8)), then SpectralRadius returns the supremum of the set of eigenvalues of *mat*. This method is described in [Bac92].

Example

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

### 5.6.4 UnweightedPrecedenceDigraph

▷ UnweightedPrecedenceDigraph(*mat*) (operation)

**Returns:** A digraph.

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

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

Example

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

## 5.7 Matrix semigroups

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

- IsXMatrixSemigroup (5.7.1)
- IsXMatrixMonoid (5.7.2)
- IsFinite (5.7.3)
- IsTorsion (5.7.4)
- NormalizeSemigroup (5.7.5)

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

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

### 5.7.1 Matrix semigroup filters

▷ IsMatrixOverSemiringSemigroup(*obj*) (category)
▷ IsBooleanMatSemigroup(*obj*) (category)
▷ IsMatrixOverFiniteFieldSemigroup(*obj*) (category)
▷ IsMaxPlusMatrixSemigroup(*obj*) (category)
▷ IsMinPlusMatrixSemigroup(*obj*) (category)
▷ IsTropicalMatrixSemigroup(*obj*) (category)
▷ IsTropicalMaxPlusMatrixSemigroup(*obj*) (category)
▷ IsTropicalMinPlusMatrixSemigroup(*obj*) (category)
▷ IsNTPMatrixSemigroup(*obj*) (category)
▷ IsIntegerMatrixSemigroup(*obj*) (category)

**Returns:** true or false.

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

### 5.7.2 Matrix monoid filters

▷ IsMatrixOverSemiringMonoid(*obj*) (category)
▷ IsBooleanMatMonoid(*obj*) (category)
▷ IsMatrixOverFiniteFieldMonoid(*obj*) (category)
▷ IsMaxPlusMatrixMonoid(*obj*) (category)
▷ IsMinPlusMatrixMonoid(*obj*) (category)
▷ IsTropicalMatrixMonoid(*obj*) (category)
▷ IsTropicalMaxPlusMatrixMonoid(*obj*) (category)
▷ IsTropicalMinPlusMatrixMonoid(*obj*) (category)
▷ IsNTPMatrixMonoid(*obj*) (category)
▷ IsIntegerMatrixMonoid(*obj*) (category)

**Returns:** true or false.

The above are the currently supported types of matrix monoids. They correspond to the matrix semigroup types in Section 5.7.1.

### 5.7.3 IsFinite

▷ IsFinite(*S*) (property)

**Returns:** true or false.

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

Example

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

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

### 5.7.4 IsTorsion

▷ `IsTorsion(`*S*`)` (attribute)

**Returns:** `true` or `false`.

If *S* is a max-plus matrix semigroup (i.e. belongs to the category `IsMaxPlusMatrixSemigroup` (5.7.1)), then `IsTorsion` returns `true` if *S* is torsion and `false` otherwise. This method is based on [Gau96] and draws on [Bur16], [Bac92] and [Far09].

Example

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

### 5.7.5 NormalizeSemigroup

▷ `NormalizeSemigroup(`*S*`)` (operation)

**Returns:** A semigroup.

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

Example

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

# Chapter 6

# Semigroups and monoids defined by generating sets

In this chapter we describe the various ways that semigroups and monoids defined by generating sets can be created in Semigroups ; where the generators are, for example, those elements described in earlier chapters of this manual.

## 6.1 Underlying algorithms

Computing the Green's structure of a semigroup or monoid is a fundamental step in almost every algorithm for semigroups. There are two fundamental algorithms in the Semigroups package for computing the Green's structure of a semigroup defined by a set of generators. In the next two subsections we briefly describe these two algorithms.

### 6.1.1 Acting semigroups

The first of the fundamental algorithms for computing a semigroup defined by a generating set is described in [EEMP19]. When applied to a semigroup or monoid with relatively large subgroups, or $\mathscr{D}$-classes, these are the most efficient methods in the Semigroups package. For example, the complexity of computing, say, the size of a transformation semigroup that happens to be a group, is the same as the complexity of the Schreier-Sims Algorithm (polynomial in the number of points acted on by the transformations) for a permutation group.

In theory, these algorithms can be applied to compute any subsemigroup of a regular semigroup; but so far in the Semigroups package they are only implemented for semigroups of: transformations (see (**Reference: Transformations**)), partial permutations (see (**Reference: Partial permutations**)), bipartitions (see Chapter 3), matrices over a finite field (see Section 5.4); subsemigroups of regular Rees 0-matrix semigroups over permutation groups (see Chapter (**Reference: Rees Matrix Semigroups**)), and subsemigroups of McAlister triples (see Section 8.4).

We refer to semigroups to which the algorithms in [EEMP19] can be applied as *acting semigroups*, and such semigroups belong to the category `IsActingSemigroup` (6.1.2).

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

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

There are methods in Semigroups for computing individual Green's classes of an acting semigroup without computing the entire data structure of the underlying semigroup; see `GreensRClassOfElementNC` (10.1.3). It is also possible to compute the $\mathscr{R}$-classes, the number of elements and test membership in a semigroup without computing all the elements; see, for example, `GreensRClasses` (10.1.4), `RClassReps` (10.1.5), `IteratorOfRClasses` (10.2.2), or `NrRClasses` (10.1.9). This may be useful if you want to study a very large semigroup where computing all the elements of the semigroup is not feasible.

### 6.1.2 IsActingSemigroup

▷ `IsActingSemigroup(`*obj*`)` (category)

**Returns:** `true` or `false`.

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

Example

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

### 6.1.3 The Froidure-Pin Algorithm

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

Both implementations outperform the algorithms for acting semigroups when applied to semigroups with small (trivial) subgroups. This method is also used to determine the structure of a semigroup when the algorithms described in [EEMP19] do not apply. It is possible to specify which methods should be applied to a given semigroup; see Section 6.3.

A semigroup to which the Froidure-Pin Algorithm can be applied in Semigroups satisfy `CanUseFroidurePin` (6.1.4). Every acting semigroup in Semigroups satisfies `CanUseFroidurePin` (6.1.4) and the Froidure-Pin Algorithm is used to compute certain properties or attributes.

Currently, the libsemigroups implementation of the Froidure-Pin Algorithm can be applied to semigroups consisting of the following types of elements: transformations (see (**Reference: Transformations**)), partial permutations (see (**Reference: Partial permutations**)), bipartitions (see Chapter 3), partitioned binary relations (see Chapter 4) as defined in [MM13]; and matrices over the following semirings:

- the *Boolean semiring* $\{0,1\}$ where $0+0=0$, $0+1=1+1=1+0=1$, and $1\cdot 0=0\cdot 0=0\cdot 1=0$

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

(see Chapter 5).

The version of the Froidure-Pin Algorithm [FP97] written in C within the Semigroups package kernel module can be used to compute any other semigroup in GAP which satisfies `CanUseGapFroidurePin` (6.1.4). In theory, any finite semigroup can be computed using this algorithm. However, the condition that the semigroup satisfies `CanUseGapFroidurePin` (6.1.4) is imposed to avoid this method being used when it is inappropriate. If implementing a new type of semigroup in GAP, then simply do

Example

```
InstallTrueMethod(CanUseGapFroidurePin,
                        MyNewSemigroupType);
```

to make your new semigroup type `MyNewSemigroupType` use this version of the Froidure-Pin Algorithm. To make this work efficiently it is necessary that a hash function is implemented for the elements of `MyNewSemigroupType`; more details will be included in a future edition of this manual.

Mostly due to the way that GAP handles memory, this implementation is approximately 4 times slower than the implementation in libsemigroups . This version of the Froidure-Pin Algorithm is included because it applies to a wider class of semigroups than those currently implemented in libsemigroups and it is more straightforward to extend the classes of semigroups to which it applies.

### 6.1.4 CanUseFroidurePin

▷ `CanUseFroidurePin(`*obj*`)` (property)
▷ `CanUseGapFroidurePin(`*obj*`)` (property)
▷ `CanUseLibsemigroupsFroidurePin(`*obj*`)` (property)

**Returns:** `true` or `false`.

Every semigroup satisfying `CanUseFroidurePin` is a valid input for the Froidure-Pin algorithm; see Section 6.1.3 for more details.

Basic operations for semigroups satisfying `CanUseFroidurePin` are: `AsListCanonical` (11.1.1), `EnumeratorCanonical` (11.1.1), `IteratorCanonical` (11.1.1), `PositionCanonical` (11.1.2), `Enumerate` (11.1.3), and `IsEnumerated` (11.1.4).

Example

```
gap> S := Semigroup(Transformation([1, 3, 2]));;
gap> CanUseFroidurePin(S);
true
```

```
gap> S := FreeSemigroup(3);;
gap> CanUseFroidurePin(S);
false
```

## 6.2 Semigroups represented by generators

### 6.2.1 InverseMonoidByGenerators

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

**Returns:** An inverse monoid or semigroup.

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

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

## 6.3 Options when creating semigroups

When using any of the functions:

- `InverseSemigroup` (**Reference: InverseSemigroup**),
- `InverseMonoid` (**Reference: InverseMonoid**),
- `Semigroup` (**Reference: Semigroup**),
- `Monoid` (**Reference: Monoid**),
- `SemigroupByGenerators` (**Reference: SemigroupByGenerators**),
- `MonoidByGenerators` (**Reference: MonoidByGenerators**),
- `ClosureSemigroup` (6.4.1),
- `ClosureMonoid` (6.4.1),
- `ClosureInverseSemigroup` (6.4.1),
- `ClosureInverseMonoid` (6.4.1),
- `SemigroupIdeal` (9.1.1)

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

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

`acting`
: this component should be `true` or `false`. Roughly speaking, there are two types of methods in the Semigroups package: those for semigroups which have to be fully enumerated, and those for semigroups that do not; see Section 1.1. In order for a semigroup to use the latter methods in Semigroups it must satisfy `IsActingSemigroup` (6.1.2). By default any semigroup or monoid of transformations, partial permutations, Rees 0-matrix elements, or bipartitions satisfies `IsActingSemigroup`.

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

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

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

  The default value for this option is `false`.

`hashlen`
: this component should be a positive integer, which roughly specifies the lengths of the hash tables used internally by Semigroups . Semigroups uses hash tables in several fundamental methods. The lengths of these tables are a compromise between performance and memory usage; larger tables provide better performance for large computations but use more memory. Note that it is unlikely that you will need to specify this option unless you find that GAP runs out of memory unexpectedly or that the performance of Semigroups is poorer than expected. If you find that GAP runs out of memory unexpectedly, or you plan to do a large number of computations with relatively small semigroups (say with tens of thousands of elements), then you might consider setting `hashlen` to be less than the default value of `12517` 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 `12517`.

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

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

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

cong_by_ker_trace_threshold
: this should be a positive integer, which specifies a semigroup size. If *S* is a semigroup with inverse op, and *S* has a size greater than or equal to this threshold, then any congruence defined on it may use the "kernel and trace" method to perform calculations. If its size is less than the threshold, then other methods will be used instead. The "kernel and trace" method has better complexity than the generic method, but has large overheads which make it a poor choice for small semigroups. The default value for this component is `10 ^ 5`. See Section 13.7 for more information about the "kernel and trace" method.

Example

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

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

Example

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

If you want to change the default values of the options stored in SEMIGROUPS.DefaultOptionsRec (6.3.1) for all GAP sessions, then you can edit these values in the file semigroups-5.7.1/gap/options.g.

### 6.3.1 SEMIGROUPS.DefaultOptionsRec

▷ SEMIGROUPS.DefaultOptionsRec (global variable)

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

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

## 6.4 Subsemigroups and supersemigroups

### 6.4.1 ClosureSemigroup

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

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

These operations return the semigroup, monoid, inverse semigroup or inverse monoid generated by the argument *S* and the collection of elements *coll* after removing duplicates and elements from

*coll* that are already in *S*. In most cases, the new semigroup knows at least as much information about its structure as was already known about that of *S*.

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

Example

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

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

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

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

Example

```
gap> gens := [Transformation([2, 6, 7, 2, 6, 1, 1, 5]),
>             Transformation([3, 8, 1, 4, 5, 6, 7, 1]),
>             Transformation([4, 3, 2, 7, 7, 6, 6, 5]),
>             Transformation([7, 1, 7, 4, 2, 5, 6, 3])];;
gap> S := Monoid(gens[1]);;
gap> for x in gens do
>      S := ClosureSemigroup(S, x);
>    od;
gap> S;
<transformation monoid of degree 8 with 4 generators>
gap> Size(S);
233606
gap> S := Monoid(PartialPerm([1]));
<trivial partial perm group of rank 1 with 1 generator>
gap> T := ClosureMonoid(S, [PartialPerm([2 .. 5])]);
<partial perm monoid of rank 5 with 2 generators>
gap> One(T);
<identity partial perm on [ 1, 2, 3, 4, 5 ]>
gap> T := ClosureSemigroup(S, [PartialPerm([2 .. 5])]);
<partial perm semigroup of rank 4 with 2 generators>
gap> One(T);
fail
gap> ClosureInverseMonoid(DualSymmetricInverseMonoid(3),
>                         DClass(DualSymmetricInverseMonoid(3),
>                                IdentityBipartition(3)));
<inverse block bijection monoid of degree 3 with 3 generators>
gap> S := InverseSemigroup(Bipartition([[1, -1, -3], [2, 3, -2]]),
>                          Bipartition([[1, -3], [2, -2], [3, -1]]));;
gap> T := ClosureInverseSemigroup(S, DClass(PartitionMonoid(3),
> IdentityBipartition(3)));
<inverse block bijection semigroup of degree 3 with 3 generators>
gap> T := ClosureInverseSemigroup(T, [T.1, T.1, T.1]);
<inverse block bijection semigroup of degree 3 with 3 generators>
```

```
gap> S := InverseMonoid([
> PartialPerm([5, 9, 10, 0, 6, 3, 8, 4, 0]),
> PartialPerm([10, 7, 0, 8, 0, 0, 5, 9, 1])]);;
gap> x := PartialPerm([
> 5, 1, 7, 3, 10, 0, 2, 12, 0, 14, 11, 0, 16, 0, 0, 0, 0, 6, 9, 15]);
[4,3,7,2,1,5,10,14][8,12][13,16][18,6][19,9][20,15](11)
gap> S := ClosureInverseSemigroup(S, x);
<inverse partial perm semigroup of rank 19 with 4 generators>
gap> Size(S);
9744
gap> T := Idempotents(SymmetricInverseSemigroup(10));;
gap> S := ClosureInverseSemigroup(S, T);
<inverse partial perm semigroup of rank 19 with 14 generators>
```

### 6.4.2 SubsemigroupByProperty (for a semigroup and function)

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

**Returns:** A semigroup.

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

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

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

Example

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

### 6.4.3 InverseSubsemigroupByProperty

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

**Returns:** An inverse semigroup.

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

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

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

Example

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

## 6.5 Changing the representation of a semigroup

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

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

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

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

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

(6.5.2) can be used to change the representation of a monoid, but only the latter is guaranteed to return an object in the category of monoids.

Example

```
gap> S := Monoid(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> AsSemigroup(IsBooleanMatSemigroup, S);
<monoid of 10x10 boolean matrices with 2 generators>
gap> AsMonoid(IsBooleanMatMonoid, S);
<monoid of 10x10 boolean matrices with 2 generators>
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> AsSemigroup(IsBooleanMatSemigroup, S);
<semigroup of 10x10 boolean matrices with 2 generators>
gap> AsMonoid(IsBooleanMatMonoid, S);
<monoid of 8x8 boolean matrices with 2 generators>
gap> M := Monoid([
> Bipartition([[1, -3], [2, 3, 6], [4, 7, -6], [5, -8], [8, -4, -5],
>              [-1], [-2], [-7]]),
> Bipartition([[1, 3, -6], [2, -8], [4, 8, -1], [5], [6, -3, -4],
>              [7], [-2], [-5], [-7]]),
> Bipartition([[1, 2, 4, -3, -7, -8], [3, 5, 6, 8, -4, -6],
>              [7, -1, -2, -5]])]);;
gap> AsMonoid(IsPBRMonoid, M);
<pbr monoid of size 163, degree 163 with 3 generators>
gap> AsSemigroup(IsPBRSemigroup, M);
<pbr semigroup of size 163, degree 8 with 4 generators>
```

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

### 6.5.1 IsomorphismSemigroup

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

**Returns:** An isomorphism of semigroups.

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

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

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

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

no method when *filt* is IsPartialPermSemigroup (**Reference: IsPartialPermSemigroup**) and when *S* is a non-inverse semigroup.

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

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

Example

```
gap> S := Semigroup([
> Bipartition([
>   [1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
> Bipartition([
>   [1, -4], [2, 3, 4, 5], [6], [-1, -6], [-2, -3], [-5]])]);
<bipartition semigroup of degree 6 with 2 generators>
gap> IsomorphismSemigroup(IsTransformationSemigroup, S);
<bipartition semigroup of size 11, degree 6 with 2 generators> ->
<transformation semigroup of size 11, degree 12 with 2 generators>
gap> IsomorphismSemigroup(IsBooleanMatSemigroup, S);
<bipartition semigroup of size 11, degree 6 with 2 generators> ->
<semigroup of size 11, 12x12 boolean matrices with 2 generators>
gap> IsomorphismSemigroup(IsFpSemigroup, S);
<bipartition semigroup of size 11, degree 6 with 2 generators> ->
<fp semigroup with 2 generators and 5 relations of length 27>
gap> S := InverseSemigroup([
> PartialPerm([1, 2, 3, 6, 8, 10],
>             [2, 6, 7, 9, 1, 5]),
> PartialPerm([1, 2, 3, 4, 6, 7, 8, 10],
>             [3, 8, 1, 9, 4, 10, 5, 6])]);;
gap> IsomorphismSemigroup(IsBipartitionSemigroup, S);
<inverse partial perm semigroup of rank 10 with 2 generators> ->
<inverse bipartition semigroup of degree 10 with 2 generators>
gap> S := SymmetricInverseMonoid(4);
<symmetric inverse monoid of degree 4>
gap> IsomorphismSemigroup(IsBlockBijectionSemigroup, S);
<symmetric inverse monoid of degree 4> ->
<inverse block bijection monoid of degree 5 with 3 generators>
gap> Size(Range(last));
209
gap> S := Semigroup([
> PartialPerm([3, 1]), PartialPerm([1, 3, 4])]);;
gap> IsomorphismSemigroup(IsBlockBijectionSemigroup, S);
<partial perm semigroup of rank 3 with 2 generators> ->
<block bijection semigroup of degree 5 with 2 generators>
```

### 6.5.2 IsomorphismMonoid

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

**Returns:** An isomorphism of monoids.

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

The first argument *filt* must be of the form IsXMonoid, for example, IsTransformationMonoid (**Reference: IsTransformationMonoid**), IsFpMonoid (**Reference: IsFpMonoid**), and IsBipartitionMonoid (3.8.1) are some possible values for *filt*. Note that *filt* should not be of the form IsXSemigroup; see IsomorphismSemigroup (6.5.1). The second argument *S* should be a semigroup which is mathematically a monoid but which may or may not belong to the category IsMonoid (**Reference: IsMonoid**) of monoids in GAP, i.e. *S* must satisfy IsMonoidAsSemigroup (12.1.13).

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

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

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

Example

```
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));
<transformation semigroup of degree 10 with 2 generators>
gap> IsomorphismMonoid(IsTransformationMonoid, S);
<transformation semigroup of degree 10 with 2 generators> ->
<transformation monoid of degree 8 with 2 generators>
gap> IsomorphismMonoid(IsBooleanMatMonoid, S);
<transformation semigroup of degree 10 with 2 generators> ->
<monoid of 8x8 boolean matrices with 2 generators>
gap> IsomorphismMonoid(IsFpMonoid, S);
<transformation semigroup of degree 10 with 2 generators> ->
<fp monoid with 2 generators and 17 relations of length 278>
```

### 6.5.3 AsSemigroup

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

**Returns:** A semigroup.

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

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

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

Example

```
gap> S := Semigroup([
> Bipartition([
```

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

### 6.5.4 AsMonoid

▷ `AsMonoid(`*`[filt, ]S`*`)` (operation)

**Returns:** A monoid or `fail`.

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

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

If the first argument *filt* is omitted and the semigroup *S* is not a monoid, i.e. it does not satisfy `IsMonoidAsSemigroup` (12.1.13), then `fail` is returned.

Example

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

### 6.5.5 IsomorphismPermGroup

▷ `IsomorphismPermGroup(S)` (attribute)

**Returns:** An isomorphism.

If the semigroup $S$ is mathematically a group, so that it satisfies `IsGroupAsSemigroup` (12.1.7), then `IsomorphismPermGroup` returns an isomorphism to a permutation group.

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

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

Example

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

### 6.5.6 RZMSNormalization

▷ `RZMSNormalization(R)` (attribute)

**Returns:** An isomorphism.

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

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

  If the index sets $I$ and $\Lambda$ are partitioned into $k$ parts according to the `RZMSConnectedComponents` (11.14.2) of $S$, giving a disjoint union $I = I_1 \cup \ldots \cup I_k$ and

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

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

Furthermore, if $T$ is a group (i.e. a semigroup for which `IsGroupAsSemigroup` (12.1.7) returns `true`), then the non-zero entries of the structure matrix $Q$ are chosen such that the following hold:

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

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

Example

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

### 6.5.7 RMSNormalization

▷ RMSNormalization(*R*) (attribute)

**Returns:** An isomorphism.

If *R* is a Rees matrix semigroup over a group G (i.e. a semigroup for which IsGroupAsSemigroup (12.1.7) returns true), then RMSNormalization returns an isomorphism from *R* to a *normalized* Rees matrix semigroup S over G.

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

Example

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

### 6.5.8 IsomorphismReesMatrixSemigroup (for a semigroup)

▷ IsomorphismReesMatrixSemigroup(*S*) (attribute)
▷ IsomorphismReesZeroMatrixSemigroup(*S*) (attribute)
▷ IsomorphismReesMatrixSemigroupOverPermGroup(*S*) (attribute)
▷ IsomorphismReesZeroMatrixSemigroupOverPermGroup(*S*) (attribute)

**Returns:** An isomorphism.

If the semigroup *S* is finite and simple, then IsomorphismReesMatrixSemigroup returns an isomorphism to a Rees matrix semigroup over some group (usually a permutation group), and IsomorphismReesMatrixSemigroupOverPermGroup returns an isomorphism to a Rees matrix semigroup over a permutation group.

If *S* is finite and 0-simple, then IsomorphismReesZeroMatrixSemigroup returns an isomorphism to a Rees 0-matrix semigroup over some group (usually a permutation group), and IsomorphismReesZeroMatrixSemigroupOverPermGroup returns an isomorphism to a Rees 0-matrix semigroup over a permutation group.

See also InjectionPrincipalFactor (10.4.7).

Example

```
gap> S := Semigroup(PartialPerm([1]));
<trivial partial perm group of rank 1 with 1 generator>
gap> iso := IsomorphismReesMatrixSemigroup(S);;
gap> Source(iso) = S;
true
gap> Range(iso);
<Rees matrix semigroup 1x1 over Group(())>
gap> S := Semigroup(PartialPerm([1]), PartialPerm([]));
<partial perm monoid of rank 1 with 2 generators>
```

```
gap> Range(IsomorphismReesZeroMatrixSemigroup(S));
<Rees 0-matrix semigroup 1x1 over Group(())>
```

### 6.5.9 AntiIsomorphismDualFpSemigroup

▷ AntiIsomorphismDualFpSemigroup(*S*) (attribute)
▷ AntiIsomorphismDualFpMonoid(*S*) (attribute)

**Returns:** A finitely presented semigroup or monoid.

AntiIsomorphismDualFpSemigroup returns an anti-isomorphism (MappingByFunction (**Reference: MappingByFunction**)) from the finitely presented semigroup $S$ to another finitely presented semigroup. The range finitely presented semigroup is obtained from $S$ by reversing the relations of $S$.

AntiIsomorphismDualFpMonoid works analogously when $S$ is a finitely presented monoid, and the range of the returned anti-isomorphism is a finitely presented monoid.

Example

```
gap> F := FreeSemigroup("a", "b");
<free semigroup on the generators [ a, b ]>
gap> AssignGeneratorVariables(F);
gap> R := [[a ^ 3, a], [b ^ 2, b], [(a * b) ^ 2, a]];
[ [ a^3, a ], [ b^2, b ], [ (a*b)^2, a ] ]
gap> S := F / R;
<fp semigroup with 2 generators and 3 relations of length 14>
gap> map := AntiIsomorphismDualFpSemigroup(S);
MappingByFunction( <fp semigroup with 2 generators and
  3 relations of length 14>, <fp semigroup with 2 generators and
  3 relations of length 14>
 , function( x ) ... end, function( x ) ... end )
gap> RelationsOfFpSemigroup(Range(map));
[ [ a^3, a ], [ b^2, b ], [ (b*a)^2, a ] ]
```

### 6.5.10 EmbeddingFpMonoid

▷ EmbeddingFpMonoid(*S*) (attribute)

**Returns:** A finitely presented monoid.

EmbeddingFpMonoid returns an embedding (SemigroupHomomorphismByImages (14.1.1)) from the finitely presented semigroup $S$ into a finitely presented monoid. If $S$ satisfies IsMonoidAsSemigroup (12.1.13), then the mapping returned by this function is an isomorphism (the same isomorphism as IsomorphismFpMonoid (**Reference: IsomorphismFpMonoid**)). If $S$ is not a monoid, then the range is isomorphic to $S$ with an identity adjoined (a new element not previously in $S$). The embedded copy of $S$ in the range can be recovered using Image (**Reference: Image**).

Example

```
gap> F := FreeSemigroup("a", "b");
<free semigroup on the generators [ a, b ]>
gap> AssignGeneratorVariables(F);
gap> R := [[a ^ 3, a], [b ^ 2, b], [(a * b) ^ 2, a]];
[ [ a^3, a ], [ b^2, b ], [ (a*b)^2, a ] ]
gap> S := F / R;
<fp semigroup with 2 generators and 3 relations of length 14>
gap> Size(S);
```

```
3
gap> IsMonoidAsSemigroup(S);
false
gap> map := EmbeddingFpMonoid(S);
<fp semigroup with 2 generators and 3 relations of length 14> ->
<fp monoid with 2 generators and 3 relations of length 14>
gap> Size(Range(map));
4
```

## 6.6 Random semigroups

### 6.6.1 RandomSemigroup

▷ `RandomSemigroup(`*arg...*`)` (function)
▷ `RandomMonoid(`*arg...*`)` (function)
▷ `RandomInverseSemigroup(`*arg...*`)` (function)
▷ `RandomInverseMonoid(`*arg...*`)` (function)

**Returns:** A semigroup.

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

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

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

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

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

**number of generators**

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

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

**degree / dimension**

The third argument, if present, should be a positive integer *`n`*, which specifies the degree or dimension of the generators. For example, if the first argument *`filter`* is

`IsTransformationSemigroup`, then the value of this argument is the degree of the transformations in the returned semigroup; or if *`filter`* is `IsMatrixOverFiniteFieldSemigroup`, then this argument is the dimension of the matrices in the returned semigroup.

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

**threshold**

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

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

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

Example

```
gap> RandomSemigroup();
<semigroup of 10x10 max-plus matrices with 12 generators>
gap> RandomMonoid(IsTransformationMonoid);
<transformation monoid of degree 9 with 7 generators>
gap> RandomMonoid(IsPartialPermMonoid, 2);
<partial perm monoid of rank 17 with 2 generators>
gap> RandomMonoid(IsPartialPermMonoid, 2, 3);
<partial perm monoid of rank 3 with 2 generators>
gap> RandomInverseSemigroup(IsTropicalMinPlusMatrixSemigroup);
<semigroup of 6x6 tropical min-plus matrices with 14 generators>
gap> RandomInverseSemigroup(IsTropicalMinPlusMatrixSemigroup, 1);
<semigroup of 6x6 tropical min-plus matrices with 14 generators>
gap> RandomSemigroup(IsTropicalMinPlusMatrixSemigroup, 2);
<semigroup of 11x11 tropical min-plus matrices with 2 generators>
gap> RandomSemigroup(IsTropicalMinPlusMatrixSemigroup, 2, 1);
<semigroup of 1x1 tropical min-plus matrices with 2 generators>
gap> RandomSemigroup(IsTropicalMinPlusMatrixSemigroup, 2, 1, 3);
gap> last.1;
Matrix(IsTropicalMinPlusMatrix, [[infinity]], 3)
gap> RandomSemigroup(IsNTPMatrixSemigroup, 2, 1, 3, 4);
<semigroup of 1x1 ntp matrices with 2 generators>
gap> last.1;
Matrix(IsNTPMatrix, [[2]], 3, 4)
gap> RandomSemigroup(IsReesMatrixSemigroup, 2, 2);
<Rees matrix semigroup 2x2 over
  <permutation group of size 659 with 1 generator>>
gap> RandomSemigroup(IsReesZeroMatrixSemigroup, 2, 2, Group((1, 2), (3, 4)));
<Rees 0-matrix semigroup 2x2 over Group([ (1,2), (3,4) ])>
gap> RandomInverseMonoid(IsMatrixOverFiniteFieldMonoid, 2, 2);
<monoid of 3x3 matrices over GF(421^4) with 3 generators>
gap> RandomInverseMonoid(IsMatrixOverFiniteFieldMonoid, 2, 2, GF(7));
<monoid of 3x3 matrices over GF(7) with 2 generators>
gap> RandomSemigroup(IsBipartitionSemigroup, 5, 5);
<bipartition semigroup of degree 5 with 5 generators>
gap> RandomMonoid(IsBipartitionMonoid, 5, 5);
```

```
<bipartition monoid of degree 5 with 5 generators>
gap> RandomSemigroup(IsBooleanMatSemigroup);
<semigroup of 3x3 boolean matrices with 18 generators>
gap> RandomMonoid(IsBooleanMatMonoid);
<monoid of 11x11 boolean matrices with 19 generators>
```

# Chapter 7

# Standard examples

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

## 7.1 Transformation semigroups

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

### 7.1.1 CatalanMonoid

▷ `CatalanMonoid(n)` (operation)

**Returns:** A transformation monoid.

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

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

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

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

Example

```
gap> S := CatalanMonoid(6);
<transformation monoid of degree 6 with 5 generators>
gap> Size(S);
132
```

### 7.1.2 EndomorphismsPartition

▷ `EndomorphismsPartition(list)` (operation)

**Returns:** A transformation monoid.

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

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

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

EndomorphismsPartition returns a monoid with a minimal size generating set, as described in [ABMS15].

Example

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

### 7.1.3 PartialTransformationMonoid

▷ PartialTransformationMonoid(*n*) (operation)

**Returns:** A transformation monoid.

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

Example

```
gap> S := PartialTransformationMonoid(5);
<regular transformation monoid of degree 6 with 4 generators>
gap> Size(S);
7776
```

### 7.1.4 SingularTransformationSemigroup

▷ SingularTransformationSemigroup(*n*) (operation)

▷ SingularTransformationMonoid(*n*) (operation)

**Returns:** The semigroup of non-invertible transformations.

If *n* is an 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(4);
<regular transformation semigroup ideal of degree 4 with 1 generator>
gap> Size(S);
232
```

### 7.1.5 Semigroups of order-preserving transformations

▷ OrderEndomorphisms(*n*) (operation)

▷ SingularOrderEndomorphisms(*n*) (operation)

▷ OrderAntiEndomorphisms(*n*) (operation)

▷ PartialOrderEndomorphisms(*n*) (operation)

▷ PartialOrderAntiEndomorphisms(*n*) (operation)

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

`OrderEndomorphisms(n)`

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

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

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

`SingularOrderEndomorphisms(n)`

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

`OrderAntiEndomorphisms(n)`

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

`PartialOrderEndomorphisms(n)`

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

`PartialOrderAntiEndomorphisms(n)`

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

Example

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

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

### 7.1.6 EndomorphismMonoid (for a digraph)

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

**Returns:** A monoid.

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

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

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

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

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

Example

```
gap> gr := Digraph(List([1 .. 3], x -> [1 .. 3]));;
gap> EndomorphismMonoid(gr);
<transformation monoid of degree 3 with 3 generators>
gap> gr := CompleteDigraph(3);;
gap> EndomorphismMonoid(gr);
<transformation group of size 6, degree 3 with 2 generators>
gap> S := EndomorphismMonoid(gr, [1, 2, 2]);;
gap> IsGroupAsSemigroup(S);
true
gap> Size(S);
2
gap> S := EndomorphismMonoid(gr, [[1], [2, 3]]);;
gap> IsGroupAsSemigroup(S);
true
```

## 7.2 Semigroups of partial permutations

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

### 7.2.1 MunnSemigroup

▷ `MunnSemigroup(S)` (attribute)

**Returns:** The Munn semigroup of a semilattice.

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

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

Example

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

### 7.2.2 RookMonoid

▷ `RookMonoid(n)` (operation)

**Returns:** An inverse monoid of partial permutations.

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

Example

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

### 7.2.3 Inverse monoids of order-preserving partial permutations

▷ `POI(n)` (operation)
▷ `PODI(n)` (operation)
▷ `POPI(n)` (operation)
▷ `PORI(n)` (operation)

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

`POI(n)`

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

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

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

`PODI(n)`

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

`POPI(n)`

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

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

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

`PORI(n)`

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

Example

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

## 7.3 Semigroups of bipartitions

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

### 7.3.1 PartitionMonoid

▷ `PartitionMonoid(n)` (operation)

▷ `RookPartitionMonoid(n)` (operation)

▷ SingularPartitionMonoid(*n*) (operation)

**Returns:** A bipartition monoid.

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

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

If *n* is positive, then RookPartitionMonoid returns a submonoid of the partition monoid of degree *n* + 1 consisting of those bipartitions with *n* + 1 and -*n* - 1 in the same block; see [HR05], [Gro06], and [Eas19].

Example

```
gap> S := PartitionMonoid(4);
<regular bipartition *-monoid of size 4140, degree 4 with 4
 generators>
gap> Size(S);
4140
gap> T := SingularPartitionMonoid(4);
<regular bipartition *-semigroup ideal of degree 4 with 1 generator>
gap> Size(S) - Size(T) = Factorial(4);
true
gap> S := RookPartitionMonoid(4);
<regular bipartition *-monoid of degree 5 with 5 generators>
gap> Size(S);
21147
```

### 7.3.2 BrauerMonoid

▷ BrauerMonoid(*n*) (operation)

▷ PartialBrauerMonoid(*n*) (operation)

▷ SingularBrauerMonoid(*n*) (operation)

**Returns:** A bipartition monoid.

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

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

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

Example

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

```
true
gap> Size(S);
76
```

### 7.3.3 JonesMonoid

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

**Returns:** A bipartition monoid.

If *n* is a non-negative integer, then this operation returns the Jones monoid of degree *n*. The *Jones monoid* is the subsemigroup of the Brauer monoid consisting of those bipartitions that are planar; see PlanarPartitionMonoid (7.3.9). The Jones monoid is sometimes referred to as the TEMPERLEY-LIEB MONOID.

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

Example

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

### 7.3.4 PartialJonesMonoid

▷ PartialJonesMonoid(*n*) (operation)

**Returns:** A bipartition monoid.

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

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

See PartialBrauerMonoid (7.3.2), JonesMonoid (7.3.3), and MotzkinMonoid (7.3.6).

Example

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

```
323
gap> Size(S);
143
gap> Size(T);
14
```

### 7.3.5 AnnularJonesMonoid

▷ AnnularJonesMonoid(*n*) (operation)

**Returns:** A bipartition monoid.

If *n* is a non-negative integer, then AnnularJonesMonoid returns the annular Jones monoid of degree *n*. The *annular Jones monoid* is the subsemigroup of the partition monoid consisting of all annular bipartitions whose blocks have size 2 (annular bipartitions are defined in Chapter 3). See BrauerMonoid (7.3.2).

Example

```
gap> S := AnnularJonesMonoid(4);
<regular bipartition *-monoid of degree 4 with 2 generators>
```

### 7.3.6 MotzkinMonoid

▷ MotzkinMonoid(*n*) (operation)

**Returns:** A bipartition monoid.

If *n* is a non-negative integer, then this operation returns the Motzkin monoid of degree *n*. The *Motzkin monoid* is the subsemigroup of the partial Brauer monoid consisting of those bipartitions that are planar (planar bipartitions are defined in Chapter 3).

Note that the Motzkin monoid of degree *n* contains the partial Jones monoid of degree *n*, but in general, these monoids are not equal; see PartialJonesMonoid (7.3.4).

Example

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

### 7.3.7 DualSymmetricInverseSemigroup

▷ DualSymmetricInverseSemigroup(*n*) (operation)
▷ DualSymmetricInverseMonoid(*n*) (operation)
▷ SingularDualSymmetricInverseMonoid(*n*) (operation)
▷ PartialDualSymmetricInverseMonoid(*n*) (operation)

**Returns:** An inverse bipartition monoid.

If *n* is a positive integer, then the operations DualSymmetricInverseSemigroup and DualSymmetricInverseMonoid return the dual symmetric inverse monoid of degree *n*, which is the subsemigroup of the partition monoid consisting of the block bijections of degree *n*.

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

PartialDualSymmetricInverseMonoid returns the submonoid of the dual symmetric inverse monoid of degree *n* + 1 consisting of those block bijections with *n* + 1 and -*n* - 1 in the same block; see [KM11] and [KMU15].

See IsBlockBijection (3.5.17).

Example

```
gap> Number(PartitionMonoid(3), IsBlockBijection);
25
gap> S := DualSymmetricInverseSemigroup(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> Size(S);
25
gap> S := PartialDualSymmetricInverseMonoid(5);
<inverse block bijection monoid of degree 6 with 4 generators>
gap> Size(S);
29072
```

### 7.3.8 UniformBlockBijectionMonoid

▷ UniformBlockBijectionMonoid(*n*) (operation)
▷ FactorisableDualSymmetricInverseMonoid(*n*) (operation)
▷ SingularUniformBlockBijectionMonoid(*n*) (operation)
▷ PartialUniformBlockBijectionMonoid(*n*) (operation)
▷ SingularFactorisableDualSymmetricInverseMonoid(*n*) (operation)
▷ PlanarUniformBlockBijectionMonoid(*n*) (operation)
▷ SingularPlanarUniformBlockBijectionMonoid(*n*) (operation)

**Returns:** An inverse bipartition monoid.

If *n* is a positive integer, then this operation returns the uniform block bijection monoid of degree *n*. The *uniform block bijection monoid* is the submonoid of the partition monoid consisting of the block bijections of degree *n* where the number of positive integers in a block equals the number of negative integers in that block. The uniform block bijection monoid is also referred to as the *factorisable dual symmetric inverse monoid*.

SingularUniformBlockBijectionMonoid returns the ideal of the uniform block bijection monoid consisting of the non-invertible elements (i.e. those not in the group of units), when *n* is at least 2.

PlanarUniformBlockBijectionMonoid returns the submonoid of the uniform block bijection monoid consisting of the planar elements (i.e. those in the planar partition monoid, see PlanarPartitionMonoid (7.3.9)).

SingularPlanarUniformBlockBijectionMonoid returns the ideal of the planar uniform block bijection monoid consisting of the non-invertible elements (i.e. those not in the group of units), when *n* is at least 2.

PartialUniformBlockBijectionMonoid returns the submonoid of the uniform block bijection monoid of degree *n* + 1 consisting of those uniform block bijections with *n* + 1 and -*n* - 1 in the same block.

See IsUniformBlockBijection (3.5.18).

Example

```
gap> S := UniformBlockBijectionMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> Size(PlanarUniformBlockBijectionMonoid(8));
128
gap> S := DualSymmetricInverseMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> IsFactorisableInverseMonoid(S);
false
gap> S := UniformBlockBijectionMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> IsFactorisableInverseMonoid(S);
true
gap> S := AsSemigroup(IsBipartitionSemigroup,
>                     SymmetricInverseMonoid(5));
<inverse bipartition monoid of degree 5 with 3 generators>
gap> IsFactorisableInverseMonoid(S);
true
gap> S := PartialUniformBlockBijectionMonoid(5);
<inverse block bijection monoid of degree 6 with 4 generators>
gap> NrIdempotents(S);
203
gap> IsFactorisableInverseMonoid(S);
true
```

### 7.3.9 PlanarPartitionMonoid

▷ `PlanarPartitionMonoid(`*n*`)` (operation)

▷ `SingularPlanarPartitionMonoid(`*n*`)` (operation)

**Returns:** A bipartition monoid.

If *n* is a positive integer, then this operation returns the planar partition monoid of degree *n* which is the monoid consisting of all the planar bipartitions of degree *n* (planar bipartitions are defined in Chapter 3).

`SingularPlanarPartitionMonoid` returns the ideal of the planar partition monoid consisting of the non-invertible elements (i.e. those not in the group of units).

Example

```
gap> S := PlanarPartitionMonoid(3);
<regular bipartition *-monoid of degree 3 with 5 generators>
gap> Size(S);
132
gap> T := SingularPlanarPartitionMonoid(3);
<regular bipartition *-semigroup ideal of degree 3 with 1 generator>
gap> Size(T);
131
gap> Difference(S, T);
[ <block bijection: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ]> ]
```

### 7.3.10 ModularPartitionMonoid

▷ ModularPartitionMonoid(*m*, *n*) (operation)
▷ SingularModularPartitionMonoid(*m*, *n*) (operation)
▷ PlanarModularPartitionMonoid(*m*, *n*) (operation)
▷ SingularPlanarModularPartitionMonoid(*m*, *n*) (operation)

**Returns:** A bipartition monoid.

If *m* and *n* are positive integers, then this operation returns the modular-*m* partition monoid of degree *n*. The *modular-m partition monoid* is the submonoid of the partition monoid such that the numbers of positive and negative integers contained in each block are congruent mod *m*.

SingularModularPartitionMonoid returns the ideal of the modular partition monoid consisting of the non-invertible elements (i.e. those not in the group of units), when either *m = n = 1* or *m, n > 1*.

PlanarModularPartitionMonoid returns the submonoid of the modular-*m* partition monoid consisting of the planar elements (i.e. those in the planar partition monoid, see PlanarPartitionMonoid (7.3.9)).

SingularPlanarModularPartitionMonoid returns the ideal of the planar modular partition monoid consisting of the non-invertible elements (i.e. those not in the group of units), when either *m = n = 1* or *m, n > 1*.

Example

```
gap> S := ModularPartitionMonoid(3, 6);
<regular bipartition *-monoid of degree 6 with 4 generators>
gap> Size(S);
36243
gap> S := SingularModularPartitionMonoid(1, 1);
<commutative inverse bipartition semigroup ideal of degree 1 with
  1 generator>
gap> Size(SingularModularPartitionMonoid(2, 4));
355
gap> S := PlanarModularPartitionMonoid(4, 9);
<regular bipartition *-monoid of degree 9 with 14 generators>
gap> Size(S);
1795
gap> S := SingularPlanarModularPartitionMonoid(3, 5);
<regular bipartition *-semigroup ideal of degree 5 with 1 generator>
gap> Size(SingularPlanarModularPartitionMonoid(1, 2));
13
```

### 7.3.11 ApsisMonoid

▷ ApsisMonoid(*m*, *n*) (operation)
▷ SingularApsisMonoid(*m*, *n*) (operation)
▷ CrossedApsisMonoid(*m*, *n*) (operation)
▷ SingularCrossedApsisMonoid(*m*, *n*) (operation)

**Returns:** A bipartition monoid.

If *m* and *n* are positive integers, then this operation returns the *m*-apsis monoid of degree *n*. The *m-apsis monoid* is the monoid of bipartitions generated when the diapses in generators of the Jones monoid are replaced with *m*-apses. Note that an *m-apsis* is a block that contains precisely *m* consecutive integers.

SingularApsisMonoid returns the ideal of the apsis monoid consisting of the non-invertible elements (i.e. those not in the group of units), when $m \leq n$.

CrossedApsisGeneratedMonoid returns the semigroup generated by the symmetric group of degree $n$ and the $m$-apsis monoid of degree $n$.

SingularCrossedApsisMonoid returns the ideal of the crossed apsis monoid consisting of the non-invertible elements (i.e. those not in the group of units), when $m$ <= $n$.

Example

```
gap> S := ApsisMonoid(3, 7);
<regular bipartition *-monoid of degree 7 with 5 generators>
gap> Size(S);
320
gap> T := SingularApsisMonoid(3, 7);
<regular bipartition *-semigroup ideal of degree 7 with 1 generator>
gap> Difference(S, T) = [One(S)];
true
gap> Size(CrossedApsisMonoid(2, 5));
945
gap> SingularCrossedApsisMonoid(4, 6);
<regular bipartition *-semigroup ideal of degree 6 with 1 generator>
```

## 7.4 Standard PBR semigroups

In this section, we describe the operations in Semigroups that can be used to create standard examples of semigroups of partitioned binary relations (PBRs). See Chapter 4 for more information about PBRs.

### 7.4.1 FullPBRMonoid

▷ FullPBRMonoid(n) (operation)

**Returns:** A PBR monoid.

If $n$ is a positive integer not greater than 2, then this operation returns the monoid consisting of all of the partitioned binary relations (PBRs) of degree $n$; called the *full PBR monoid*. There are 2 ^ ((2 * n) ^ 2) PBRs of degree $n$. The full PBR monoid of degree $n$ is currently too large to compute when $n \geq 3$.

The full PBR monoid is not regular in general.

Example

```
gap> S := FullPBRMonoid(1);
<pbr monoid of degree 1 with 4 generators>
gap> S := FullPBRMonoid(2);
<pbr monoid of degree 2 with 10 generators>
```

## 7.5 Semigroups of matrices over a finite field

In this section, we describe the operations in Semigroups that can be used to create semigroups of matrices over a finite field that belonging to several standard classes of example. See the section 'Matrices over finite fields' for more information about matrices over a finite field.

### 7.5.1 FullMatrixMonoid

▷ FullMatrixMonoid(*d*, *q*) (operation)
▷ GeneralLinearMonoid(*d*, *q*) (operation)
▷ GLM(*d*, *q*) (operation)

**Returns:** A matrix monoid.

These operations return the full matrix monoid of *d* by *d* matrices over the field with *q* elements. The *full matrix monoid*, also known as the *general linear monoid*, with these parameters, is the monoid consisting of all *d* by *d* matrices with entries from the field GF(*q*). This monoid has *q* ^ (*d* ^ 2) elements.

Example

```
gap> S := FullMatrixMonoid(2, 4);
<general linear monoid 2x2 over GF(2^2)>
gap> Size(S);
256
gap> S = GeneralLinearMonoid(2, 4);
true
gap> GLM(2, 2);
<general linear monoid 2x2 over GF(2)>
```

### 7.5.2 SpecialLinearMonoid

▷ SpecialLinearMonoid(*d*, *q*) (operation)
▷ SLM(*d*, *q*) (operation)

**Returns:** A matrix monoid.

These operations return the special linear monoid of *d* by *d* matrices over the field with *q* elements. The *special linear monoid* is the monoid consisting of all *d* by *d* matrices with entries from the field GF(*q*) that have determinant 0 or 1. In other words, the special linear monoid is formed from the general linear monoid of the same parameters by replacing its group of units (the general linear group) by the special linear group.

Example

```
gap> S := SpecialLinearMonoid(2, 4);
<regular monoid of 2x2 matrices over GF(2^2) with 3 generators>
gap> S = SLM(2, 4);
true
gap> Size(S);
136
```

### 7.5.3 IsFullMatrixMonoid

▷ IsFullMatrixMonoid(*S*) (property)
▷ IsGeneralLinearMonoid(*S*) (property)

IsFullMatrixMonoid and IsGeneralLinearMonoid return true if the semigroup S was created using either of the commands FullMatrixMonoid (7.5.1) or GeneralLinearMonoid (7.5.1) and false otherwise.

Example

```
gap> S := RandomSemigroup(IsTransformationSemigroup, 4, 4);;
gap> IsFullMatrixMonoid(S);
```

```
false
gap> S := GeneralLinearMonoid(3, 3);
<general linear monoid 3x3 over GF(3)>
gap> IsFullMatrixMonoid(S);
true
```

## 7.6 Semigroups of boolean matrices

In this section, we describe the operations in Semigroups that can be used to create semigroups of boolean matrices belonging to several standard classes of example. See the section 'Boolean matrices' for more information about boolean matrices.

### 7.6.1 FullBooleanMatMonoid

▷ `FullBooleanMatMonoid(d)` (operation)

**Returns:** The monoid of all boolean matrices of dimension `d`.

If `d` is a positive integer less than or equal to 5, then this operation returns the full boolean matrix monoid of dimension `d`. The *full boolean matrix monoid of dimension* `d` is the monoid consisting of all `d` by `d` boolean matrices, and has `2 ^ (n ^ 2)` matrices.

`FullBooleanMatMonoid` returns a monoid with a generating set that is minimal in size. These generating sets are pre-computed.

Example

```
gap> S := FullBooleanMatMonoid(3);
<monoid of 3x3 boolean matrices with 5 generators>
gap> Size(S);
512
```

### 7.6.2 RegularBooleanMatMonoid

▷ `RegularBooleanMatMonoid(d)` (operation)

**Returns:** A monoid of boolean matrices.

If `d` is a positive integer, then `RegularBooleanMatMonoid` returns the monoid generated by the regular `d` by `d` boolean matrices. Note that this monoid is *not* regular in general. `RegularBooleanMatMonoid(d)` is generated by the four boolean matrices `A, B, C, D`, whose `true` entries are:

- `A[i][i + 1]` and `A[n][1]`, for $i \in \{1,\ldots,n-1\}$;
- `B[1][2]`, `B[2][1]`, and `B[i][i]` for $i \in \{3,\ldots,n\}$;
- `C[1][2]` and `C[i][i]`, for $i \in \{2,\ldots,n-1\}$; and
- `D[1][2]`, `D[i][i]`, for $i \in \{2,\ldots,n\}$, and `D[n][1]`.

This monoid has nearly `2 ^ (n ^ 2)` elements.

Example

```
gap> S := RegularBooleanMatMonoid(3);
<monoid of 3x3 boolean matrices with 4 generators>
gap> Size(S);
506
```

### 7.6.3 ReflexiveBooleanMatMonoid

▷ `ReflexiveBooleanMatMonoid(d)` (operation)

**Returns:** A monoid of boolean matrices.

If `d` is a positive integer less than or equal to 5, then this operation returns the monoid consisting of all reflexive `d` by `d` boolean matrices. A boolean matrix `mat` is *reflexive* if each entry of its leading diagonal is `true`, i.e. if `mat[i][i]` is `true` for all $i \in \{1, \ldots, d\}$.

The generating sets for the monoids returned by `ReflexiveBooleanMatMonoid` are pre-computed, and read from a file. Small generating sets are not known for $d \geq 6$.

Example

```
gap> S := ReflexiveBooleanMatMonoid(3);
<monoid of 3x3 boolean matrices with 8 generators>
gap> Size(S);
64
```

### 7.6.4 HallMonoid

▷ `HallMonoid(d)` (operation)

**Returns:** A monoid of boolean matrices.

If `d` is a positive integer less than or equal to 5, then this operation returns the monoid consisting of Hall matrices of degree `d`. A *Hall matrix* is a boolean matrix in which every column and every row contains at least one `true` entry. Equivalently, a Hall matrix is a boolean matrix that contains a permutation.

A Hall matrix of dimension `d` corresponds to a solution to Hall's Marriage Problem, when there are two collections of `d` people. Thus the number of solutions to Hall's Marriage Problem in this instance is the number of elements of `HallMonoid(d)`.

The operation `HallMonoid` returns a monoid with a generating set that is minimal in size. These generating sets are pre-computed, and a minimal generating set is not known for larger dimensions.

Example

```
gap> S := HallMonoid(3);
<monoid of 3x3 boolean matrices with 4 generators>
gap> Size(S);
247
```

### 7.6.5 GossipMonoid

▷ `GossipMonoid(d)` (operation)

**Returns:** A monoid of boolean matrices.

If `d` is a positive integer, then this operation returns the `d` by `d` gossip monoid. The *gossip monoid* is defined to be the monoid generated by the collection of all `d` by `d` boolean matrices that define an equivalence relation; see `IsEquivalenceBooleanMat` (5.3.16).

For $d \geq 2$, `GossipMonoid(d)` returns a monoid with $\binom{d}{2}$ generators. The generating set is the collection of boolean matrices that define an equivalence relation that has one equivalence class of size 2, and no other non-trivial equivalence classes. Note that this generating set is strictly contained within the collection of all equivalence relation boolean matrices.

The number of elements of `GossipMonoid(d)` is known for some small values of `d` — see [BDF15] for more information about the gossip monoid, and its size for $d \leq 9$.

Example

```
gap> S := GossipMonoid(3);
<monoid of 3x3 boolean matrices with 3 generators>
gap> Size(S);
11
```

### 7.6.6 TriangularBooleanMatMonoid

▷ `TriangularBooleanMatMonoid(d)` (operation)
▷ `UnitriangularBooleanMatMonoid(d)` (operation)

**Returns:** A monoid of boolean matrices.

If `d` is a positive integer, then `TriangularBooleanMatMonoid` returns the monoid consisting of the upper-triangular `d` by `d` boolean matrices. A boolean matrix is *upper-triangular* if the entry in row `i`, column `j` is `false` whenever `i > j`.

`UnitriangularBooleanMatMonoid` returns the subsemigroup of the `TriangularBooleanMatMonoid` that consists of reflexive upper-triangular boolean matrices; see `ReflexiveBooleanMatMonoid` (7.6.3).

Example

```
gap> S := TriangularBooleanMatMonoid(3);
<monoid of 3x3 boolean matrices with 6 generators>
gap> Size(S);
64
gap> T := UnitriangularBooleanMatMonoid(4);
<monoid of 4x4 boolean matrices with 6 generators>
gap> Size(T);
64
```

## 7.7 Semigroups of matrices over a semiring

In this section, we describe the operations in Semigroups that can be used to create semigroups of matices over a semiring that belong to several standard classes of example. See Chapter 5 for more information about matrices over a semiring.

### 7.7.1 FullTropicalMaxPlusMonoid

▷ `FullTropicalMaxPlusMonoid(d, t)` (operation)

**Returns:** A monoid of tropical max plus matrices.

If `d = 2` and `t` is a positive integer, then `FullTropicalMaxPlusMonoid` returns the monoid consisting of all `d` by `d` matrices with entries from the tropical max-plus semiring with threshold `t`. A small generating set for larger values of `d` is not currently known.

This monoid contains `(t + 2) ^ (d ^ 2)` elements.

Example

```
gap> S := FullTropicalMaxPlusMonoid(2, 5);
<monoid of 2x2 tropical max-plus matrices with 24 generators>
gap> Size(S);
2401
gap> (5 + 2) ^ (2 ^ 2);
2401
```

### 7.7.2 FullTropicalMinPlusMonoid

▷ `FullTropicalMinPlusMonoid(d, t)` (operation)

**Returns:** A monoid of tropical min plus matrices.

If `d` is equal to 2 or 3, and `t` is a positive integer, then `FullTropicalMinPlusMonoid` returns the monoid consisting of all `d` by `d` matrices with entries from the tropical min-plus semiring with threshold `t`. A small generating set for larger values of `d` is not currently known.

This monoid contains `(t + 2) ^ (d ^ 2)` elements.

Example

```
gap> S := FullTropicalMinPlusMonoid(2, 3);
<monoid of 2x2 tropical min-plus matrices with 7 generators>
gap> Size(S);
625
gap> (3 + 2) ^ (2 ^ 2);
625
```

## 7.8 Examples in various representations

In this section, we describe the functions in Semigroups that can be used to create standard semigroups in various representations. For all of these examples, the default representation is as a semigroup of transformations. In general, these functions do not return a representation of minimal degree.

### 7.8.1 TrivialSemigroup

▷ `TrivialSemigroup([filt][,] [deg])` (function)

**Returns:** A trivial semigroup.

A TRIVIAL semigroup is a semigroup with precisely one element. This function returns a trivial semigroup in the representation given by the filter *filt*, and (if possible) with the degree of the representation given by the non-negative integer *deg*.

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

- `IsTransformationSemigroup` (the default, if *filt* is not specified),
- `IsPartialPermSemigroup`,
- `IsBipartitionSemigroup`,
- `IsBlockBijectionSemigroup`,
- `IsPBRSemigroup`,
- `IsBooleanMatSemigroup`.

If the optional argument *deg* is not specified, then the smallest possible degree will be used.

Example

```
gap> S := TrivialSemigroup();
<trivial transformation group of degree 0 with 1 generator>
gap> Size(S);
1
gap> S := TrivialSemigroup(3);
<trivial transformation group of degree 3 with 1 generator>
```

```
gap> S := TrivialSemigroup(IsBipartitionSemigroup, 2);
<trivial block bijection group of degree 2 with 1 generator>
gap> Elements(S);
[ <block bijection: [ 1, 2, -1, -2 ]> ]
```

### 7.8.2 MonogenicSemigroup

▷ `MonogenicSemigroup([filt, ]m, r)` (function)

**Returns:** A monogenic semigroup with index `m` and period `r`.

If `m` and `r` are positive integers, then this function returns a monogenic semigroup `S` with index `m` and period `r` in the representation given by the filter `filt`.

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

- `IsTransformationSemigroup` (the default, if `filt` is not specified),
- `IsPartialPermSemigroup`,
- `IsBipartitionSemigroup`,
- `IsBlockBijectionSemigroup`,
- `IsPBRSemigroup`,
- `IsBooleanMatSemigroup`.

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

See `IsMonogenicSemigroup` (12.1.12) for more information about monogenic semigroups.

Example

```
gap> S := MonogenicSemigroup(5, 3);
<commutative non-regular transformation semigroup of size 7, degree 8
 with 1 generator>
gap> IsMonogenicSemigroup(S);
true
gap> I := MinimalIdeal(S);;
gap> IsGroupAsSemigroup(I);
true
gap> StructureDescription(I);
"C3"
gap> S := MonogenicSemigroup(IsBlockBijectionSemigroup, 9, 1);
<commutative non-regular block bijection semigroup of size 9,
 degree 10 with 1 generator>
```

### 7.8.3 RectangularBand

▷ `RectangularBand([filt, ]m, n)` (function)

**Returns:** An `m` by `n` rectangular band.

If `m` and `n` are positive integers, then this function returns a semigroup isomorphic to an `m` by `n` rectangular band, in the representation given by the filter `filt`.

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

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

See IsRectangularBand (12.1.16) for more information about rectangular bands.

Example

```
gap> T := RectangularBand(5, 6);
<regular transformation semigroup of size 30, degree 10 with 6
 generators>
gap> IsRectangularBand(T);
true
gap> S := RectangularBand(IsReesMatrixSemigroup, 4, 8);
<Rees matrix semigroup 4x8 over Group(())>
gap> IsRectangularBand(S);
true
gap> IsCompletelySimpleSemigroup(S) and IsHTrivial(S);
true
```

### 7.8.4 FreeSemilattice

▷ FreeSemilattice(*[filt, ]n*) (function)

**Returns:** A free semilattice with *n* generators.

If *n* is a positive integer, then this function returns a free semilattice with *n* generators in the representation given by the filter *filt*. The optional argument *filt* may be one of the following:

- IsTransformationSemigroup (the default, if *filt* is not specified),
- IsTransformationMonoid,
- IsPartialPermSemigroup,
- IsPartialPermMonoid,
- IsFpSemigroup,
- IsFpMonoid,
- IsBipartitionSemigroup,
- IsBipartitionMonoid,
- IsPBRSemigroup,
- IsPBRMonoid,
- IsBooleanMatSemigroup,
- IsBooleanMatMonoid,

- IsNTPMatrixSemigroup,
- IsNTPMatrixMonoid,
- IsMaxPlusMatrixSemigroup,
- IsMaxPlusMatrixMonoid,
- IsMinPlusMatrixSemigroup,
- IsMinPlusMatrixMonoid,
- IsTropicalMaxPlusMatrixSemigroup,
- IsTropicalMaxPlusMatrixMonoid,
- IsTropicalMinPlusMatrixSemigroup,
- IsTropicalMinPlusMatrixMonoid,
- IsProjectiveMaxPlusMatrixSemigroup,
- IsProjectiveMaxPlusMatrixMonoid,
- IsIntegerMatrixSemigroup,
- IsIntegerMatrixMonoid.

Example

```
gap> S := FreeSemilattice(IsTransformationSemigroup, 5);
<inverse transformation semigroup of size 31, degree 6 with 5
 generators>
gap> T := FreeSemilattice(IsPartialPermSemigroup, 3);
<inverse partial perm semigroup of size 7, rank 3 with 3 generators>
gap> U := FreeSemilattice(IsBooleanMatSemigroup, 4);
<inverse semigroup of size 15, 5x5 boolean matrices with 4 generators>
```

### 7.8.5 ZeroSemigroup

▷ ZeroSemigroup(*[filt, ]n*) (function)

**Returns:** A zero semigroup of order *n*.

If *n* is a positive integer, then this function returns a zero semigroup of order *n* in the representation given by the filter *filt*.

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

- IsTransformationSemigroup (the default, if *filt* is not specified),
- IsPartialPermSemigroup,
- IsBipartitionSemigroup,
- IsBlockBijectionSemigroup,
- IsPBRSemigroup,

- IsBooleanMatSemigroup,
- IsReesZeroMatrixSemigroup (provided that *n* > 1).

See IsZeroSemigroup (12.1.31) for more information about zero semigroups.

Example

```
gap> S := ZeroSemigroup(5);
<commutative non-regular transformation semigroup of size 5, degree 5
 with 4 generators>
gap> IsZeroSemigroup(S);
true
gap> S := ZeroSemigroup(IsPartialPermSemigroup, 15);
<commutative non-regular partial perm semigroup of size 15, rank 14
 with 14 generators>
gap> Size(S);
15
gap> z := MultiplicativeZero(S);
<empty partial perm>
gap> IsZeroSemigroup(S);
true
gap> ForAll(S, x -> ForAll(S, y -> x * y = z));
true
```

### 7.8.6 LeftZeroSemigroup

▷ LeftZeroSemigroup(*[filt, ]n*) (function)
▷ RightZeroSemigroup(*[filt, ]n*) (function)

**Returns:** A left zero (or right zero) semigroup of order *n*.

If *n* is a positive integer, then this function returns a left zero (or right zero, as appropriate) semigroup of order *n* in the representation given by the filter *filt*. If *filt* is not specified then the default representation is IsTransformationSemigroup.

The function LeftZeroSemigroup([*filt*,] *n*) simply calls RectangularBand([*filt*,] *n*, 1) and the function RightZeroSemigroup([*filt*,] *n*) simply calls RectangularBand([*filt*,] 1, *n*).

For more information about RectangularBand, including its permitted values of *filt*, see RectangularBand (7.8.3). See IsLeftZeroSemigroup (12.1.10) and IsRightZeroSemigroup (12.1.19) for more information about left zero and right zero semigroups.

Example

```
gap> S := LeftZeroSemigroup(20);
<transformation semigroup of degree 6 with 20 generators>
gap> IsLeftZeroSemigroup(S);
true
gap> ForAll(Tuples(S, 2), p -> p[1] * p[2] = p[1]);
true
gap> S := RightZeroSemigroup(IsBipartitionSemigroup, 5);
<regular bipartition semigroup of size 5, degree 3 with 5 generators>
gap> IsRightZeroSemigroup(S);
true
```

### 7.8.7 BrandtSemigroup

▷ BrandtSemigroup(*[[filt, ]G, ]n*) (function)

**Returns:** An *n* by *n* Brandt semigroup over the group *G*.

If *n* is a positive integer, then this function returns an *n* by *n* Brandt semigroup over the group *G* in the representation given by the filter *filt*.

The optional argument *filt* can be any of the following:

- IsPartialPermSemigroup (the default, if *filt* is not specified),
- IsReesZeroMatrixSemigroup,
- IsTransformationSemigroup,
- IsBipartitionSemigroup,
- IsPBRSemigroup,
- IsBooleanMatSemigroup,
- IsNTPMatrixSemigroup,
- IsMaxPlusMatrixSemigroup,
- IsMinPlusMatrixSemigroup,
- IsTropicalMaxPlusMatrixSemigroup,
- IsTropicalMinPlusMatrixSemigroup,
- IsProjectiveMaxPlusMatrixSemigroup,
- IsIntegerMatrixSemigroup.

The optional argument *G* defaults to a trivial permutation group. If present *G* must be a permutation group, unless *filt* is IsReesZeroMatrixSemigroup when *G* may be any type of finite group.

See IsBrandtSemigroup (12.2.2) for more information about Brandt semigroups.

Example

```
gap> S := BrandtSemigroup(5);
<0-simple inverse partial perm semigroup of rank 5 with 4 generators>
gap> IsBrandtSemigroup(S);
true
gap> S := BrandtSemigroup(IsTransformationSemigroup, 15);
<0-simple transformation semigroup of degree 16 with 28 generators>
gap> Size(S);
226
gap> MultiplicativeZero(S);
Transformation( [ 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
  16, 16, 16 ] )
gap> S := BrandtSemigroup(Group((1, 2)), 3);
<0-simple inverse partial perm semigroup of rank 6 with 3 generators>
gap> S := BrandtSemigroup(IsTransformationSemigroup, Group((1, 2)), 3);
<0-simple transformation semigroup of degree 7 with 5 generators>
gap> S := BrandtSemigroup(IsReesZeroMatrixSemigroup,
```

```
>                          DihedralGroup(4),
>                          2);
<Rees 0-matrix semigroup 2x2 over <pc group of size 4 with
 2 generators>>
```

## 7.9 Free bands

This chapter describes the functions in Semigroups for dealing with free bands. This part of the manual and the functions described herein were originally written by Julius Jonušas, with later additions by Reinis Cirpons, Tom Conti-Leslie, and Murray Whyte

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.

### 7.9.1 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. In the second and third forms, the *`names`* must be distinct. The resulting semigroup is always finite.

Example

```
gap> FreeBand(6);
<free band on the generators [ x1, x2, x3, x4, x5, x6 ]>
gap> FreeBand(6, "b");
<free band on the generators [ b1, b2, b3, b4, b5, b6 ]>
gap> FreeBand("a", "b", "c");
<free band on the generators [ a, b, c ]>
gap> FreeBand("a", "b", "c");
<free band on the generators [ a, b, c ]>
gap> S := FreeBand(["a", "b", "c"]);
<free band on the generators [ a, b, c ]>
gap> Size(S);
159
gap> gens := Generators(S);
[ a, b, c ]
gap> S.1 * S.2;
ab
```

### 7.9.2 IsFreeBandCategory

▷ IsFreeBandCategory (category)

IsFreeBandCategory is the category of semigroups created using FreeBand (7.9.1).

Example

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

### 7.9.3 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
```

### 7.9.4 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
```

### 7.9.5 IsFreeBandElementCollection

▷ IsFreeBandElementCollection (Category)

Every collection of elements of a free band belongs to the category IsFreeBandElementCollection. For example, every free band belongs to IsFreeBandElementCollection.

### 7.9.6 IsFreeBandSubsemigroup

▷ IsFreeBandSubsemigroup (filter)

IsFreeBandSubsemigroup is a synonym for IsSemigroup and IsFreeBandElementCollection.

Example

```
gap> S := FreeBand(2);
<free band on the generators [ x1, x2 ]>
gap> x := S.1;
x1
gap> y := S.2;
x2
gap> new := Semigroup([x * y, x]);
<semigroup with 2 generators>
gap> IsFreeBand(new);
false
gap> IsFreeBandSubsemigroup(new);
true
```

### 7.9.7 ContentOfFreeBandElement

▷ ContentOfFreeBandElement(*x*) (attribute)
▷ ContentOfFreeBandElementCollection(*coll*) (attribute)

**Returns:** A list of integers

The content of a free band element *x* is the set of generators appearing in the word representing the element *x* of the free band.

The function ContentOfFreeBandElement returns the content of the free band element *x* represented as a list of integers, where 1 represents the first generator, 2 the second generator, and so on.

The function ContentOfFreeBandElementCollection returns the least list C for the collection of free band elements *coll* such that the content of every element in *coll* is contained in C.

Example

```
gap> S := FreeBand(2);
<free band on the generators [ x1, x2 ]>
gap> x := S.1;
x1
gap> y := S.2;
x2
gap> ContentOfFreeBandElement(x);
[ 1 ]
gap> ContentOfFreeBandElement(x * y);
[ 1, 2 ]
gap> ContentOfFreeBandElement(x * y * x);
[ 1, 2 ]
gap> ContentOfFreeBandElementCollection([x, y]);
[ 1, 2 ]
```

### 7.9.8 EqualInFreeBand

▷ EqualInFreeBand(*u*, *v*) (operation)

This operation takes a pair *u* and *v* of lists of positive integers or strings, representing words in a free semigroup.

Where F is a free band over some alphabet containing the letters occurring in $u$ and $v$, this operation returns true if $u$ and $v$ are equal in F, and false otherwise.

Note that this operation is for lists and strings, as opposed to FreeBandElement objects.

This is an implementation of an algorithm described by Jakub Radoszewski and Wojciech Rytter in [RR10].

Example

```
gap> EqualInFreeBand("aa", "a");
true
gap> EqualInFreeBand("abcacba", "abcba");
true
gap> EqualInFreeBand("aab", "aac");
false
gap> EqualInFreeBand([1, 3, 3], [2]);
false
```

### 7.9.9 GreensDClassOfElement (for a free band and element)

▷ GreensDClassOfElement($S$, $x$) (operation)

**Returns:** A Green's $\mathscr{D}$-class

Let $S$ be a free band. Two elements of $S$ are $\mathscr{D}$-related if and only if they have the same content i.e. the set of generators appearing in any factorization of the elements. Therefore, a $\mathscr{D}$-class of a free band element $x$ is the set of elements of $S$ which have the same content as $x$.

Example

```
gap> S := FreeBand(3, "b");
<free band on the generators [ b1, b2, b3 ]>
gap> x := S.1 * S.2;
b1b2
gap> D := GreensDClassOfElement(S, x);
<Green's D-class: b1b2>
gap> IsGreensDClass(D);
true
```

### 7.9.10 Operators

The following operators are also included 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.

## 7.10 Graph inverse semigroups

In this chapter we describe a class of semigroups arising from directed graphs.

The functionality in Semigroups for graph inverse semigroups was written jointly by Zak Mesyan (UCCS) and J. D. Mitchell (St Andrews). The functionality for graph inverse semigroup congruences was written by Marina Anagnostopoulou-Merkouri (St Andrews).

### 7.10.1 GraphInverseSemigroup

▷ `GraphInverseSemigroup(E)` (operation)

**Returns:** A graph inverse semigroup.

If $E$ is a digraph (i.e. it satisfies `IsDigraph` (**Digraphs: IsDigraph**)), then `GraphInverseSemigroup` returns the graph inverse semigroup $G(E)$ where, roughly speaking, elements correspond to paths in the graph $E$.

Let us describe $E$ as a digraph $E = (E^0, E^1, r, s)$, where $E^0$ is the set of vertices, $E^1$ is the set of edges, and $r$ and $s$ are functions $E^1 \to E^0$ giving the *range* and *source* of an edge, respectively. The *graph inverse semigroup* $G(E)$ *of* $E$ is the semigroup-with-zero generated by the sets $E^0$ and $E^1$, together with a set of variables $\{e^{-1} \mid e \in E^1\}$, satisfying the following relations for all $v, w \in E^0$ and $e, f \in E^1$:

**(V)** $vw = \delta_{v,w} \cdot v$,

**(E1)** $s(e) \cdot e = e \cdot r(e) = e$,

**(E2)** $r(e) \cdot e^{-1} = e^{-1} \cdot s(e) = e^{-1}$,

**(CK1)**
$e^{-1} \cdot f = \delta_{e,f} \cdot r(e)$.

(Here $\delta$ is the Kronecker delta.) We define $v^{-1} = v$ for each $v \in E^0$, and for any path $y = e_1 \dots e_n$ ($e_1 \dots e_n \in E^1$) we let $y^{-1} = e_n^{-1} \dots e_1^{-1}$. With this notation, every nonzero element of $G(E)$ can be written uniquely as $xy^{-1}$ for some paths $x, y$ in $E$, by the CK1 relation.

For a more complete description, see [MM16].

Example

```
gap> gr := Digraph([[2, 5, 8, 10], [2, 3, 4, 5, 6, 8, 9, 10], [1],
>                   [3, 5, 7, 8, 10], [2, 5, 7], [3, 6, 7, 9, 10],
>                   [1, 4], [1, 5, 9], [1, 2, 7, 8], [3, 5]]);
<immutable digraph with 10 vertices, 37 edges>
gap> S := GraphInverseSemigroup(gr);
<infinite graph inverse semigroup with 10 vertices, 37 edges>
gap> GeneratorsOfInverseSemigroup(S);
[ e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_10, e_11, e_12,
  e_13, e_14, e_15, e_16, e_17, e_18, e_19, e_20, e_21, e_22, e_23,
  e_24, e_25, e_26, e_27, e_28, e_29, e_30, e_31, e_32, e_33, e_34,
  e_35, e_36, e_37, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_10
 ]
gap> AssignGeneratorVariables(S);
gap> e_1 * e_1 ^ -1;
e_1e_1^-1
gap> e_1 ^ -1 * e_1 ^ -1;
0
gap> e_1 ^ -1 * e_1;
v_2
```

### 7.10.2 Range (for a graph inverse semigroup element)

▷ Range(x) (attribute)
▷ Source(x) (attribute)

**Returns:** A graph inverse semigroup element.

If x is an element of a graph inverse semigroup (i.e. it satisfies IsGraphInverseSemigroupElement (7.10.4)), then Range and Source give, respectively, the start and end vertices of x when viewed as a path in the digraph over which the semigroup is defined.

For a fuller description, see GraphInverseSemigroup (7.10.1).

Example

```
gap> gr := Digraph([[], [1], [3]]);;
gap> S := GraphInverseSemigroup(gr);;
gap> e := S.1;
e_1
gap> Source(e);
v_2
gap> Range(e);
v_1
```

### 7.10.3 IsVertex (for a graph inverse semigroup element)

▷ IsVertex(x) (operation)

**Returns:** true or false.

If x is an element of a graph inverse semigroup (i.e. it satisfies IsGraphInverseSemigroupElement (7.10.4)), then this attribute returns true if x corresponds to a vertex in the digraph over which the semigroup is defined, and false otherwise.

For a fuller description, see GraphInverseSemigroup (7.10.1).

Example

```
gap> gr := Digraph([[], [1], [3]]);;
gap> S := GraphInverseSemigroup(gr);;
gap> e := S.1;
e_1
gap> IsVertex(e);
false
gap> v := S.3;
v_1
gap> IsVertex(v);
true
gap> z := v * e;
0
gap> IsVertex(z);
false
```

### 7.10.4 IsGraphInverseSemigroup

▷ IsGraphInverseSemigroup(x) (filter)
▷ IsGraphInverseSemigroupElement(x) (filter)

**Returns:** true or false.

The category IsGraphInverseSemigroup contains any semigroup defined over a digraph using the GraphInverseSemigroup (7.10.1) operation. The category IsGraphInverseSemigroupElement contains any element contained in such a semigroup.

Example

```
gap> gr := Digraph([[], [1], [3]]);;
gap> S := GraphInverseSemigroup(gr);
<infinite graph inverse semigroup with 3 vertices, 2 edges>
gap> IsGraphInverseSemigroup(S);
true
gap> x := GeneratorsOfSemigroup(S)[1];
e_1
gap> IsGraphInverseSemigroupElement(x);
true
```

### 7.10.5 GraphOfGraphInverseSemigroup

▷ GraphOfGraphInverseSemigroup(*S*) (attribute)

**Returns:** A digraph.

If *S* is a graph inverse semigroup (i.e. it satisfies IsGraphInverseSemigroup (7.10.4)), then this attribute returns the original digraph over which *S* was defined (most likely the argument given to GraphInverseSemigroup (7.10.1) to create *S*).

Example

```
gap> gr := Digraph([[], [1], [3]]);
<immutable digraph with 3 vertices, 2 edges>
gap> S := GraphInverseSemigroup(gr);;
gap> GraphOfGraphInverseSemigroup(S);
<immutable digraph with 3 vertices, 2 edges>
```

### 7.10.6 IsGraphInverseSemigroupElementCollection

▷ IsGraphInverseSemigroupElementCollection (category)

Every collection of elements of a graph inverse semigroup belongs to the category IsGraphInverseSemigroupElementCollection. For example, every graph inverse semigroup belongs to IsGraphInverseSemigroupElementCollection.

### 7.10.7 IsGraphInverseSubsemigroup

▷ IsGraphInverseSubsemigroup (filter)

IsGraphInverseSubsemigroup is a synonym for IsSemigroup and IsInverseSemigroup and IsGraphInverseSemigroupElementCollection.

See IsGraphInverseSemigroupElementCollection (7.10.6) and IsInverseSemigroup (**Reference: IsInverseSemigroup**).

Example

```
gap> gr := Digraph([[], [1], [2]]);
<immutable digraph with 3 vertices, 2 edges>
gap> S := GraphInverseSemigroup(gr);
<finite graph inverse semigroup with 3 vertices, 2 edges>
```

```
gap> Elements(S);
[ e_2^-1, e_1^-1, e_1^-1e_2^-1, 0, e_1, e_1e_1^-1, e_1e_1^-1e_2^-1,
  e_2, e_2e_2^-1, e_2e_1, e_2e_1e_1^-1, e_2e_1e_1^-1e_2^-1, v_1, v_2,
  v_3 ]
gap> T := InverseSemigroup(Elements(S){[3, 5]});;
gap> IsGraphInverseSubsemigroup(T);
true
```

### 7.10.8 VerticesOfGraphInverseSemigroup

▷ `VerticesOfGraphInverseSemigroup(`*S*`)` (attribute)

**Returns:** A list.

If *S* is a graph inverse semigroup (i.e. it satisfies `IsGraphInverseSemigroup` (7.10.4)), then this attribute returns the list of vertices of *S*.

Example

```
gap> D := Digraph([[3, 4], [3, 4], [4], []]);
<immutable digraph with 4 vertices, 5 edges>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 4 vertices, 5 edges>
gap> VerticesOfGraphInverseSemigroup(S);
[ v_1, v_2, v_3, v_4 ]
gap> D := ChainDigraph(12);
<immutable chain digraph with 12 vertices>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 12 vertices, 11 edges>
gap> VerticesOfGraphInverseSemigroup(S);
[ v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_10, v_11, v_12 ]
```

### 7.10.9 IndexOfVertexOfGraphInverseSemigroup

▷ `IndexOfVertexOfGraphInverseSemigroup(`*v*`)` (attribute)

**Returns:** A positive integer.

If *v* is a vertex of a graph inverse semigroup (i.e. it satisfies `IsGraphInverseSemigroupElement` (7.10.4)), then this attribute returns the index of this vertex in its graph inverse semigroup.

Example

```
gap> D := Digraph([[3, 4], [3, 4], [4], []]);
<immutable digraph with 4 vertices, 5 edges>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 4 vertices, 5 edges>
gap> IndexOfVertexOfGraphInverseSemigroup(v_1);
1
gap> IndexOfVertexOfGraphInverseSemigroup(v_3);
3
```

### 7.10.10 LeftOne (for a graph inverse semigroup element)

▷ `LeftOne(`*x*`)` (operation)

**Returns:** A graph inverse semigroup element.

`LeftOne(`*x*`)` returns the unique idempotent `e` such that `e * `*x*` = `*x* and `e` and *x* are related by Green's $\mathscr{R}$-relation.

Example

```
gap> S := GraphInverseSemigroup(ChainDigraph(3));
<finite graph inverse semigroup with 3 vertices, 2 edges>
gap> x := S.1 * S.2;
e_1e_2
gap> LeftOne(x);
e_1e_2e_2^-1e_1^-1
gap> LeftOne(x) * x = x;
true
```

### 7.10.11 RightOne (for a graph inverse semigroup element)

▷ `RightOne(x)` (operation)

**Returns:** A graph inverse semigroup element.

`RightOne(x)` returns the unique idempotent `e` such that `x * e = x` and `e` and `x` are related by Green's $\mathscr{L}$-relation.

Example

```
gap> S := GraphInverseSemigroup(ChainDigraph(3));
<finite graph inverse semigroup with 3 vertices, 2 edges>
gap> x := S.1 * S.2;
e_1e_2
gap> RightOne(x);
v_3
gap> x * RightOne(x) = x;
true
```

## 7.11 Free inverse semigroups

This chapter describes the functions in Semigroups for dealing with free inverse semigroups. This part of the manual and the functions described herein were written by Julius Jonušas.

An inverse semigroup $F$ is said to be *free* on a non-empty set $X$ if there is a map $f$ from $F$ to $X$ such that for every inverse semigroup $S$ and a map $g$ from $X$ to $S$ there exists a unique homomorphism $g'$ from $F$ to $S$ such that $fg' = g$. Moreover, by this universal property, every inverse semigroup can be expressed as a quotient of a free inverse semigroup.

The internal representation of an element of a free inverse semigroup uses a Munn tree. A *Munn tree* is a directed tree with distinguished start and terminal vertices and where the edges are labeled by generators so that two edges labeled by the same generator are only incident to the same vertex if one of the edges is coming in and the other is leaving the vertex. For more information regarding free inverse semigroups and the Munn representations see Section 5.10 of [How95].

See also (**Reference: Inverse semigroups and monoids**), (**Reference: Partial permutations**) and (**Reference: Free Groups, Monoids and Semigroups**).

An element of a free inverse semigroup in Semigroups is displayed, by default, as a shortest word corresponding to the element. However, there might be more than one word of the minimum length. For example, if $x$ and $y$ are generators of a free inverse semigroup, then

$$xyy^{-1}xx^{-1}x^{-1} = xxx^{-1}yy^{-1}x^{-1}.$$

See `MinimalWord` (7.11.7). Therefore we provide 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 the word corresponding to the shortest path from the start vertex to the terminal vertex. 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` (7.11.6).

### 7.11.1 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. In the second and third forms, the *`names`* must be distinct. 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
```

### 7.11.2 IsFreeInverseSemigroupCategory

▷ `IsFreeInverseSemigroupCategory(`*`obj`*`)` (category)

Every free inverse semigroup in GAP created by `FreeInverseSemigroup` (7.11.1) belongs to the category `IsFreeInverseSemigroupCategory`. 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` (7.11.4).

### 7.11.3 IsFreeInverseSemigroup

▷ `IsFreeInverseSemigroup(`*`S`*`)` (property)

**Returns:** `true` or `false`

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

### 7.11.4 IsFreeInverseSemigroupElement

▷ `IsFreeInverseSemigroupElement` (Category)

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

### 7.11.5 IsFreeInverseSemigroupElementCollection

▷ `IsFreeInverseSemigroupElementCollection` (Category)

Every collection of elements of a free inverse semigroup belongs to the category `IsFreeInverseSemigroupElementCollection`. For example, every free inverse semigroup belongs to `IsFreeInverseSemigroupElementCollection`.

### 7.11.6 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"
```

### 7.11.7 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 may be 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"
```

### 7.11.8 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
```

### 7.11.9 Operators 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.

# Chapter 8

# Standard constructions

In this chapter we describe some standard ways of constructing semigroups and monoids from other semigroups that are available in the Semigroups package.

## 8.1 Products of semigroups

In this section, we describe the functions in Semigroups that can be used to create various products of semigroups.

### 8.1.1 DirectProduct

▷ `DirectProduct(`*`S[, T, ...]`*`)` (function)
▷ `DirectProductOp(`*`list, S`*`)` (operation)

**Returns:** A transformation semigroup.

The function `DirectProduct` takes an arbitrary positive number of finite semigroups, and returns a semigroup that is isomorphic to their direct product.

If these finite semigroups are all partial perm semigroups, all bipartition semigroups, or all PBR semigroups, then `DirectProduct` returns a semigroup of the same type. Otherwise, `DirectProduct` returns a transformation semigroup.

The operation `DirectProductOp` is included for consistency with the GAP library (see `DirectProductOp` (**Reference: DirectProductOp**)). It takes exactly two arguments, namely a non-empty list *`list`* of semigroups and one of these semigroups, *`S`*, and returns the same result as `CallFuncList(DirectProduct, `*`list`*`)`.

If `D` is the direct product of a collection of semigroups, then an embedding of the `i`th factor into `D` can be accessed with the command `Embedding(D, i)`, and a projection of `D` onto its `i`th factor can be accessed with the command `Projection(D, i)`; see `Embedding` (**Reference: Embedding**) and `Projection` (**Reference: Projection**) for more information.

Example

```
gap> S := InverseMonoid([PartialPerm([2, 1])]);;
gap> T := InverseMonoid([PartialPerm([1, 2, 3])]);;
gap> D := DirectProduct(S, T);
<commutative inverse partial perm monoid of rank 5 with 1 generator>
gap> Elements(D);
[ <identity partial perm on [ 1, 2, 3, 4, 5 ]>, (1,2)(3)(4)(5) ]
gap> S := PartitionMonoid(2);;
```

```
gap> D := DirectProduct(S, S, S);; IsRegularSemigroup(D);; D;
<regular bipartition monoid of size 3375, degree 6 with 9 generators>
gap> S := Semigroup([PartialPerm([2, 5, 0, 1, 3]),
>                    PartialPerm([5, 2, 4, 3])]);;
gap> T := Semigroup([Bipartition([[1, -2], [2], [3, -1, -3]])]);;
gap> D := DirectProduct(S, T);
<transformation semigroup of size 122, degree 9 with 63 generators>
gap> Size(D) = Size(S) * Size(T);
true
```

### 8.1.2 WreathProduct

▷ `WreathProduct(`*M*`, `*S*`)` (operation)

**Returns:** A transformation semigroup.

If *M* is a transformation monoid (or a permutation group) of degree `m`, and *S* is a transformation semigroup (or permutation group) of degree `s`, the operation `WreathProduct(`*M*`, `*S*`)` returns the wreath product of *M* and *S*, in terms of an embedding in the full transformation monoid of degree `m * s`.

Example

```
gap> T := FullTransformationMonoid(3);;
gap> C := Group((1, 3));;
gap> W := WreathProduct(T, C);;
gap> Size(W);
39366
gap> WW := WreathProduct(C, T);;
gap> Size(WW);
216
```

## 8.2 Dual semigroups

The *dual semigroup* of a semigroup `S` is the semigroup with the same underlying set of elements but with reversed multiplication; this is anti-isomorphic to `S`. In Semigroups a semigroup and its dual are represented with disjoint sets of elements.

### 8.2.1 DualSemigroup

▷ `DualSemigroup(`*S*`)` (attribute)

**Returns:** The dual semigroup of the given semigroup.

The dual semigroup of a semigroup *S* is the semigroup with the same underlying set as *S*, but with multiplication reversed; this is anti-isomorphic to *S*. This attribute returns a semigroup isomorphic to the dual semigroup of *S*.

Example

```
gap> S := Semigroup([Transformation([1, 4, 3, 2, 2]),
> Transformation([5, 4, 4, 1, 2])]);;
gap> D := DualSemigroup(S);
<dual semigroup of <transformation semigroup of degree 5 with 2
 generators>>
gap> Size(S) = Size(D);
```

```
true
gap> NrDClasses(S) = NrDClasses(D);
true
```

### 8.2.2 IsDualSemigroupRep

▷ IsDualSemigroupRep(*sgrp*) (category)

**Returns:** Returns true if *sgrp* lies in the category of dual semigroups.

Semigroups created using DualSemigroup (8.2.1) normally lie in this category, with the exception of semigroups which are the dual of semigroups already lying in this category. That is, a semigroup lies in the category IsDualSemigroupRep if and only if the corresponding dual semigroup does not. Note that this is not a Representation in the GAP sense, and will likely be renamed in a future major release of the package.

Example

```
gap> S := Semigroup([Transformation([3, 5, 1, 1, 2]),
> Transformation([1, 2, 4, 4, 3])]);
<transformation semigroup of degree 5 with 2 generators>
gap> D := DualSemigroup(S);
<dual semigroup of <transformation semigroup of degree 5 with 2
 generators>>
gap> IsDualSemigroupRep(D);
true
gap> R := DualSemigroup(D);
<transformation semigroup of degree 5 with 2 generators>
gap> IsDualSemigroupRep(R);
false
gap> R = S;
true
gap> T := Range(IsomorphismTransformationSemigroup(D));
<transformation semigroup of size 16, degree 17 with 2 generators>
gap> IsDualSemigroupRep(T);
false
gap> x := Representative(D);
<Transformation( [ 3, 5, 1, 1, 2 ] ) in the dual semigroup>
gap> V := Semigroup(x);
<dual semigroup of <commutative transformation semigroup of degree 5
 with 1 generator>>
gap> IsDualSemigroupRep(V);
true
```

### 8.2.3 IsDualSemigroupElement

▷ IsDualSemigroupElement(*elt*) (category)

**Returns:** Returns true if *elt* has the representation of a dual semigroup element.

Elements of a dual semigroup obtained using AntiIsomorphismDualSemigroup (8.2.4) normally lie in this category. The exception is for elements obtained by applying the map AntiIsomorphismDualSemigroup (8.2.4) to elements already in this category. That is, the elements of a semigroup lie in the category IsDualSemigroupElement if and only if the elements of the corresponding dual semigroup do not.

Example

```
gap> S := SingularPartitionMonoid(4);;
gap> D := DualSemigroup(S);;
gap> s := GeneratorsOfSemigroup(S)[1];;
gap> map := AntiIsomorphismDualSemigroup(S);;
gap> t := s ^ map;
<<block bijection: [ 1, 2, -1, -2 ], [ 3, -3 ], [ 4, -4 ]>
  in the dual semigroup>
gap> IsDualSemigroupElement(t);
true
gap> inv := InverseGeneralMapping(map);;
gap> x := t ^ inv;
<block bijection: [ 1, 2, -1, -2 ], [ 3, -3 ], [ 4, -4 ]>
gap> IsDualSemigroupElement(x);
false
```

### 8.2.4 AntiIsomorphismDualSemigroup

▷ AntiIsomorphismDualSemigroup(*S*) (attribute)

**Returns:** An anti-isomorphism from *S* to the corresponding dual semigroup.

The dual semigroup of *S* mathematically has the same underlying set as *S*, but is represented with a different set of elements in Semigroups . This function returns a mapping which is an anti-isomorphism from *S* to its dual.

Example

```
gap> S := PartitionMonoid(3);
<regular bipartition *-monoid of size 203, degree 3 with 4 generators>
gap> map := AntiIsomorphismDualSemigroup(S);
MappingByFunction( <regular bipartition *-monoid of size 203,
 degree 3 with 4 generators>, <dual semigroup of
<regular bipartition *-monoid of size 203, degree 3 with 4 generators>
 >, function( x ) ... end, function( x ) ... end )
gap> inv := InverseGeneralMapping(map);;
gap> x := Bipartition([[1, -2], [2, -3], [3, -1]]);
<block bijection: [ 1, -2 ], [ 2, -3 ], [ 3, -1 ]>
gap> y := Bipartition([[1], [2, -2], [3, -3], [-1]]);
<bipartition: [ 1 ], [ 2, -2 ], [ 3, -3 ], [ -1 ]>
gap> (x ^ map) * (y ^ map) = (y * x) ^ map;
true
gap> x ^ map;
<<block bijection: [ 1, -2 ], [ 2, -3 ], [ 3, -1 ]>
  in the dual semigroup>
```

## 8.3 Strong semilattices of semigroups

In this section, we describe how Semigroups can be used to create and manipulate strong semilattices of semigroups (SSSs). Strong semilattices of semigroups are described, for example, in Section 4.1 of [How95]. They consist of a meet-semilattice $Y$ along with a collection of semigroups $S_a$ for each $a$ in $Y$, and a collection of homomorphisms $f_{ab} : S_a \to S_b$ for each $a$ and $b$ in $Y$ such that $a \geq b$.

The product of two elements $x \in S_a, y \in S_b$ is defined to lie in the semigroup $S_c$, corresponding to the meet $c$ of $a, b \in Y$. The exact element of $S_c$ equal to the product is obtained using the homomorphisms of the SSS: $xy = (xf_{ac})(yf_{bc})$.

### 8.3.1 StrongSemilatticeOfSemigroups

▷ `StrongSemilatticeOfSemigroups(D, L, H)` (operation)

**Returns:** A strong semilattice of semigroups.

If `D` is a digraph, `L` is a list of semigroups, and `H` is a list of lists of maps, then this function returns a corresponding `IsStrongSemilatticeOfSemigroups` object. The format of the arguments is not required to be exactly analogous to Howie's description above, but consistency amongst the arguments is required:

- `D` must be a digraph whose `DigraphReflexiveTransitiveClosure` (**Digraphs: DigraphReflexiveTransitiveClosure**) is a meet-semilattice. For example, `Digraph([2, 3], [4], [4], [])` is valid and produces a semilattice where the meet of 2 and 3 is 1. See `IsMeetSemilatticeDigraph` (**Digraphs: IsMeetSemilatticeDigraph**).
- `L` must contain as many semigroups as there are vertices in `D`.
- `H` must be a list with as many elements as there are vertices in `D`. Each element of `H` must itself be a (possibly empty) list with as many entries as the corresponding vertex of `D` has out-edges. The entries of each sublist must be the corresponding homomorphisms: for example, if `D` is entered as above, then `H[1][2]` must be the homomorphism $f_3 1$, i.e. `H[1][2]` is an `IsMapping` object whose domain is a superset of `L[3]` and whose range is a subset of `L[1]`.

Note that in the example above, the edge $1 \to 4$ is not entered as part of the argument `D`, but it is still an edge in the reflexive transitive closure of `D`. When creating the object, GAP creates the homomorphism $f_{41}$ by composing the mappings along paths that lead from 4 to 1, and checks that composing along all possible paths produces the same result.

### 8.3.2 SSSE

▷ `SSSE(SSS, n, x)` (operation)

**Returns:** An element of a strong semilattice of semigroups.

If `n` is a vertex of the underlying semilattice of the strong semilattice of semigroups `SSS`, and if `x` is an element of the `n`th semigroup of `SSS`, then this function returns the element of `SSS` which lies in semigroup number `n` and which corresponds to the element `x` in that semigroup.

This function returns an `IsSSSE` (8.3.3) object. SSSEs from the same strong semilattice of semigroups can be compared and multiplied.

Example

```
gap> D := Digraph([[2, 3], [4], [4], []]);;
gap> S4 := FullTransformationMonoid(2);;
gap> S3 := FullTransformationMonoid(3);;
gap> pairs := [[Transformation([1, 2]), Transformation([2, 1])]];;
gap> cong := SemigroupCongruence(S4, pairs);;
gap> S2 := S4 / cong;;
gap> S1 := TrivialSemigroup();;
gap> L := [S1, S2, S3, S4];;
gap> idfn := t -> IdentityTransformation;;
```

```
gap> f21 := SemigroupHomomorphismByFunction(S2, S1, idfn);;
gap> f31 := SemigroupHomomorphismByFunction(S3, S1, idfn);;
gap> f42 := HomomorphismQuotientSemigroup(cong);;
gap> f43 := SemigroupHomomorphismByFunction(S4, S3, IdFunc);;
gap> H := [[f21, f31], [f42], [f43], []];;
gap> SSS := StrongSemilatticeOfSemigroups(D, L, H);
<strong semilattice of 4 semigroups>
gap> Size(SSS);
34
gap> x := SSSE(SSS, 3, Elements(S3)[10]);
SSSE(3, Transformation( [ 2, 1, 1 ] ))
gap> y := SSSE(SSS, 4, Elements(S4)[1]);
SSSE(4, Transformation( [ 1, 1 ] ))
gap> x * y;
SSSE(3, Transformation( [ 1, 1, 1 ] ))
```

### 8.3.3 IsSSSE

▷ `IsSSSE(`*obj*`)` (filter)

**Returns:** `true` or `false`.

All elements of an SSS belong in the category `IsSSSE` (for "Strong Semilattice of Semigroups Element").

### 8.3.4 IsStrongSemilatticeOfSemigroups

▷ `IsStrongSemilatticeOfSemigroups(`*obj*`)` (filter)

**Returns:** `true` or `false`.

Every Strong Semilattice of Semigroups in GAP belongs to the category `IsStrongSemilatticeOfSemigroups`. Basic operations in this category allow the user to recover the three essential elements of an SSS object: `SemilatticeOfStrongSemilatticeOfSemigroups` (8.3.5), `SemigroupsOfStrongSemilatticeOfSemigroups` (8.3.6), and `HomomorphismsOfStrongSemilatticeOfSemigroups` (8.3.7).

### 8.3.5 SemilatticeOfStrongSemilatticeOfSemigroups

▷ `SemilatticeOfStrongSemilatticeOfSemigroups(`*SSS*`)` (attribute)

**Returns:** A meet-semilattice digraph.

If *SSS* is a strong semilattice of semigroups, this function returns the underlying semilattice structure as a digraph. Note that this may not be equal to the digraph passed as input when *SSS* was created: rather, it is the reflexive transitive closure of the input digraph.

### 8.3.6 SemigroupsOfStrongSemilatticeOfSemigroups

▷ `SemigroupsOfStrongSemilatticeOfSemigroups(`*SSS*`)` (attribute)

**Returns:** A list of semigroups.

If *SSS* is a strong semilattice of semigroups, this function returns the list of semigroups that make up *SSS*. The position of a semigroup in the list corresponds to the node of the semilattice where that semigroup lies.

### 8.3.7 HomomorphismsOfStrongSemilatticeOfSemigroups

▷ `HomomorphismsOfStrongSemilatticeOfSemigroups(`*SSS*`)` (attribute)

**Returns:** A list of lists of mappings.

If *SSS* is a strong semilattice of $n$ semigroups, this function returns an $n \times n$ list where the $(i, j)$th entry of the list is the homomorphism $f_{ji}$, provided $i \leq j$ in the semilattice. If this last condition is not true, then the entry is `fail`.

### 8.3.8 UnderlyingSemilatticeOfSemigroups

▷ `UnderlyingSemilatticeOfSemigroups(`*el*`)` (operation)

**Returns:** A strong semilattice of semigroups.

If *el* is an element of a strong semilattice of semigroups (i.e. is in the category `IsSSSE` (8.3.3)), then this operation returns the strong semilattice of semigroups object that *el* belongs to.

## 8.4 McAlister triple semigroups

In this section, we describe the functions in GAP for creating and computing with McAlister triple semigroups and their subsemigroups. This implementation is based on the section in Chapter 5 of [How95] but differs from the treatment in Howie by using right actions instead of left. Some definitions found in the documentation are changed for this reason.

The importance of the McAlister triple semigroups lies in the fact that they are exactly the E-unitary inverse semigroups, which are an important class in the study of inverse semigroups.

First we define E-unitary inverse semigroups. It is standard to denote the subsemigroup of a semigroup consisting of its idempotents by `E`. A semigroup `S` is said to be *E-unitary* if for all `e` in `E` and for all `s` in `S`:

- `es` $\in$ `E` implies `s` $\in$ `E`,
- `se` $\in$ `E` implies `s` $\in$ `E`.

For inverse semigroups these two conditions are equivalent. We are only interested in *E-unitary inverse semigroups*. Before defining McAlister triple semigroups we define a McAlister triple. A *McAlister triple* is a triple `(G,X,Y)` which consists of:

- a partial order `X`,
- a subset `Y` of `X`,
- a group `G` which acts on `X`, on the right, by order automorphisms. That means for all `A,B` $\in$ `X` and for all `g` $\in$ `G`: `A` $\leq$ `B` if and only if `Ag` $\leq$ `Bg`.

Furthermore, `(G,X,Y)` must satisfy the following four properties to be a McAlister triple:

**M1** `Y` is a subset of `X` which is a join-semilattice together with the restriction of the order relation of `X` to `Y`.

**M2** `Y` is an order ideal of `X`. That is to say, for all `A` $\in$ `X` and for all `B` $\in$ `Y`: if `A` $\leq$ `B`, then `A` $\in$ `Y`.

**M3** Every element of `X` is the image of some element in `Y` moved by an element of `G`. That is to say, for every `A` $\in$ `X`, there exists some `B` $\in$ `Y` and there exists `g` $\in$ `G` such that `A` = `Bg`.

**M4** Finally, for all g ∈ G, the intersection {yg : y ∈ Y} ∩ Y is non-empty.

We may define an E-unitary inverse semigroup using a McAlister triple. Given (G,X,Y) let M(G,X,Y) be the set of all pairs (A,g) in Y x G such that A acted on by the inverse of g is in Y together with multiplication defined by

(A,g)*(B,h) = (Join(A,Bg^-1),hg)

where Join is the natural join operation of the semilattice and Bg^-1 is B acted on by the inverse of g. With this operation, M(G,X,Y) is a semigroup which we call a *McAlister triple semigroup* over (G,X,Y). In fact every McAlister triple semigroup is an E-unitary inverse semigroup and every E-unitary inverse semigroup is isomorphic to some McAlister triple semigroup. Note that there need not be a unique McAlister triple semigroup for a particular McAlister triple because in general there is more than one way for a group to act on a partial order.

### 8.4.1 IsMcAlisterTripleSemigroup

▷ IsMcAlisterTripleSemigroup(*S*) (filter)

**Returns:** true or false.

This function returns true if *S* is a McAlister triple semigroup. A *McAlister triple semigroup* is a special representation of an E-unitary inverse semigroup IsEUnitaryInverseSemigroup (12.2.3) created by McAlisterTripleSemigroup (8.4.2).

### 8.4.2 McAlisterTripleSemigroup

▷ McAlisterTripleSemigroup(*G, X, Y[, act]*) (operation)

**Returns:** A McAlister triple semigroup.

The following documentation covers the technical information needed to create McAlister triple semigroups in GAP, the underlying theory can be read in the introduction to Chapter 8.4.

In this implementation the partial order X of a McAlister triple is represented by a Digraph (**Digraphs: Digraph**) object *X*. The digraph represents a partial order in the sense that vertices are the elements of the partial order and the order relation is defined by A ≤ B if and only if there is an edge from B to A. The semilattice Y of the McAlister triple should be an induced subdigraph *Y* of *X* and the DigraphVertexLabels (**Digraphs: DigraphVertexLabels**) must correspond to the vertices of *X* on which *Y* is induced. That means that the following:

*Y* = InducedSubdigraph(*X*, DigraphVertexLabels(*Y*))

must return true. Herein if we say that a vertex A of *X* is 'in' *Y* then we mean there is a vertex of *Y* whose label is A. Alternatively the user may choose to give the argument *Y* as the vertices of *X* on which *Y* is the induced subdigraph.

A McAlister triple semigroup is created from a quadruple (*G*, *X*, *Y*, *act*) where:

- *G* is a finite group.
- *X* is a digraph satisfying IsPartialOrderDigraph (**Digraphs: IsPartialOrderDigraph**).
- *Y* is a digraph satisfying IsJoinSemilatticeDigraph (**Digraphs: IsJoinSemilatticeDigraph**) which is an induced subdigraph of *X* satisfying the aforementioned labeling criteria. Furthermore the OutNeighbours (**Digraphs: OutNeighbours**) of each vertex of *X* which is in *Y* must contain only vertices which are in *Y*.

- *act* is a function which takes as its first argument a vertex of the digraph *X*, its second argument should be an element of *G*, and it must return a vertex of *X*. *act* must be a right action, meaning that act(A,gh)=act(act(A,g),h) holds for all A in *X* and g,h ∈ *G*. Furthermore the permutation representation of this action must be a subgroup of the automorphism group of *X*. That means we require the following to return true:

  IsSubgroup(AutomorphismGroup(*X*), Image(ActionHomomorphism(*G*, DigraphVertices(*X*), *act*));

  Furthermore every vertex of *X* must be in the orbit of some vertex of *X* which is in *Y*. Finally, *act* must fix the vertex of *X* which is the minimal vertex of *Y*, i.e. the unique vertex of *Y* whose only out-neighbour is itself.

For user convenience, there are multiple versions of McAlisterTripleSemigroup. When the argument *act* is omitted it is assumed to be OnPoints (**Reference: OnPoints**). Additionally, the semilattice argument *Y* may be replaced by a homogeneous list *sub_ver* of vertices of *X*. When *sub_ver* is provided, McAlisterTripleSemigroup is called with *Y* equalling InducedSubdigraph(*X*, *sub_ver*) with the appropriate labels.

Example

```
gap> x := Digraph([[1], [1, 2], [1, 2, 3], [1, 4], [1, 4, 5]]);
<immutable digraph with 5 vertices, 11 edges>
gap> y := InducedSubdigraph(x, [1, 4, 5]);
<immutable digraph with 3 vertices, 6 edges>
gap> DigraphVertexLabels(y);
[ 1, 4, 5 ]
gap> A := AutomorphismGroup(x);
Group([ (2,4)(3,5) ])
gap> S := McAlisterTripleSemigroup(A, x, y, OnPoints);
<McAlister triple semigroup over Group([ (2,4)(3,5) ])>
gap> T := McAlisterTripleSemigroup(A, x, y);
<McAlister triple semigroup over Group([ (2,4)(3,5) ])>
gap> S = T;
false
gap> IsIsomorphicSemigroup(S, T);
true
```

### 8.4.3 McAlisterTripleSemigroupGroup

▷ McAlisterTripleSemigroupGroup(*S*) (attribute)

**Returns:** A group.

Returns the group used to create the McAlister triple semigroup *S* via McAlisterTripleSemigroup (8.4.2).

### 8.4.4 McAlisterTripleSemigroupPartialOrder

▷ McAlisterTripleSemigroupPartialOrder(*S*) (attribute)

**Returns:** A partial order digraph.

Returns the IsPartialOrderDigraph (**Digraphs: IsPartialOrderDigraph**) used to create the McAlister triple semigroup *S* via McAlisterTripleSemigroup (8.4.2).

### 8.4.5 McAlisterTripleSemigroupSemilattice

▷ McAlisterTripleSemigroupSemilattice(*S*) (attribute)

**Returns:** A join-semilattice digraph.

Returns the IsJoinSemilatticeDigraph (**Digraphs: IsJoinSemilatticeDigraph**) used to create the McAlister triple semigroup *S* via McAlisterTripleSemigroup (8.4.2).

### 8.4.6 McAlisterTripleSemigroupAction

▷ McAlisterTripleSemigroupAction(*S*) (attribute)

**Returns:** A function.

Returns the action used to create the McAlister triple semigroup *S* via McAlisterTripleSemigroup (8.4.2).

### 8.4.7 IsMcAlisterTripleSemigroupElement

▷ IsMcAlisterTripleSemigroupElement(*x*) (filter)

▷ IsMTSE(x) (filter)

**Returns:** true or false.

Returns true if *x* is an element of a McAlister triple semigroup; in particular, this returns true if *x* has been created by McAlisterTripleSemigroupElement (8.4.8). The functions IsMTSE and IsMcAlisterTripleSemigroupElement are synonyms. The mathematical description of these objects can be found in the introduction to Chapter 8.4.

### 8.4.8 McAlisterTripleSemigroupElement

▷ McAlisterTripleSemigroupElement(*S*, *A*, *g*) (operation)

▷ MTSE(*S*, *A*, *g*) (operation)

**Returns:** A McAlister triple semigroup element.

Returns the *McAlister triple semigroup element* of the McAlister triple semigroup *S* which corresponds to a label *A* of a vertex from the McAlisterTripleSemigroupSemilattice (8.4.5) of *S* and a group element *g* of the McAlisterTripleSemigroupGroup (8.4.3) of *S*. The pair *(A,g)* only represents an element of *S* if the following holds: *A* acted on by the inverse of *g* (via McAlisterTripleSemigroupAction (8.4.6)) is a vertex of the join-semilattice of *S*.

The functions MTSE and McAlisterTripleSemigroupElement are synonyms.

Example

```
gap> x := Digraph([[1], [1, 2], [1, 2, 3], [1, 4], [1, 4, 5]]);
<immutable digraph with 5 vertices, 11 edges>
gap> y := InducedSubdigraph(x, [1, 2, 3]);
<immutable digraph with 3 vertices, 6 edges>
gap> A := AutomorphismGroup(x);
Group([ (2,4)(3,5) ])
gap> S := McAlisterTripleSemigroup(A, x, y, OnPoints);
<McAlister triple semigroup over Group([ (2,4)(3,5) ])>
gap> T := McAlisterTripleSemigroup(A, x, y);
<McAlister triple semigroup over Group([ (2,4)(3,5) ])>
gap> S = T;
false
gap> IsIsomorphicSemigroup(S, T);
true
```

```
gap> a := MTSE(S, 1, (2, 4)(3, 5));
(1, (2,4)(3,5))
gap> b := MTSE(S, 2, ());
(2, ())
gap> a * a;
(1, ())
gap> IsMTSE(a * a);
true
gap> a = MTSE(T, 1, (2, 4)(3, 5));
false
gap> a * b;
(1, (2,4)(3,5))
```

# Chapter 9

# Ideals

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

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

The methods in the Semigroups package apply to any ideal of a semigroup that is created using the function `SemigroupIdeal` (9.1.1) 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.

## 9.1 Creating ideals

### 9.1.1 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>
```

### 9.1.2 Ideals (for a semigroup)

▷ Ideals(*S*) (attribute)

**Returns:** A list of ideals.

If *S* is a finite non-empty semigroup, then this attribute returns a list of the non-empty two-sided ideals of *S*.

The ideals are returned in no particular order, and each ideal uses the minimum possible number of generators (see GeneratorsOfSemigroupIdeal (9.2.1)).

Example

```
gap> S := Semigroup([Transformation([4, 3, 4, 1]),
>                    Transformation([4, 3, 2, 2])]);
<transformation semigroup of degree 4 with 2 generators>
gap> Ideals(S);
[ <non-regular transformation semigroup ideal of degree 4 with
      1 generator>,
 <non-regular transformation semigroup ideal of degree 4 with
     1 generator>,
<non-regular transformation semigroup ideal of degree 4 with
    2 generators>,
<regular transformation semigroup ideal of degree 4 with 1 generator>,
<non-regular transformation semigroup ideal of degree 4 with
  1 generator>,
<regular transformation semigroup ideal of degree 4 with 1 generator>
]
```

## 9.2 Attributes of ideals

### 9.2.1 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 define *I* when it was created.

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

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

Example

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

```
gap> I = Semigroup(GeneratorsOfSemigroupIdeal(I));
false
```

### 9.2.2 MinimalIdealGeneratingSet

▷ `MinimalIdealGeneratingSet(I)` (attribute)

**Returns:** A minimal set of ideal generators of an ideal.

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

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

Example

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

### 9.2.3 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` (9.1.1) or `SemigroupIdealByGenerators`. This function returns the semigroup containing the generators of the semigroup (i.e. `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**)) which are used to compute the ideal.

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

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

Example

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

```
gap> I := SemigroupIdeal(S, MultiplicativeZero(S));
<regular Rees 0-matrix semigroup ideal with 1 generator>
gap> SupersemigroupOfIdeal(I);
<subsemigroup of 1050x56 Rees 0-matrix semigroup with 10 generators>
gap> J := SemigroupIdeal(I, Representative(MinimalDClass(S)));
<regular Rees 0-matrix semigroup ideal with 1 generator>
gap> Parent(J) = I;
true
gap> SupersemigroupOfIdeal(J) = I;
false
```

# Chapter 10

# Green's relations

In this chapter we describe the functions in Semigroups for computing Green's classes and related properties of semigroups.

## 10.1 Creating Green's classes and representatives

In this section, we describe the methods in the Semigroups package for creating Green's classes.

### 10.1.1 XClassOfYClass

▷ `DClassOfHClass(class)` (method)
▷ `DClassOfLClass(class)` (method)
▷ `DClassOfRClass(class)` (method)
▷ `LClassOfHClass(class)` (method)
▷ `RClassOfHClass(class)` (method)

**Returns:** A Green's class.

`XClassOfYClass` returns the `X`-class containing the `Y`-class *`class`* where `X` and `Y` should be replaced by an appropriate choice of `D`, `H`, `L`, and `R`.

Note that if it is not known to GAP whether or not the representative of *`class`* is an element of the semigroup containing *`class`*, then no attempt is made to check this.

The same result can be produced using:

Example

```
First(GreensXClasses(S), x -> Representative(x) in class);
```

but this might be substantially slower. Note that `XClassOfYClass` is also likely to be faster than

Example

```
GreensXClassOfElement(S, Representative(class));
```

`DClass` can also be used as a synonym for `DClassOfHClass`, `DClassOfLClass`, and `DClassOfRClass`; `LClass` as a synonym for `LClassOfHClass`; and `RClass` as a synonym for `RClassOfHClass`. See also `GreensDClassOfElement` (**Reference: GreensDClassOfElement**) and `GreensDClassOfElementNC` (10.1.3).

Example

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

```
>                   Transformation([3, 2, 1]),
>                   Transformation([1, 3, 1]));;
gap> R := GreensRClassOfElement(S, Transformation([3, 2, 1]));
<Green's R-class: Transformation( [ 3, 2, 1 ] )>
gap> DClassOfRClass(R);
<Green's D-class: Transformation( [ 3, 2, 1 ] )>
gap> IsGreensDClass(DClassOfRClass(R));
true
gap> S := InverseSemigroup(
> PartialPerm([2, 6, 7, 0, 0, 9, 0, 1, 0, 5]),
> PartialPerm([3, 8, 1, 9, 0, 4, 10, 5, 0, 6]));
<inverse partial perm semigroup of rank 10 with 2 generators>
gap> x := S.1;
[3,7][8,1,2,6,9][10,5]
gap> H := HClass(S, x);
<Green's H-class: [3,7][8,1,2,6,9][10,5]>
gap> R := RClassOfHClass(H);
<Green's R-class: [3,7][8,1,2,6,9][10,5]>
gap> L := LClass(H);;
gap> L = LClass(S, PartialPerm([1, 2, 0, 0, 5, 6, 7, 0, 9]));
true
gap> DClass(R) = DClass(L);
true
gap> DClass(H) = DClass(L);
true
```

### 10.1.2 GreensXClassOfElement

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

**Returns:** A Green's class.

These functions produce essentially the same output as the GAP library functions with the same names; see GreensDClassOfElement (**Reference: GreensDClassOfElement**). The main difference is that these functions can be applied to a wider class of objects:

GreensDClassOfElement **and** DClass

*X* must be a semigroup.

GreensHClassOfElement **and** HClass

*X* can be a semigroup, $\mathscr{R}$-class, $\mathscr{L}$-class, or $\mathscr{D}$-class. If *R* is a *IxJ* Rees matrix semigroup or a Rees 0-matrix semigroup, and *i* and *j* are integers of the corresponding index sets, then GreensHClassOfElement returns the $\mathscr{H}$-class in row *i* and column *j*.

`GreensLClassOfElement` **and** `LClass`
: *X* can be a semigroup or $\mathscr{D}$-class.

`GreensRClassOfElement` **and** `RClass`
: *X* can be a semigroup or $\mathscr{D}$-class.

Note that `GreensXClassOfElement` and `XClass` are synonyms and have identical output. The shorter command is provided for the sake of convenience.

### 10.1.3 GreensXClassOfElementNC

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

**Returns:** A Green's class.

These functions are essentially the same as `GreensDClassOfElement` (10.1.2) except that no effort is made to verify if *f* is an element of *X*. More precisely, `GreensXClassOfElementNC` and `XClassNC` first check if *f* has already been shown to be an element of *X*. If it is not known to GAP if *f* is an element of *X*, then no further attempt to verify this is made.

Note that `GreensXClassOfElementNC` and `XClassNC` are synonyms and have identical output. The shorter command is provided for the sake of convenience.

It can be quicker to compute the class of an element using `GreensRClassOfElementNC`, say, than using `GreensRClassOfElement` if it is known *a priori* that *f* is an element of *X*. On the other hand, if *f* is not an element of *X*, then the results of this computation are unpredictable.

For example, if

Example
```
x := Transformation([15, 18, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
```

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

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

returns an answer relatively quickly, whereas

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

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

See also `GreensRClassOfElement` (**Reference: GreensRClassOfElement**) and `RClassOfHClass` (10.1.1).

Example
```
gap> S := RandomSemigroup(IsTransformationSemigroup, 2, 1000);;
gap> x := [1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 1, 2, 2, 1];;
gap> x := EvaluateWord(Generators(S), x);;
gap> R := GreensRClassOfElementNC(S, x);;
```

```
gap> Size(R);
1
gap> L := GreensLClassOfElementNC(S, x);;
gap> Size(L);
1
gap> x := PartialPerm([2, 3, 4, 5, 0, 0, 6, 8, 10, 11]);;
gap> L := LClass(POI(11), x);
<Green's L-class: [1,2,3,4,5][7,6][9,10,11](8)>
gap> Size(L);
165
```

### 10.1.4 GreensXClasses

▷ GreensDClasses(*obj*) (method)
▷ DClasses(*obj*) (method)
▷ GreensHClasses(*obj*) (method)
▷ HClasses(*obj*) (method)
▷ GreensJClasses(*obj*) (method)
▷ JClasses(*obj*) (method)
▷ GreensLClasses(*obj*) (method)
▷ LClasses(*obj*) (method)
▷ GreensRClasses(*obj*) (method)
▷ RClasses(*obj*) (method)

**Returns:** A list of Green's classes.

These functions produce essentially the same output as the GAP library functions with the same names; see GreensDClasses (**Reference: GreensDClasses**). The main difference is that these functions can be applied to a wider class of objects:

GreensDClasses **and** DClasses
*X* should be a semigroup.

GreensHClasses **and** HClasses
*X* can be a semigroup, $\mathscr{R}$-class, $\mathscr{L}$-class, or $\mathscr{D}$-class.

GreensLClasses **and** LClasses
*X* can be a semigroup or $\mathscr{D}$-class.

GreensRClasses **and** RClasses
*X* can be a semigroup or $\mathscr{D}$-class.

Note that GreensXClasses and XClasses are synonyms and have identical output. The shorter command is provided for the sake of convenience.

See also DClassReps (10.1.5), IteratorOfDClassReps (10.2.1), IteratorOfDClasses (10.2.2), and NrDClasses (10.1.9).

Example

```
gap> S := Semigroup(Transformation([3, 4, 4, 4]),
>                   Transformation([4, 3, 1, 2]));;
gap> GreensDClasses(S);
[ <Green's D-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's D-class: Transformation( [ 4, 3, 1, 2 ] )>,
```

```
  <Green's D-class: Transformation( [ 4, 4, 4, 4 ] )> ]
gap> GreensRClasses(S);
[ <Green's R-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 3, 1, 2 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 3, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 3, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 4, 3 ] )> ]
gap> D := GreensDClasses(S)[1];
<Green's D-class: Transformation( [ 3, 4, 4, 4 ] )>
gap> GreensLClasses(D);
[ <Green's L-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's L-class: Transformation( [ 1, 2, 2, 2 ] )> ]
gap> GreensRClasses(D);
[ <Green's R-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 3, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 3, 4, 4 ] )>,
  <Green's R-class: Transformation( [ 4, 4, 4, 3 ] )> ]
gap> R := GreensRClasses(D)[1];
<Green's R-class: Transformation( [ 3, 4, 4, 4 ] )>
gap> GreensHClasses(R);
[ <Green's H-class: Transformation( [ 3, 4, 4, 4 ] )>,
  <Green's H-class: Transformation( [ 1, 2, 2, 2 ] )> ]
gap> S := InverseSemigroup([
> PartialPerm([2, 4, 1]), PartialPerm([3, 0, 4, 1])]);;
gap> GreensDClasses(S);
[ <Green's D-class: <identity partial perm on [ 1, 2, 4 ]>>,
  <Green's D-class: <identity partial perm on [ 1, 3, 4 ]>>,
  <Green's D-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's D-class: <identity partial perm on [ 4 ]>>,
  <Green's D-class: <empty partial perm>> ]
gap> GreensLClasses(S);
[ <Green's L-class: <identity partial perm on [ 1, 2, 4 ]>>,
  <Green's L-class: [4,2,1,3]>,
  <Green's L-class: <identity partial perm on [ 1, 3, 4 ]>>,
  <Green's L-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's L-class: [3,1,2]>, <Green's L-class: [1,4][3,2]>,
  <Green's L-class: [1,3,4]>, <Green's L-class: [3,1,4]>,
  <Green's L-class: [1,2](3)>,
  <Green's L-class: <identity partial perm on [ 4 ]>>,
  <Green's L-class: [4,1]>, <Green's L-class: [4,3]>,
  <Green's L-class: [4,2]>, <Green's L-class: <empty partial perm>> ]
gap> D := GreensDClasses(S)[3];
<Green's D-class: <identity partial perm on [ 1, 3 ]>>
gap> GreensLClasses(D);
[ <Green's L-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's L-class: [3,1,2]>, <Green's L-class: [1,4][3,2]>,
  <Green's L-class: [1,3,4]>, <Green's L-class: [3,1,4]>,
  <Green's L-class: [1,2](3)> ]
gap> GreensRClasses(D);
[ <Green's R-class: <identity partial perm on [ 1, 3 ]>>,
  <Green's R-class: [2,1,3]>, <Green's R-class: [2,3][4,1]>,
```

```
    <Green's R-class: [4,3,1]>, <Green's R-class: [4,1,3]>,
    <Green's R-class: [2,1](3)> ]
```

### 10.1.5 XClassReps

▷ DClassReps(*obj*) (attribute)
▷ HClassReps(*obj*) (attribute)
▷ LClassReps(*obj*) (attribute)
▷ RClassReps(*obj*) (attribute)

**Returns:** A list of representatives.

XClassReps returns a list of the representatives of the Green's classes of *obj*, which can be a semigroup, $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{R}$-class where appropriate.

The same output can be obtained by calling, for example:

Example

```
List(GreensXClasses(obj), Representative);
```

Note that if the Green's classes themselves are not required, then XClassReps will return an answer more quickly than the above, since the Green's class objects are not created.

See also GreensDClasses (10.1.4), IteratorOfDClassReps (10.2.1), IteratorOfDClasses (10.2.2), and NrDClasses (10.1.9).

Example

```
gap> S := Semigroup(Transformation([3, 4, 4, 4]),
>                   Transformation([4, 3, 1, 2]));;
gap> DClassReps(S);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 4, 3, 1, 2 ] ),
  Transformation( [ 4, 4, 4, 4 ] ) ]
gap> LClassReps(S);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 1, 2, 2, 2 ] ),
  Transformation( [ 4, 3, 1, 2 ] ), Transformation( [ 4, 4, 4, 4 ] ),
  Transformation( [ 2, 2, 2, 2 ] ), Transformation( [ 3, 3, 3, 3 ] ),
  Transformation( [ 1, 1, 1, 1 ] ) ]
gap> D := GreensDClasses(S)[1];
<Green's D-class: Transformation( [ 3, 4, 4, 4 ] )>
gap> LClassReps(D);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 1, 2, 2, 2 ] ) ]
gap> RClassReps(D);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 4, 4, 3, 4 ] ),
  Transformation( [ 4, 3, 4, 4 ] ), Transformation( [ 4, 4, 4, 3 ] ) ]
gap> R := GreensRClasses(D)[1];;
gap> HClassReps(R);
[ Transformation( [ 3, 4, 4, 4 ] ), Transformation( [ 1, 2, 2, 2 ] ) ]
gap> S := SymmetricInverseSemigroup(6);;
gap> e := InverseSemigroup(Idempotents(S));;
gap> M := MunnSemigroup(e);;
gap> L := LClassNC(M, PartialPerm([51, 63], [51, 47]));;
gap> HClassReps(L);
[ <identity partial perm on [ 47, 51 ]>, [27,47](51), [50,47](51),
  [64,47](51), [63,47](51), [59,47](51) ]
```

### 10.1.6 MinimalDClass

▷ `MinimalDClass(`$S$`)` (attribute)

**Returns:** The minimal $\mathscr{D}$-class of a semigroup.

The minimal ideal of a semigroup is the least ideal with respect to containment. `MinimalDClass` returns the $\mathscr{D}$-class corresponding to the minimal ideal of the semigroup $S$. Equivalently, `MinimalDClass` returns the minimal $\mathscr{D}$-class with respect to the partial order of $\mathscr{D}$-classes.

It is significantly easier to find the minimal $\mathscr{D}$-class of a semigroup, than to find its $\mathscr{D}$-classes.

See also `PartialOrderOfDClasses` (10.1.10), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), `MinimalIdeal` (11.8.1) and `RepresentativeOfMinimalIdeal` (11.8.2).

Example

```
gap> D := MinimalDClass(JonesMonoid(8));
<Green's D-class: <bipartition: [ 1, 2 ], [ 3, 4 ], [ 5, 6 ],
  [ 7, 8 ], [ -1, -2 ], [ -3, -4 ], [ -5, -6 ], [ -7, -8 ]>>
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 5, 7, 8, 9], [2, 6, 9, 1, 5, 3, 8]),
> PartialPerm([1, 3, 4, 5, 7, 8, 9], [9, 4, 10, 5, 6, 7, 1]));;
gap> MinimalDClass(S);
<Green's D-class: <empty partial perm>>
```

### 10.1.7 MaximalXClasses

▷ `MaximalDClasses(`$S$`)` (attribute)

▷ `MaximalLClasses(`$S$`)` (attribute)

▷ `MaximalRClasses(`$S$`)` (attribute)

**Returns:** The maximal $\mathscr{D}$, $\mathscr{L}$, or $\mathscr{R}$-classes of a semigroup.

Let `X` be one of Green's $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{R}$-relations. Then `MaximalXClasses` returns the maximal Green's `X`-classes with respect to the partial order of `X`-classes.

See also `PartialOrderOfDClasses` (10.1.10), `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**), and `MinimalDClass` (10.1.6).

Example

```
gap> MaximalDClasses(BrauerMonoid(8));
[ <Green's D-class: <block bijection: [ 1, -1 ], [ 2, -2 ],
      [ 3, -3 ], [ 4, -4 ], [ 5, -5 ], [ 6, -6 ], [ 7, -7 ],
      [ 8, -8 ]>> ]
gap> MaximalDClasses(FullTransformationMonoid(5));
[ <Green's D-class: IdentityTransformation> ]
gap> S := Semigroup(
> PartialPerm([1, 2, 3, 4, 5, 6, 7], [3, 8, 1, 4, 5, 6, 7]),
> PartialPerm([1, 2, 3, 6, 8], [2, 6, 7, 1, 5]),
> PartialPerm([1, 2, 3, 4, 6, 8], [4, 3, 2, 7, 6, 5]),
> PartialPerm([1, 2, 4, 5, 6, 7, 8], [7, 1, 4, 2, 5, 6, 3]));;
gap> MaximalDClasses(S);
[ <Green's D-class: [2,8](1,3)(4)(5)(6)(7)>,
  <Green's D-class: [8,3](1,7,6,5,2)(4)> ]
```

### 10.1.8 NrRegularDClasses

▷ NrRegularDClasses(*S*) (attribute)
▷ RegularDClasses(*S*) (attribute)

**Returns:** A positive integer, or a list.

NrRegularDClasses returns the number of regular $\mathscr{D}$-classes of the semigroup *S*.

RegularDClasses returns a list of the regular $\mathscr{D}$-classes of the semigroup *S*.

See also IsRegularGreensClass (10.3.2) and IsRegularDClass (**Reference: IsRegularDClass**).

Example

```
gap> S := Semigroup(Transformation([1, 3, 4, 1, 3, 5]),
>                   Transformation([5, 1, 6, 1, 6, 3]));;
gap> NrRegularDClasses(S);
3
gap> NrDClasses(S);
7
gap> AsSet(RegularDClasses(S));
[ <Green's D-class: Transformation( [ 1, 4, 1, 1, 4, 3 ] )>,
  <Green's D-class: Transformation( [ 1, 1, 1, 1, 1 ] )>,
  <Green's D-class: Transformation( [ 1, 1, 1, 1, 1, 1 ] )> ]
```

### 10.1.9 NrXClasses

▷ NrDClasses(*obj*) (attribute)
▷ NrHClasses(*obj*) (attribute)
▷ NrLClasses(*obj*) (attribute)
▷ NrRClasses(*obj*) (attribute)

**Returns:** A positive integer.

NrXClasses returns the number of Green's classes in *obj* where *obj* can be a semigroup, $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{R}$-class where appropriate. If the actual Green's classes are not required, then it is more efficient to use

Example

```
NrHClasses(obj)
```

than

Example

```
Length(HClasses(obj))
```

since the Green's classes themselves are not created when NrXClasses is called.

See also GreensRClasses (10.1.4), GreensRClasses (**Reference: GreensRClasses**), IteratorOfRClasses (10.2.2), and

Example

```
gap> S := Semigroup(
>  Transformation([1, 2, 5, 4, 3, 8, 7, 6]),
>  Transformation([1, 6, 3, 4, 7, 2, 5, 8]),
>  Transformation([2, 1, 6, 7, 8, 3, 4, 5]),
>  Transformation([3, 2, 3, 6, 1, 6, 1, 2]),
>  Transformation([5, 2, 3, 6, 3, 4, 7, 4]));;
gap> x := Transformation([2, 5, 4, 7, 4, 3, 6, 3]);;
gap> R := RClass(S, x);
```

```
<Green's R-class: Transformation( [ 2, 5, 4, 7, 4, 3, 6, 3 ] )>
gap> NrHClasses(R);
12
gap> D := DClass(R);
<Green's D-class: Transformation( [ 2, 5, 4, 7, 4, 3, 6, 3 ] )>
gap> NrHClasses(D);
72
gap> L := LClass(S, x);
<Green's L-class: Transformation( [ 2, 5, 4, 7, 4, 3, 6, 3 ] )>
gap> NrHClasses(L);
6
gap> NrHClasses(S);
1555
gap> S := Semigroup(Transformation([4, 6, 5, 2, 1, 3]),
>                   Transformation([6, 3, 2, 5, 4, 1]),
>                   Transformation([1, 2, 4, 3, 5, 6]),
>                   Transformation([3, 5, 6, 1, 2, 3]),
>                   Transformation([5, 3, 6, 6, 6, 2]),
>                   Transformation([2, 3, 2, 6, 4, 6]),
>                   Transformation([2, 1, 2, 2, 2, 4]),
>                   Transformation([4, 4, 1, 2, 1, 2]));;
gap> NrRClasses(S);
150
gap> Size(S);
6342
gap> x := Transformation([1, 3, 3, 1, 3, 5]);;
gap> D := DClass(S, x);
<Green's D-class: Transformation( [ 1, 3, 3, 1, 3, 5 ] )>
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
```

### 10.1.10 PartialOrderOfXClasses

▷ `PartialOrderOfDClasses(`*S*`)` (attribute)
▷ `PartialOrderOfLClasses(`*S*`)` (attribute)
▷ `PartialOrderOfRClasses(`*S*`)` (attribute)

**Returns:** A digraph.

Let `X` be one of Green's $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{R}$-relations. Then `PartialOrderOfXClasses` returns a digraph `D` where `OutNeighbours(D)[i]` contains every `j` such that `GreensXClasses(S)[j]` is immediately less than `GreensXClasses(S)[i]` in the partial order of `X`-classes of $S$. The reflexive

transitive closure of the digraph D is the partial order of X-classes of $S$ (in the sense of the Digraphs package).

The partial order on the X-classes is defined as follows.

**Green's $\mathscr{D}$-relation:**

$x \leq y$ if and only if $S^1xS^1$ is a subset of $S^1yS^1$.

**Green's $\mathscr{L}$-relation:**

$x \leq y$ if and only if $S^1x$ is a subset of $S^1y$.

**Green's $\mathscr{R}$-relation:**

$x \leq y$ if and only if $xS^1$ is a subset of $yS^1$.

See also GreensDClasses (10.1.4), GreensDClasses (**Reference: GreensDClasses**), IsGreensLessThanOrEqual (**Reference: IsGreensLessThanOrEqual**), and \< (10.3.1).

Example

```
gap> S := Semigroup(Transformation([2, 4, 1, 2]),
>                   Transformation([3, 3, 4, 1]));;
gap> PartialOrderOfDClasses(S);
<immutable digraph with 4 vertices, 3 edges>
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);
<immutable digraph with 5 vertices, 4 edges>
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
```

### 10.1.11 LengthOfLongestDClassChain

▷ LengthOfLongestDClassChain($S$) (attribute)

**Returns:** A non-negative integer.

If $S$ is a semigroup, then LengthOfLongestDClassChain returns the length of the longest chain in the partial order defined by PartialOrderOfDClasses($S$). See PartialOrderOfDClasses (10.1.10).

The partial order on the $\mathscr{D}$-classes is defined by $x \leq y$ if and only if $S^1xS^1$ is a subset of $S^1yS^1$. A *chain* of $\mathscr{D}$-classes is a collection of n $\mathscr{D}$-classes $D_1, D_2, \ldots D_n$ such that $D_1 < D_2 < \cdots < D_n$. The *length* of such a chain is n - 1.

Example

```
gap> S := TrivialSemigroup();;
gap> LengthOfLongestDClassChain(S);
0
gap> T := ZeroSemigroup(5);;
gap> LengthOfLongestDClassChain(T);
1
gap> U := MonogenicSemigroup(14, 7);;
gap> LengthOfLongestDClassChain(U);
13
gap> V := FullTransformationMonoid(6);
<full transformation monoid of degree 6>
gap> LengthOfLongestDClassChain(V);
5
```

### 10.1.12 IsGreensDGreaterThanFunc

▷ IsGreensDGreaterThanFunc($S$) (attribute)

**Returns:** A function.

IsGreensDGreaterThanFunc($S$) returns a function func such that for any two elements x and y of $S$, func(x, y) returns true if the $\mathscr{D}$-class of x in $S$ is greater than or equal to the $\mathscr{D}$-class of y in $S$ under the usual ordering of Green's $\mathscr{D}$-classes of a semigroup.

Example

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

```
false
gap> IsGreensLessThanOrEqual(DClass(S, S.3),
>                            DClass(S, S.1));
true
gap> IsGreensLessThanOrEqual(DClass(S, S.1),
>                            DClass(S, S.3));
false
```

## 10.2 Iterators and enumerators of classes and representatives

In this section, we describe the methods in the Semigroups package for incrementally determining Green's classes or their representatives.

### 10.2.1 IteratorOfXClassReps

▷ `IteratorOfDClassReps(`*S*`)` (operation)
▷ `IteratorOfHClassReps(`*S*`)` (operation)

**Returns:** An iterator.

Returns an iterator of the representatives of the Green's classes contained in the semigroup `S`. See (**Reference: Iterators**) for more information on iterators.

See also `GreensRClasses` (**Reference: GreensRClasses**), `GreensRClasses` (10.1.4), and `IteratorOfRClasses` (10.2.2).

### 10.2.2 IteratorOfXClasses

▷ `IteratorOfDClasses(`*S*`)` (operation)
▷ `IteratorOfRClasses(`*S*`)` (operation)

**Returns:** An iterator.

Returns an iterator of the Green's classes in the semigroup `S`. See (**Reference: Iterators**) for more information on iterators.

This function is useful if you are, for example, looking for an $\mathscr{R}$-class of a semigroup with a particular property but do not necessarily want to compute all of the $\mathscr{R}$-classes.

See also `GreensRClasses` (10.1.4), `GreensRClasses` (**Reference: GreensRClasses**), and `NrRClasses` (10.1.9).

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: [8,19,14][11,4][13,15,5][17,20]>
gap> Size(R);
10020240
```

## 10.3 Properties of Green's classes

In this section, we describe the properties and operators of Green's classes that are available in the Semigroups package.

### 10.3.1 Less than for Green's classes

▷ \<(*left-expr*, *right-expr*) (method)

**Returns:** `true` or `false`.

The Green's class *left-expr* is less than or equal to *right-expr* if they belong to the same semigroup and the representative of *left-expr* is less than the representative of *right-expr* under <; see also `Representative` (**Reference: Representative**).

Please note that this is not the usual order on the Green's classes of a semigroup as defined in (**Reference: Green's Relations**). See also `IsGreensLessThanOrEqual` (**Reference: IsGreensLessThanOrEqual**).

Example

```
gap> S := FullTransformationSemigroup(4);;
gap> A := GreensRClassOfElement(S, Transformation([2, 1, 3, 1]));
<Green's R-class: Transformation( [ 2, 1, 3, 1 ] )>
gap> B := GreensRClassOfElement(S, Transformation([1, 2, 3, 4]));
<Green's R-class: IdentityTransformation>
gap> A < B;
false
gap> B < A;
true
gap> IsGreensLessThanOrEqual(A, B);
true
gap> IsGreensLessThanOrEqual(B, A);
false
gap> S := SymmetricInverseSemigroup(4);;
gap> A := GreensJClassOfElement(S, PartialPerm([1, 3, 4]));;
gap> B := GreensJClassOfElement(S, PartialPerm([3, 1]));;
gap> A < B;
true
gap> B < A;
false
gap> IsGreensLessThanOrEqual(A, B);
false
gap> IsGreensLessThanOrEqual(B, A);
true
```

### 10.3.2 IsRegularGreensClass

▷ IsRegularGreensClass(*class*) (property)

**Returns:** `true` or `false`.

This function returns `true` if *class* is a regular Green's class and `false` if it is not. See also `IsRegularDClass` (**Reference: IsRegularDClass**), `IsGroupHClass` (**Reference: IsGroupHClass**), `GroupHClassOfGreensDClass` (**Reference: GroupHClassOfGreensDClass**), `GroupHClass` (10.4.1), `NrIdempotents` (11.10.2), `Idempotents` (11.10.1), and `IsRegularSemigroupElement` (**Reference: IsRegularSemigroupElement**).

The function `IsRegularDClass` produces the same output as the GAP library functions with the same name; see `IsRegularDClass` (**Reference: IsRegularDClass**).

Example

```
gap> S := Monoid(Transformation([10, 8, 7, 4, 1, 4, 10, 10, 7, 2]),
>                Transformation([5, 2, 5, 5, 9, 10, 8, 3, 8, 10]));;
```

```
gap> f := Transformation([1, 1, 10, 8, 8, 8, 1, 1, 10, 8]);;
gap> R := RClass(S, f);;
gap> IsRegularGreensClass(R);
true
gap> S := Monoid(Transformation([2, 3, 4, 5, 1, 8, 7, 6, 2, 7]),
>                Transformation([3, 8, 7, 4, 1, 4, 3, 3, 7, 2]));;
gap> f := Transformation([3, 8, 7, 4, 1, 4, 3, 3, 7, 2]);;
gap> R := RClass(S, f);;
gap> IsRegularGreensClass(R);
false
gap> NrIdempotents(R);
0
gap> S := Semigroup(Transformation([2, 1, 3, 1]),
>                   Transformation([3, 1, 2, 1]),
>                   Transformation([4, 2, 3, 3]));;
gap> f := Transformation([4, 2, 3, 3]);;
gap> L := GreensLClassOfElement(S, f);;
gap> IsRegularGreensClass(L);
false
gap> R := GreensRClassOfElement(S, f);;
gap> IsRegularGreensClass(R);
false
gap> g := Transformation([4, 4, 4, 4]);;
gap> IsRegularSemigroupElement(S, g);
true
gap> IsRegularGreensClass(LClass(S, g));
true
gap> IsRegularGreensClass(RClass(S, g));
true
gap> IsRegularDClass(DClass(S, g));
true
gap> DClass(S, g) = RClass(S, g);
false
```

### 10.3.3 IsGreensClassNC

▷ `IsGreensClassNC(`*`class`*`)` (property)

**Returns:** `true` or `false`.

A Green's class *`class`* of a semigroup `S` satisfies `IsGreensClassNC` if it was not known to GAP that the representative of *`class`* was an element of `S` at the point that *`class`* was created.

## 10.4 Attributes of Green's classes

In this section, we describe the attributes of Green's classes that are available in the Semigroups package.

### 10.4.1 GroupHClass

▷ `GroupHClass(`*`class`*`)` (attribute)

**Returns:** A group $\mathscr{H}$-class of the $\mathscr{D}$-class *`class`* if it is regular and `fail` if it is not.

GroupHClass is a synonym for GroupHClassOfGreensDClass (**Reference: GroupHClassOfGreensDClass**).

See also IsGroupHClass (**Reference: IsGroupHClass**), IsRegularDClass (**Reference: IsRegularDClass**), IsRegularGreensClass (10.3.2), and IsRegularSemigroup (12.1.18).

Example

```
gap> S := Semigroup(Transformation([2, 6, 7, 2, 6, 1, 1, 5]),
>                   Transformation([3, 8, 1, 4, 5, 6, 7, 1]));;
gap> IsRegularSemigroup(S);
false
gap> iter := IteratorOfDClasses(S);;
gap> repeat D := NextIterator(iter); until IsRegularDClass(D);
gap> D;
<Green's D-class: Transformation( [ 6, 1, 1, 6, 1, 2, 2, 6 ] )>
gap> NrIdempotents(D);
12
gap> NrRClasses(D);
8
gap> NrLClasses(D);
4
gap> GroupHClass(D);
<Green's H-class: Transformation( [ 1, 2, 2, 1, 2, 6, 6, 1 ] )>
gap> GroupHClassOfGreensDClass(D);
<Green's H-class: Transformation( [ 1, 2, 2, 1, 2, 6, 6, 1 ] )>
gap> StructureDescription(GroupHClass(D));
"S3"
gap> repeat D := NextIterator(iter); until not IsRegularDClass(D);
gap> D;
<Green's D-class: Transformation( [ 7, 5, 2, 2, 6, 1, 1, 2 ] )>
gap> IsRegularDClass(D);
false
gap> GroupHClass(D);
fail
gap> S := InverseSemigroup(
> PartialPerm([2, 1, 6, 0, 3]), PartialPerm([3, 5, 2, 0, 0, 6]));;
gap> x := PartialPerm([1 .. 3], [6, 3, 1]);;
gap> First(DClasses(S), x -> not IsTrivial(GroupHClass(x)));
<Green's D-class: <identity partial perm on [ 1, 2 ]>>
gap> StructureDescription(GroupHClass(last));
"C2"
```

### 10.4.2 SchutzenbergerGroup

▷ SchutzenbergerGroup(*class*) (attribute)

**Returns:** A group.

SchutzenbergerGroup returns the generalized Schutzenberger group (defined below) of the $\mathscr{R}$-, $\mathscr{D}$-, $\mathscr{L}$-, or $\mathscr{H}$-class *class*.

If f is an element of a semigroup of transformations or partial permutations and im(f) denotes the image of f, then the *generalized Schutzenberger group* of im(f) is the permutation group

$$\{ g|_{\mathrm{im}(f)} : \mathrm{im}(f*g) = \mathrm{im}(f) \}.$$

The generalized Schutzenberger group of the kernel `ker(f)` of a transformation `f` or the domain `dom(f)` of a partial permutation `f` is defined analogously.

The generalized Schutzenberger group of a Green's class is then defined as follows.

**$\mathscr{R}$-class**
: The generalized Schutzenberger group of the image or range of the representative of the $\mathscr{R}$-class.

**$\mathscr{L}$-class**
: The generalized Schutzenberger group of the kernel or domain of the representative of the $\mathscr{L}$-class.

**$\mathscr{H}$-class**
: The intersection of the generalized Schutzenberger groups of the $\mathscr{R}$- and $\mathscr{L}$-class containing the $\mathscr{H}$-class.

**$\mathscr{D}$-class**
: The intersection of the generalized Schutzenberger groups of the $\mathscr{R}$- and $\mathscr{L}$-class containing the representative of the $\mathscr{D}$-class.

The output of this attribute is difficult to describe for other types of semigroup. However, a general description is given in [EEMP19].

Example

```
gap> S := Semigroup(Transformation([4, 4, 3, 5, 3]),
>                   Transformation([5, 1, 1, 4, 1]),
>                   Transformation([5, 5, 4, 4, 5]));;
gap> f := Transformation([5, 5, 4, 4, 5]);;
gap> SchutzenbergerGroup(RClass(S, f));
Group([ (4,5) ])
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 7],
>             [9, 2, 4, 8]),
> PartialPerm([1, 2, 6, 7, 8, 9, 10],
>             [6, 8, 4, 5, 9, 1, 3]),
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 9],
>             [7, 4, 1, 6, 9, 5, 2, 3]));;
gap> List(DClasses(S), SchutzenbergerGroup);
[ Group(()), Group(()), Group(()), Group(()), Group([ (4,9) ]),
  Group(()), Group(()), Group([ (5,8,6), (5,8) ]), Group(()),
  Group(()), Group(()), Group(()), Group(()), Group(()),
  Group([ (1,7,5,6,9,3) ]), Group([ (1,6)(3,5) ]), Group(()),
  Group(()), Group(()), Group(()), Group(()), Group(()), Group(()) ]
```

### 10.4.3 StructureDescriptionSchutzenbergerGroups

▷ `StructureDescriptionSchutzenbergerGroups(`*S*`)` (attribute)

**Returns:** Distinct structure descriptions of the Schutzenberger groups of a semigroup.

`StructureDescriptionSchutzenbergerGroups` returns the distinct values of `StructureDescription` (**Reference: StructureDescription**) when it is applied to the Schutzenberger groups of the $\mathscr{R}$-classes of the semigroup *S*.

Example

```
gap> S := Semigroup([
>  PartialPerm([1, 2, 3], [2, 5, 4]),
>  PartialPerm([1, 2, 3], [4, 1, 2]),
>  PartialPerm([1, 2, 3], [5, 2, 3]),
>  PartialPerm([1, 2, 4, 5], [2, 1, 4, 3]),
>  PartialPerm([1, 2, 5], [2, 3, 5]),
>  PartialPerm([1, 2, 3, 5], [2, 3, 5, 4]),
>  PartialPerm([1, 2, 3, 5], [4, 2, 5, 1]),
>  PartialPerm([1, 2, 3, 5], [5, 2, 4, 3]),
>  PartialPerm([1, 2, 5], [5, 4, 3])]);;
gap> StructureDescriptionSchutzenbergerGroups(S);
[ "1", "C2", "S3" ]
gap> S := Monoid(
> Bipartition([[1, 2, 5, -1, -2], [3, 4, -3, -5], [-4]]),
> Bipartition([[1, 2, -2], [3, -1], [4], [5], [-3, -4], [-5]]),
> Bipartition([[1], [2, 3, -5], [4, -3], [5, -2], [-1, -4]]));
<bipartition monoid of degree 5 with 3 generators>
gap> StructureDescriptionSchutzenbergerGroups(S);
[ "1", "C2" ]
```

### 10.4.4 StructureDescriptionMaximalSubgroups

▷ `StructureDescriptionMaximalSubgroups(`*S*`)` (attribute)

**Returns:** Distinct structure descriptions of the maximal subgroups of a semigroup.

`StructureDescriptionMaximalSubgroups` returns the distinct values of `StructureDescription` (**Reference: StructureDescription**) when it is applied to the maximal subgroups of the semigroup *S*.

Example

```
gap> S := DualSymmetricInverseSemigroup(6);
<inverse block bijection monoid of degree 6 with 3 generators>
gap> StructureDescriptionMaximalSubgroups(S);
[ "1", "C2", "S3", "S4", "S5", "S6" ]
gap> S := Semigroup(
>  PartialPerm([1, 3, 4, 5, 8],
>              [8, 3, 9, 4, 5]),
>  PartialPerm([1, 2, 3, 4, 8],
>              [10, 4, 1, 9, 6]),
>  PartialPerm([1, 2, 3, 4, 5, 6, 7, 10],
>              [4, 1, 6, 7, 5, 3, 2, 10]),
>  PartialPerm([1, 2, 3, 4, 6, 8, 10],
>              [4, 9, 10, 3, 1, 5, 2]));;
gap> StructureDescriptionMaximalSubgroups(S);
[ "1", "C2", "C3", "C4" ]
```

### 10.4.5 MultiplicativeNeutralElement (for an H-class)

▷ `MultiplicativeNeutralElement(`*H*`)` (method)

**Returns:** A semigroup element or `fail`.

If the $\mathscr{H}$-class *H* of a semigroup `S` is a subgroup of `S`, then `MultiplicativeNeutralElement` returns the identity of *H*. If *H* is not a subgroup of `S`, then `fail` is returned.

Example

```
gap> S := Semigroup([PartialPerm([1, 5, 2]),
>  PartialPerm([2, 0, 4]), PartialPerm([4, 1, 5]),
>  PartialPerm([1, 0, 3, 0, 4]), PartialPerm([1, 2, 0, 3, 5]),
>  PartialPerm([1, 3, 2, 0, 5]), PartialPerm([5, 0, 0, 4, 3])]);;
gap> H := HClass(S, PartialPerm([1, 2]));;
gap> MultiplicativeNeutralElement(H);
<identity partial perm on [ 1, 2 ]>
gap> H := HClass(S, PartialPerm([1, 4]));;
gap> MultiplicativeNeutralElement(H);
fail
```

### 10.4.6 StructureDescription (for an H-class)

▷ `StructureDescription(`*class*`)` (attribute)

**Returns:** A string or `fail`.

`StructureDescription` returns the value of `StructureDescription` (**Reference: StructureDescription**) when it is applied to a group isomorphic to the group $\mathscr{H}$-class *class*. If *class* is not a group $\mathscr{H}$-class, then `fail` is returned.

Example

```
gap> S := Semigroup(
> PartialPerm([1, 2, 3, 4, 6, 7, 8, 9],
>             [1, 9, 4, 3, 5, 2, 10, 7]),
> PartialPerm([1, 2, 4, 7, 8, 9],
>             [6, 2, 4, 9, 1, 3]));;
gap> H := HClass(S, PartialPerm([1, 2, 3, 4, 7, 9],
>                               [1, 7, 3, 4, 9, 2]));;
gap> StructureDescription(H);
"C6"
```

### 10.4.7 InjectionPrincipalFactor

▷ `InjectionPrincipalFactor(`*D*`)` (attribute)

▷ `InjectionNormalizedPrincipalFactor(`*D*`)` (attribute)

▷ `IsomorphismReesMatrixSemigroup(`*D*`)` (attribute)

**Returns:** A injective mapping.

If the $\mathscr{D}$-class *D* is a subsemigroup of a semigroup S, then the *principal factor* of *D* is just *D* itself. If *D* is not a subsemigroup of S, then the principal factor of *D* is the semigroup with elements *D* and a new element 0 with multiplication of $x, y \in D$ defined by:

$$xy = \begin{cases} x * y \text{ (in } S) & \text{if } x * y \in D \\ 0 & \text{if } xy \notin D. \end{cases}$$

`InjectionPrincipalFactor` returns an injective function from the $\mathscr{D}$-class *D* to a Rees (0-)matrix semigroup, which contains the principal factor of *D* as a subsemigroup.

If *D* is a subsemigroup of its parent semigroup, then the function returned by `InjectionPrincipalFactor` or `IsomorphismReesMatrixSemigroup` is an isomorphism from *D* to a Rees matrix semigroup; see `ReesMatrixSemigroup` (**Reference: ReesMatrixSemigroup**).

If *D* is not a semigroup, then the function returned by `InjectionPrincipalFactor` is an injective function from *D* to a Rees 0-matrix semigroup isomorphic to the principal factor of

*D*; see ReesZeroMatrixSemigroup (**Reference: ReesZeroMatrixSemigroup**). In this case, IsomorphismReesMatrixSemigroup and IsomorphismReesZeroMatrixSemigroup returns an error.

InjectionNormalizedPrincipalFactor returns the composition of InjectionPrincipalFactor with RZMSNormalization (6.5.6) or RMSNormalization (6.5.7) as appropriate.

See also PrincipalFactor (10.4.8).

Example

```
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 6, 8, 10],
>             [2, 6, 7, 9, 1, 5]),
> PartialPerm([1, 2, 3, 4, 6, 7, 8, 10],
>             [3, 8, 1, 9, 4, 10, 5, 6]));;
gap> x := PartialPerm([1, 2, 5, 6, 7, 9],
>                     [1, 2, 5, 6, 7, 9]);;
gap> D := GreensDClassOfElement(S, x);
<Green's D-class: <identity partial perm on [ 1, 2, 5, 6, 7, 9 ]>>
gap> R := Range(InjectionPrincipalFactor(D));
<Rees 0-matrix semigroup 3x3 over Group(())>
gap> MatrixOfReesZeroMatrixSemigroup(R);
[ [ (), 0, 0 ], [ 0, (), 0 ], [ 0, 0, () ] ]
gap> Size(R);
10
gap> Size(D);
9
gap> S := Semigroup(
> Bipartition([[1, 2, 3, -3, -5], [4], [5, -2], [-1, -4]]),
> Bipartition([[1, 3, 5], [2, 4, -3], [-1, -2, -4, -5]]),
> Bipartition([[1, 5, -2, -4], [2, 3, 4, -1, -5], [-3]]),
> Bipartition([[1, 5, -1, -2, -3], [2, 4, -4], [3, -5]]));;
gap> D := GreensDClassOfElement(S,
> Bipartition([[1, 5, -2, -4], [2, 3, 4, -1, -5], [-3]]));
<Green's D-class: <bipartition: [ 1, 5, -2, -4 ], [ 2, 3, 4, -1, -5 ]
   , [ -3 ]>>
gap> InjectionNormalizedPrincipalFactor(D);
MappingByFunction( <Green's D-class: <bipartition: [ 1, 5, -2, -4 ],
  [ 2, 3, 4, -1, -5 ], [ -3 ]>>, <Rees matrix semigroup 1x1 over
  Group([ (1,2) ])>, function( x ) ... end, function( x ) ... end )
```

### 10.4.8 PrincipalFactor

▷ PrincipalFactor(*D*) (attribute)

▷ NormalizedPrincipalFactor(*D*) (attribute)

**Returns:** A Rees (0-)matrix semigroup.

If *D* is a $\mathscr{D}$-class of semigroup, then PrincipalFactor(*D*) is just shorthand for Range(InjectionPrincipalFactor(*D*)), and NormalizedPrincipalFactor(*D*) is shorthand for Range(InjectionNormalizedPrincipalFactor(*D*)).

See InjectionPrincipalFactor (10.4.7) and InjectionNormalizedPrincipalFactor (10.4.7) for more details.

Example

```
gap> S := Semigroup([PartialPerm([1, 2, 3], [1, 3, 4]),
>  PartialPerm([1, 2, 3], [2, 5, 3]),
>  PartialPerm([1, 2, 3, 4], [2, 4, 1, 5]),
>  PartialPerm([1, 3, 5], [5, 1, 3])]);;
gap> PrincipalFactor(MinimalDClass(S));
<Rees matrix semigroup 1x1 over Group(())>
gap> MultiplicativeZero(S);
<empty partial perm>
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 4, 5, -1, -3], [-2, -5], [-4]]),
> Bipartition([[1, -5], [2, 3, 4, 5, -1, -3], [-2, -4]]),
> Bipartition([[1, 5, -4], [2, 4, -1, -5], [3, -2, -3]]));;
gap> D := MinimalDClass(S);
<Green's D-class: <bipartition: [ 1, 2, 3, 4, 5, -1, -3 ],
  [ -2, -5 ], [ -4 ]>>
gap> NormalizedPrincipalFactor(D);
<Rees matrix semigroup 1x5 over Group(())>
```

## 10.5 Operations for Green's relations and classes

In this section, we describe some operations related to Green's classes that are available in the Semigroups package.

### 10.5.1 LeftGreensMultiplier

▷ `LeftGreensMultiplier(S, a, b)` (operation)
▷ `RightGreensMultiplier(S, a, b)` (operation)

**Returns:** An element.

If $S$ is a semigroup, and `a` and `b` are $\mathscr{L}$-related elements of $S$, then `LeftGreensMultiplier` returns an element `s` such that `s * a = b`. The element `s` is of the same type as the elements of $S$ but may or may not be an element of $S$. In particular, if $S$ is not a monoid and `a = b`, then `One(GeneratorsOfSemigroup(S))` or an adjoined identity may be returned. Even if `a <> b`, then it is not guaranteed that the returned element `s` will belong to $S$. It is guaranteed that the left action of `s` on the elements of the $\mathscr{L}$-class of `a` is the same as the left action of an element of $S$ with the identity adjoined.

`LeftGreensMultiplier` gives an error if `a` and `b` are not $\mathscr{L}$-related elements of $S$.

The operation `RightGreensMultiplier` is defined analogously.

Example

```
gap> S := Semigroup(Transformation([4, 4, 3, 5, 3]),
>                   Transformation([5, 1, 1, 4, 1]),
>                   Transformation([5, 5, 4, 4, 5]));;
gap> a := Transformation([5, 5, 4, 4, 5]);;
gap> LeftGreensMultiplier(S, a, a);
Transformation( [ 1, 1, 3, 3, 1 ] )
gap> RightGreensMultiplier(S, a, a);
Transformation( [ 5, 5, 5, 4, 5 ] )
gap> b := Transformation([5, 4, 4, 5, 4]);
Transformation( [ 5, 4, 4, 5, 4 ] )
```

```
gap> s := LeftGreensMultiplier(S, a, b);
Transformation( [ 1, 3, 3, 1, 3 ] )
gap> s * a;
Transformation( [ 5, 4, 4, 5, 4 ] )
gap> b := Transformation([4, 4, 5, 5, 4]);
Transformation( [ 4, 4, 5, 5, 4 ] )
gap> s := RightGreensMultiplier(S, a, b);
Transformation( [ 4, 4, 4, 5, 4 ] )
gap> a * s = b;
true
```

# Chapter 11

# Attributes and operations for semigroups

In this chapter we describe the methods that are available in Semigroups for determining the attributes of a semigroup, and the operations which can be applied to a semigroup.

## 11.1 Accessing the elements of a semigroup

### 11.1.1 AsListCanonical

▷ `AsListCanonical(`*S*`)` (attribute)
▷ `EnumeratorCanonical(`*S*`)` (attribute)
▷ `IteratorCanonical(`*S*`)` (operation)

**Returns:** A list, enumerator, or iterator.

When the argument *S* is a semigroup satisfying `CanUseFroidurePin` (6.1.4), `AsListCanonical` returns a list of the elements of *S* in the order they are enumerated by the Froidure-Pin Algorithm. This is the same as the order used to index the elements of *S* in `RightCayleyDigraph` (11.2.1) and `LeftCayleyDigraph` (11.2.1).

`EnumeratorCanonical` and `IteratorCanonical` return an enumerator and an iterator where the elements are ordered in the same way as `AsListCanonical`. Using `EnumeratorCanonical` or `IteratorCanonical` will often use less memory than `AsListCanonical`, but may have slightly worse performance if the elements of the semigroup are looped over repeatedly. `EnumeratorCanonical` returns the same list as `AsListCanonical` if `AsListCanonical` has ever been called for *S*.

If *S* is an acting semigroup, then the value returned by `AsList` may not equal the value returned by `AsListCanonical`. `AsListCanonical` exists so that there is a method for obtaining the elements of *S* in the particular order used by `RightCayleyDigraph` (11.2.1) and `LeftCayleyDigraph` (11.2.1).

See also `PositionCanonical` (11.1.2).

Example

```
gap> S := Semigroup(Transformation([1, 3, 2]));;
gap> AsListCanonical(S);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> IteratorCanonical(S);
<iterator>
gap> EnumeratorCanonical(S);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> S := Monoid([Matrix(IsBooleanMat, [[1, 0, 0],
>                                       [0, 1, 0],
```

```
>                                        [0, 1, 0]])]);
<commutative monoid of 3x3 boolean matrices with 1 generator>
gap> it := IteratorCanonical(S);
<iterator>
gap> NextIterator(it);
Matrix(IsBooleanMat, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
gap> en := EnumeratorCanonical(S);
<enumerator of <commutative monoid of size 2, 3x3 boolean matrices
 with 1 generator>>
gap> en[1];
Matrix(IsBooleanMat, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
gap> Position(en, en[1]);
1
gap> Length(en);
2
```

### 11.1.2 PositionCanonical

▷ `PositionCanonical(`*S*`, `*x*`)` (operation)

**Returns:** A positive integer or `fail`.

When the argument *S* is a semigroup satisfying `CanUseFroidurePin` (6.1.4) and *x* is an element of *S*, `PositionCanonical` returns the position of *x* in `AsListCanonical(`*S*`)` or equivalently the position of *x* in `EnumeratorCanonical(`*S*`)`.

See also `AsListCanonical` (11.1.1) and `EnumeratorCanonical` (11.1.1).

Example

```
gap> S := FullTropicalMaxPlusMonoid(2, 3);
<monoid of 2x2 tropical max-plus matrices with 13 generators>
gap> x := Matrix(IsTropicalMaxPlusMatrix, [[1, 3], [2, 1]], 3);
Matrix(IsTropicalMaxPlusMatrix, [[1, 3], [2, 1]], 3)
gap> PositionCanonical(S, x);
234
gap> EnumeratorCanonical(S)[234] = x;
true
```

### 11.1.3 Enumerate

▷ `Enumerate(`*S*`[, `*limit*`])` (operation)

**Returns:** A semigroup (the argument).

If *S* is a semigroup with representation `CanUseFroidurePin` (6.1.4) and *limit* is a positive integer, then this operation can be used to enumerate at least *limit* elements of *S*, or `Size(`*S*`)` elements if this is less than *limit*, using the Froidure-Pin Algorithm.

If the optional second argument *limit* is not given, then the semigroup is enumerated until all of its elements have been found.

Example

```
gap> S := FullTransformationMonoid(7);
<full transformation monoid of degree 7>
gap> Enumerate(S, 1000);
<full transformation monoid of degree 7>
```

### 11.1.4 IsEnumerated

▷ `IsEnumerated(S)` (operation)

**Returns:** `true` or `false`.

If `S` is a semigroup with representation `CanUseFroidurePin` (6.1.4), then this operation returns `true` if the Froidure-Pin Algorithm has been run to completion (i.e. all of the elements of `S` have been found) and `false` if `S` has not been fully enumerated.

## 11.2 Cayley graphs

### 11.2.1 RightCayleyDigraph

▷ `RightCayleyDigraph(S)` (attribute)

▷ `LeftCayleyDigraph(S)` (attribute)

**Returns:** A digraph.

When the argument `S` is a semigroup satisfying `CanUseFroidurePin` (6.1.4), `RightCayleyDigraph` returns the right Cayley graph of `S`, as a `Digraph` (**Digraphs: Digraph**) `digraph` where the vertex `OutNeighbours(digraph)[i][j]` is `PositionCanonical(S, AsListCanonical(S)[i] * GeneratorsOfSemigroup(S)[j])`. The digraph returned by `LeftCayleyDigraph` is defined analogously.

The digraph returned by this attribute belongs to the category `IsCayleyDigraph` (**Digraphs: IsCayleyDigraph**), the semigroup `S` and the generators used to create the digraph can be recovered from the digraph using `SemigroupOfCayleyDigraph` (**Digraphs: SemigroupOfCayleyDigraph**) and `GeneratorsOfCayleyDigraph` (**Digraphs: GeneratorsOfCayleyDigraph**).

Example

```
gap> S := FullTransformationMonoid(2);
<full transformation monoid of degree 2>
gap> RightCayleyDigraph(S);
<immutable multidigraph with 4 vertices, 12 edges>
gap> LeftCayleyDigraph(S);
<immutable multidigraph with 4 vertices, 12 edges>
```

## 11.3 Random elements of a semigroup

### 11.3.1 Random (for a semigroup)

▷ `Random(S)` (method)

**Returns:** A random element.

This function returns a random element of the semigroup `S`. If the elements of `S` have been calculated, then one of these is chosen randomly. Otherwise, if the data structure for `S` is known, then a random element of a randomly chosen $\mathscr{R}$-class is returned. If the data structure for `S` has not been calculated, then a short product (at most `2 * Length(GeneratorsOfSemigroup(S))`) of generators is returned.

## 11.4 Properties of elements in a semigroup

### 11.4.1 IndexPeriodOfSemigroupElement

▷ `IndexPeriodOfSemigroupElement(x)` (operation)

**Returns:** A list of two positive integers.

If `x` is a semigroup element, then `IndexPeriodOfSemigroupElement(x)` returns the pair `[m, r]`, where `m` and `r` are the least positive integers such that `x^(m + r) = x ^ m`. The number `m` is known as the *index* of `x`, and the number `r` is known as the *period* of `x`.

Example

```
gap> x := Transformation([2, 6, 3, 5, 6, 1]);;
gap> IndexPeriodOfSemigroupElement(x);
[ 2, 3 ]
gap> m := IndexPeriodOfSemigroupElement(x)[1];;
gap> r := IndexPeriodOfSemigroupElement(x)[2];;
gap> x ^ (m + r) = x ^ m;
true
gap> x := PartialPerm([0, 2, 3, 0, 5]);
<identity partial perm on [ 2, 3, 5 ]>
gap> IsIdempotent(x);
true
gap> IndexPeriodOfSemigroupElement(x);
[ 1, 1 ]
```

### 11.4.2 SmallestIdempotentPower

▷ `SmallestIdempotentPower(x)` (attribute)

**Returns:** A positive integer.

If `x` is a semigroup element, then `SmallestIdempotentPower(x)` returns the least positive integer `n` such that `x^n` is an idempotent. The smallest idempotent power of `x` is the least multiple of the period of `x` that is greater than or equal to the index of `x`; see `IndexPeriodOfSemigroupElement` (11.4.1).

Example

```
gap> x := Transformation([4, 1, 4, 5, 1]);
Transformation( [ 4, 1, 4, 5, 1 ] )
gap> SmallestIdempotentPower(x);
3
gap> ForAll([1 .. 2], i -> not IsIdempotent(x ^ i));
true
gap> IsIdempotent(x ^ 3);
true
gap> x := Bipartition([[1, 2, -3, -4], [3, -5], [4, -1], [5, -2]]);
<block bijection: [ 1, 2, -3, -4 ], [ 3, -5 ], [ 4, -1 ], [ 5, -2 ]>
gap> SmallestIdempotentPower(x);
4
gap> ForAll([1 .. 3], i -> not IsIdempotent(x ^ i));
true
gap> x := PartialPerm([]);
<empty partial perm>
gap> SmallestIdempotentPower(x);
1
```

```
gap> IsIdempotent(x);
true
```

## 11.5 Operations for elements in a semigroup

### 11.5.1 OneInverseOfSemigroupElement

▷ `OneInverseOfSemigroupElement(`*S*`, `*x*`)` (operation)

**Returns:** One inverse of an element of a semigroup or `fail`.

`OneInverseOfSemigroupElement` returns one inverse of the element `x` in the semigroup *S* and returns `fail` if this element has no inverse in *S*.

An *inverse* of an element *x* in a semigroup *S* is an element y of *S* such that *x* `*` y `*` *x* `=` *x* and y `*` *x* `*` y `=` *y*. See also `OnePseudoInverseOfSemigroupElement` (11.5.2).

Example

```
gap> S := FullTransformationMonoid(4);
<full transformation monoid of degree 4>
gap> s := Transformation([2, 3, 1, 1]);
Transformation( [ 2, 3, 1, 1 ] )
gap> OneInverseOfSemigroupElement(S, s);
Transformation( [ 3, 1, 2, 2 ] )
gap> e := IdentityTransformation;
IdentityTransformation
gap> OneInverseOfSemigroupElement(S, e);
IdentityTransformation
gap> F := FreeSemigroup(1);
<free semigroup on the generators [ s1 ]>
gap> OneInverseOfSemigroupElement(F, F.1);
Error, the semigroup is not finite
```

### 11.5.2 OnePseudoInverseOfSemigroupElement

▷ `OnePseudoInverseOfSemigroupElement(`*x*`)` (operation)

**Returns:** One pseudo-inverse of an element of a semigroup.

`OnePseudoInverseOfSemigroupElement` returns one pseudo-inverse of the element *x*. This is an element of the same type as *x* belonging to some semigroup or monoid containing *x*.

An *pseudo-inverse* of an element *x* is an element y such that *x* `*` y `*` *x* `=` *x*.

Methods are installed for this operation for the following types of elements:

**transformations**

`IsTransformation` (**Reference: IsTransformation**);

**partial perms**

`IsPartialPerm` (**Reference: IsPartialPerm**);

**bipartitions**

`IsBipartition` (3.1.1);

**matrices over finite fields**

GAP matrices with entries in a finite field;

**elements of McAlister triple semigroups**
IsMcAlisterTripleSemigroupElement (8.4.7);

**elements of regular Rees 0-matrix semigroups over groups**
IsReesZeroMatrixSemigroupElement (**Reference: IsReesZeroMatrixSemigroupElement**).

See also OneInverseOfSemigroupElement (11.5.1).

Example

```
gap> s := Transformation([2, 3, 1, 1]);
Transformation( [ 2, 3, 1, 1 ] )
gap> OnePseudoInverseOfSemigroupElement(s);
Transformation( [ 3, 1, 2, 1 ] )
gap> e := IdentityTransformation;
IdentityTransformation
gap> OnePseudoInverseOfSemigroupElement(e);
IdentityTransformation
gap> F := FreeSemigroup(1);
<free semigroup on the generators [ s1 ]>
gap> OnePseudoInverseOfSemigroupElement(F, F.1);
Error, no method found! For debugging hints type ?Recovery from NoMeth\
odFound
Error, no 1st choice method found for `OnePseudoInverseOfSemigroupElem\
ent' on 2 arguments
```

## 11.6 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 .

### 11.6.1 EvaluateWord

▷ EvaluateWord(*gens*, *w*) (operation)

**Returns:** A semigroup element.

The argument *gens* should be a collection of generators of a semigroup and the argument *w* should be a list of positive integers less than or equal to the length of *gens*. This operation evaluates the word *w* in the generators *gens*. More precisely, EvaluateWord(*gens*, *w*) returns the equivalent of:

Example

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

see also Factorization (11.6.2).

**for elements of a semigroup**
When *gens* is a list of elements of a semigroup and *w* is a list of positive integers less than or equal to the length of *gens*, this operation returns the product gens[w[1]] * gens[w[2]] * .. . * gens[w[n]] when the length of *w* is n.

**for elements of an inverse semigroup**
When *gens* is a list of elements with a semigroup inverse and *w* is a list of non-zero

integers whose absolute value does not exceed the length of *gens*, this operation returns the product gens[AbsInt(w[1])] ^ SignInt(w[1]) * .. . * gens[AbsInt(w[n])] ^ SignInt(w[n]) where n is the length of *w*.

Note that EvaluateWord(*gens*, []) returns One(*gens*) if *gens* belongs to the category IsMultiplicativeElementWithOne (**Reference: IsMultiplicativeElementWithOne**).

Example

```
gap> gens := [
> Transformation([2, 4, 4, 6, 8, 8, 6, 6]),
> Transformation([2, 7, 4, 1, 4, 6, 5, 2]),
> Transformation([3, 6, 2, 4, 2, 2, 2, 8]),
> Transformation([4, 3, 6, 4, 2, 1, 2, 6]),
> Transformation([4, 5, 1, 3, 8, 5, 8, 2])];;
gap> S := Semigroup(gens);;
gap> x := Transformation([1, 4, 6, 1, 7, 2, 7, 6]);;
gap> word := Factorization(S, x);
[ 4, 2 ]
gap> EvaluateWord(gens, word);
Transformation( [ 1, 4, 6, 1, 7, 2, 7, 6 ] )
gap> S := SymmetricInverseMonoid(10);;
gap> x := PartialPerm([2, 6, 7, 0, 0, 9, 0, 1, 0, 5]);
[3,7][8,1,2,6,9][10,5]
gap> word := Factorization(S, x);
[ -2, -2, -2, -2, -3, -2, -2, -2, -2, -2, 5, 2, 5, 5, 2, 5, 2, 2, 2,
  2, -3, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 3, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), word);
[3,7][8,1,2,6,9][10,5]
```

### 11.6.2 Factorization

▷ Factorization(*S*, *x*) (operation)

**Returns:** A word in the generators.

**for semigroups**

When *S* is a semigroup and *x* belongs to *S*, Factorization returns a word in the generators of *S* that is equal to *x*. In this case, a word is a list of positive integers where an entry i corresponds to GeneratorsOfSemigroup(S)[i]. More specifically,

Example

```
EvaluateWord(GeneratorsOfSemigroup(S), Factorization(S, x)) = x;
```

**for inverse semigroups**

When *S* is an inverse semigroup and *x* belongs to *S*, Factorization returns a word in the generators of *S* that is equal to *x*. In this case, a word is a list of non-zero integers where an entry i corresponds to GeneratorsOfSemigroup(S)[i] and -i corresponds to GeneratorsOfSemigroup(S)[i] ^ -1. As in the previous case,

Example

```
EvaluateWord(GeneratorsOfSemigroup(S), Factorization(S, x)) = x;
```

Note that Factorization does not always return a word of minimum length; see MinimalFactorization (11.6.3).

See also EvaluateWord (11.6.1) and GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**).

Example

```
gap> gens := [Transformation([2, 2, 9, 7, 4, 9, 5, 5, 4, 8]),
>             Transformation([4, 10, 5, 6, 4, 1, 2, 7, 1, 2])];;
gap> S := Semigroup(gens);;
gap> x := Transformation([1, 10, 2, 10, 1, 2, 7, 10, 2, 7]);;
gap> word := Factorization(S, x);
[ 2, 2, 1, 2 ]
gap> EvaluateWord(gens, word);
Transformation( [ 1, 10, 2, 10, 1, 2, 7, 10, 2, 7 ] )
gap> S := SymmetricInverseMonoid(8);
<symmetric inverse monoid of degree 8>
gap> x := PartialPerm([1, 2, 3, 4, 5, 8], [7, 1, 4, 3, 2, 6]);
[5,2,1,7][8,6](3,4)
gap> word := Factorization(S, x);
[ -2, -2, -2, -2, -2, -2, 2, 4, 4, 2, 3, 2, -3, -2, -2, 3, 2, -3, -2,
  -2, 4, -3, -4, 2, 2, 3, -2, -3, 4, -3, -4, 2, 2, 3, -2, -3, 2, 2,
  3, -2, -3, 2, 2, 3, -2, -3, 4, -3, -4, 3, 2, -3, -2, -2, 3, 2, -3,
  -2, -2, 4, 3, -4, 3, 2, -3, -2, -2, 3, 2, -3, -2, -2, 3, 2, 2, 3,
  2, 2, 2, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), word);
[5,2,1,7][8,6](3,4)
gap> S := DualSymmetricInverseMonoid(6);;
gap> x := S.1 * S.2 * S.3 * S.2 * S.1;
<block bijection: [ 1, 6, -4 ], [ 2, -2, -3 ], [ 3, -5 ], [ 4, -6 ],
 [ 5, -1 ]>
gap> word := Factorization(S, x);
[ -2, -2, -2, -2, -2, 4, 2 ]
gap> EvaluateWord(GeneratorsOfSemigroup(S), word);
<block bijection: [ 1, 6, -4 ], [ 2, -2, -3 ], [ 3, -5 ], [ 4, -6 ],
 [ 5, -1 ]>
```

### 11.6.3 MinimalFactorization

▷ MinimalFactorization(*S*, *x*) (operation)

**Returns:** A minimal word in the generators.

This operation returns a minimal length word in the generators of the semigroup *S* that equals the element *x*. In this case, a word is a list of positive integers where an entry i corresponds to GeneratorsOfSemigroup(*S*)[i]. More specifically,

Example

```
EvaluateWord(GeneratorsOfSemigroup(S), MinimalFactorization(S, x)) = x;
```

MinimalFactorization involves exhaustively enumerating *S* until the element *x* is found, and so MinimalFactorization may be less efficient than Factorization (11.6.2) for some semigroups.

Unlike Factorization (11.6.2) this operation does not distinguish between semigroups and inverse semigroups. See also EvaluateWord (11.6.1) and GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**).

Example

```
gap> S := Semigroup(Transformation([2, 2, 9, 7, 4, 9, 5, 5, 4, 8]),
>                   Transformation([4, 10, 5, 6, 4, 1, 2, 7, 1, 2]));
```

```
<transformation semigroup of degree 10 with 2 generators>
gap> x := Transformation([8, 8, 2, 2, 9, 2, 8, 8, 9, 9]);
Transformation( [ 8, 8, 2, 2, 9, 2, 8, 8, 9, 9 ] )
gap> Factorization(S, x);
[ 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1 ]
gap> MinimalFactorization(S, x);
[ 1, 2, 1, 1, 1, 1, 2, 2, 1 ]
```

### 11.6.4 NonTrivialFactorization

▷ NonTrivialFactorization(*S*, *x*) (operation)

**Returns:** A non-trivial word in the generators, or fail.

When *S* is a semigroup and *x* belongs to *S*, this operation returns a non-trivial word in the generators of the semigroup *S* that equals *x*, if one exists. The definition of a word in the generators is the same as given in Factorization (11.6.2) for semigroups and inverse semigroups. A word is non-trivial if it has length two or more.

If no non-trivial word for *x* exists, then *x* is an indecomposable element of *S* and this operation returns fail; see IndecomposableElements (11.7.6).

When *x* does not belong to GeneratorsOfSemigroup(*S*), any factorization of *x* is non-trivial. In this case, NonTrivialFactorization returns the same word as Factorization (11.6.2).

See also EvaluateWord (11.6.1) and GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**).

Example

```
gap> x := Transformation([5, 4, 2, 1, 3]);;
gap> y := Transformation([4, 4, 2, 4, 1]);;
gap> S := Semigroup([x, y]);
<transformation semigroup of degree 5 with 2 generators>
gap> NonTrivialFactorization(S, x * y);
[ 1, 2 ]
gap> Factorization(S, x);
[ 1 ]
gap> NonTrivialFactorization(S, x);
[ 1, 1, 1, 1, 1, 1 ]
gap> Factorization(S, y);
[ 2 ]
gap> NonTrivialFactorization(S, y);
[ 2, 1, 1, 1, 1, 1 ]
gap> z := PartialPerm([2]);;
gap> S := Semigroup(z);
<commutative partial perm semigroup of rank 1 with 1 generator>
gap> NonTrivialFactorization(S, z);
fail
```

## 11.7 Generating sets

### 11.7.1 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 a semigroup**

`Generators(S)` is a synonym for `GeneratorsOfSemigroupIdeal` (9.2.1).

**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 ] ) ]
```

### 11.7.2 SmallGeneratingSet

▷ SmallGeneratingSet(*coll*) (attribute)
▷ SmallSemigroupGeneratingSet(*coll*) (attribute)
▷ SmallMonoidGeneratingSet(*coll*) (attribute)
▷ SmallInverseSemigroupGeneratingSet(*coll*) (attribute)
▷ SmallInverseMonoidGeneratingSet(*coll*) (attribute)

**Returns:** A small generating set for a semigroup.

The attributes SmallXGeneratingSet return a relatively small generating subset of the collection of elements *coll*, which can also be a semigroup. The returned value of SmallXGeneratingSet, where applicable, has the property that

Example

```
X(SmallXGeneratingSet(coll)) = X(coll);
```

where X is any of Semigroup (**Reference: Semigroup**), Monoid (**Reference: Monoid**), InverseSemigroup (**Reference: InverseSemigroup**), or InverseMonoid (**Reference: InverseMonoid**).

If the number of generators for *S* is already relatively small, then these functions will often return the original generating set. These functions may return different results in different GAP sessions.

SmallGeneratingSet returns the smallest of the returned values of SmallXGeneratingSet which is applicable to *coll*; see Generators (11.7.1).

As neither irredundancy, nor minimal length are proven, these functions usually return an answer much more quickly than IrredundantGeneratingSubset (11.7.3). These functions can be used whenever a small generating set is desired which does not necessarily need to be minimal.

Example

```
gap> S := Semigroup([
> Transformation([1, 2, 3, 2, 4]),
> Transformation([1, 5, 4, 3, 2]),
> Transformation([2, 1, 4, 2, 2]),
> Transformation([2, 4, 4, 2, 1]),
> Transformation([3, 1, 4, 3, 2]),
> Transformation([3, 2, 3, 4, 1]),
> Transformation([4, 4, 3, 3, 5]),
> Transformation([5, 1, 5, 5, 3]),
> Transformation([5, 4, 3, 5, 2]),
> Transformation([5, 5, 4, 5, 5])]);;
gap> SmallGeneratingSet(S);
[ Transformation( [ 1, 5, 4, 3, 2 ] ), Transformation( [ 3, 2, 3, 4, 1 ] ),
  Transformation( [ 5, 4, 3, 5, 2 ] ), Transformation( [ 1, 2, 3, 2, 4 ] ),
  Transformation( [ 4, 4, 3, 3, 5 ] ) ]
gap> S := RandomInverseMonoid(IsPartialPermMonoid, 10000, 10);;
gap> SmallGeneratingSet(S);
[ [ 1 .. 10 ] -> [ 3, 2, 4, 5, 6, 1, 7, 10, 9, 8 ],
  [ 1 .. 10 ] -> [ 5, 10, 8, 9, 3, 2, 4, 7, 6, 1 ],
  [ 1, 3, 4, 5, 6, 7, 8, 9, 10 ] -> [ 1, 6, 4, 8, 2, 10, 7, 3, 9 ] ]
gap> M := MathieuGroup(24);;
gap> mat := List([1 .. 1000], x -> Random(M));;
gap> Append(mat, [1 .. 1000] * 0);
gap> mat := List([1 .. 138], x -> List([1 .. 57], x -> Random(mat)));;
gap> R := ReesZeroMatrixSemigroup(M, mat);;
gap> U := Semigroup(List([1 .. 200], x -> Random(R)));
```

```
<subsemigroup of 57x138 Rees 0-matrix semigroup with 100 generators>
gap> Length(SmallGeneratingSet(U));
84
gap> S := RandomSemigroup(IsBipartitionSemigroup, 100, 4);
<bipartition semigroup of degree 4 with 96 generators>
gap> Length(SmallGeneratingSet(S));
13
```

### 11.7.3 IrredundantGeneratingSubset

▷ `IrredundantGeneratingSubset(`*`coll`*`)` (operation)

**Returns:** A list of irredundant generators.

If *coll* is a collection of elements of a semigroup, then this function returns a subset `U` of *coll* such that no element of `U` is generated by the other elements of `U`.

Example

```
gap> S := Semigroup([
> Transformation([5, 1, 4, 6, 2, 3]),
> Transformation([1, 2, 3, 4, 5, 6]),
> Transformation([4, 6, 3, 4, 2, 5]),
> Transformation([5, 4, 6, 3, 1, 3]),
> Transformation([2, 2, 6, 5, 4, 3]),
> Transformation([3, 5, 5, 1, 2, 4]),
> Transformation([6, 5, 1, 3, 3, 4]),
> Transformation([1, 3, 4, 3, 2, 1])]);;
gap> IrredundantGeneratingSubset(S);
[ Transformation( [ 1, 3, 4, 3, 2, 1 ] ),
  Transformation( [ 2, 2, 6, 5, 4, 3 ] ),
  Transformation( [ 3, 5, 5, 1, 2, 4 ] ),
  Transformation( [ 5, 1, 4, 6, 2, 3 ] ),
  Transformation( [ 5, 4, 6, 3, 1, 3 ] ),
  Transformation( [ 6, 5, 1, 3, 3, 4 ] ) ]
gap> S := RandomInverseMonoid(IsPartialPermMonoid, 1000, 10);
<inverse partial perm monoid of degree 10 with 1000 generators>
gap> SmallGeneratingSet(S);
[ [ 1 .. 10 ] -> [ 6, 5, 1, 9, 8, 3, 10, 4, 7, 2 ],
  [ 1 .. 10 ] -> [ 1, 4, 6, 2, 8, 5, 7, 10, 3, 9 ],
  [ 1, 2, 3, 4, 6, 7, 8, 9 ] -> [ 7, 5, 10, 1, 8, 4, 9, 6 ],
  [ 1 .. 9 ] -> [ 4, 3, 5, 7, 10, 9, 1, 6, 8 ] ]
gap> IrredundantGeneratingSubset(last);
[ [ 1 .. 9 ] -> [ 4, 3, 5, 7, 10, 9, 1, 6, 8 ],
  [ 1 .. 10 ] -> [ 1, 4, 6, 2, 8, 5, 7, 10, 3, 9 ],
  [ 1 .. 10 ] -> [ 6, 5, 1, 9, 8, 3, 10, 4, 7, 2 ] ]
gap> S := RandomSemigroup(IsBipartitionSemigroup, 1000, 4);
<bipartition semigroup of degree 4 with 749 generators>
gap> SmallGeneratingSet(S);
[ <bipartition: [ 1, -3 ], [ 2, -2 ], [ 3, -1 ], [ 4, -4 ]>,
  <bipartition: [ 1, 3, -2 ], [ 2, -1, -3 ], [ 4, -4 ]>,
  <bipartition: [ 1, -4 ], [ 2, 4, -1, -3 ], [ 3, -2 ]>,
  <bipartition: [ 1, -1, -3 ], [ 2, -4 ], [ 3, 4, -2 ]>,
  <bipartition: [ 1, -2, -4 ], [ 2 ], [ 3, -3 ], [ 4, -1 ]>,
  <bipartition: [ 1, -2 ], [ 2, -1, -3 ], [ 3, 4, -4 ]>,
```

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

### 11.7.4 MinimalSemigroupGeneratingSet

▷ `MinimalSemigroupGeneratingSet(`*S*`)` (attribute)
▷ `MinimalMonoidGeneratingSet(`*S*`)` (attribute)
▷ `MinimalInverseSemigroupGeneratingSet(`*S*`)` (attribute)
▷ `MinimalInverseMonoidGeneratingSet(`*S*`)` (attribute)

**Returns:** A minimal generating set for a semigroup.

The attribute `MinimalXGeneratingSet` returns a minimal generating set for the semigroup *S*, with respect to length. The returned value of `MinimalXGeneratingSet`, where applicable, is a minimal-length list of elements of *S* with the property that

Example

```
X(MinimalXGeneratingSet(S)) = S;
```

where `X` is one of `Semigroup` (**Reference: Semigroup**), `Monoid` (**Reference: Monoid**), `InverseSemigroup` (**Reference: InverseSemigroup**), or `InverseMonoid` (**Reference: InverseMonoid**).

For many types of semigroup, it is not currently possible to find a `MinimalXGeneratingSet` with the Semigroups package.

See also `SmallGeneratingSet` (11.7.2) and `IrredundantGeneratingSubset` (11.7.3).

Example

```
gap> S := MonogenicSemigroup(3, 6);;
gap> MinimalSemigroupGeneratingSet(S);
[ Transformation( [ 2, 3, 4, 5, 6, 1, 6, 7, 8 ] ) ]
gap> S := FullTransformationMonoid(4);;
gap> MinimalSemigroupGeneratingSet(S);
[ Transformation( [ 1, 4, 2, 3 ] ), Transformation( [ 4, 3, 1, 2 ] ),
  Transformation( [ 1, 2, 3, 1 ] ) ]
gap> S := Monoid([
>  PartialPerm([2, 3, 4, 5, 1, 0, 6, 7]),
```

```
>  PartialPerm([3, 4, 5, 1, 2, 0, 0, 6])]);
<partial perm monoid of rank 8 with 2 generators>
gap> IsMonogenicMonoid(S);
true
gap> MinimalMonoidGeneratingSet(S);
[ [8,7,6](1,2,3,4,5) ]
```

### 11.7.5 GeneratorsSmallest (for a semigroup)

▷ `GeneratorsSmallest(`*S*`)` (attribute)

**Returns:** A set of elements.

For a semigroup *S*, `GeneratorsSmallest` returns the lexicographically least set of elements `X` such that `X` generates *S* as a semigroup, and such that `X` is lexicographically ordered and has the property that each `X[i]` is not generated by `X[1]`, `X[2]`, ..., `X[i-1]`.

It can be difficult to find the set of generators `X`, and it might contain a substantial proportion of the elements of *S*.

Two semigroups have the same set of elements if and only if their smallest generating sets are equal. However, due to the complexity of determining the `GeneratorsSmallest`, this is not the method used by the Semigroups package when comparing semigroups.

Example

```
gap> S := Monoid([
>  Transformation([1, 3, 4, 1]),
>  Transformation([2, 4, 1, 2]),
>  Transformation([3, 1, 1, 3]),
>  Transformation([3, 3, 4, 1])]);
<transformation monoid of degree 4 with 4 generators>
gap> GeneratorsSmallest(S);
[ Transformation( [ 1, 1, 1, 1 ] ), Transformation( [ 1, 1, 1, 2 ] ),
  Transformation( [ 1, 1, 1, 3 ] ), Transformation( [ 1, 1, 1 ] ),
  Transformation( [ 1, 1, 2, 1 ] ), Transformation( [ 1, 1, 2, 2 ] ),
  Transformation( [ 1, 1, 3, 1 ] ), Transformation( [ 1, 1, 3, 3 ] ),
  Transformation( [ 1, 1 ] ), Transformation( [ 1, 1, 4, 1 ] ),
  Transformation( [ 1, 2, 1, 1 ] ), Transformation( [ 1, 2, 2, 1 ] ),
  IdentityTransformation, Transformation( [ 1, 3, 1, 1 ] ),
  Transformation( [ 1, 3, 4, 1 ] ), Transformation( [ 2, 1, 1, 2 ] ),
  Transformation( [ 2, 2, 2 ] ), Transformation( [ 2, 4, 1, 2 ] ),
  Transformation( [ 3, 3, 3 ] ), Transformation( [ 3, 3, 4, 1 ] ) ]
gap> T := Semigroup(Bipartition([[1, 2, 3], [4, -1], [-2], [-3], [-4]]),
>                   Bipartition([[1, -3, -4], [2, 3, 4, -2], [-1]]),
>                   Bipartition([[1, 2, 3, 4, -2], [-1, -4], [-3]]),
>                   Bipartition([[1, 2, 3, 4], [-1], [-2], [-3, -4]]),
>                   Bipartition([[1, 2, -1, -2], [3, 4, -3], [-4]]));
<bipartition semigroup of degree 4 with 5 generators>
gap> GeneratorsSmallest(T);
[ <bipartition: [ 1, 2, 3, 4, -1, -2, -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -1, -2 ], [ -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -1 ], [ -2 ], [ -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -2, -3, -4 ], [ -1 ]>,
  <bipartition: [ 1, 2, 3, 4, -2 ], [ -1, -4 ], [ -3 ]>,
  <bipartition: [ 1, 2, 3, 4, -2 ], [ -1 ], [ -3, -4 ]>,
  <bipartition: [ 1, 2, 3, 4, -3 ], [ -1, -2 ], [ -4 ]>,
```

```
    <bipartition: [ 1, 2, 3, 4 ], [ -1, -2, -3 ], [ -4 ]>,
    <bipartition: [ 1, 2, 3, 4, -3, -4 ], [ -1 ], [ -2 ]>,
    <bipartition: [ 1, 2, 3 ], [ 4, -1, -2, -3 ], [ -4 ]>,
    <bipartition: [ 1, 2, -1, -2 ], [ 3, 4, -3 ], [ -4 ]>,
    <bipartition: [ 1, -3 ], [ 2, 3, 4, -1, -2 ], [ -4 ]>,
    <bipartition: [ 1, -3, -4 ], [ 2, 3, 4, -2 ], [ -1 ]> ]
```

### 11.7.6 IndecomposableElements

▷ `IndecomposableElements(S)` (attribute)

**Returns:** A list of elements.

If $S$ is a semigroup, then this attribute returns the set of elements of $S$ that are not decomposable. An element of $S$ is *decomposable* if it can be written as the product of two elements in $S$. An element of $S$ is *indecomposable* if it is not decomposable.

See also `IsSurjectiveSemigroup` (12.1.26).

Note that any generating set for $S$ contains each indecomposable element of $S$. Thus `IndecomposableElements(S)` is a subset of `GeneratorsOfSemigroup(S)`.

Example

```
gap> S := Semigroup([
> Transformation([1, 1, 2, 3]),
> Transformation([1, 1, 1, 2])]);
<transformation semigroup of degree 4 with 2 generators>
gap> x := IndecomposableElements(S);
[ Transformation( [ 1, 1, 2, 3 ] ) ]
gap> IsSubset(GeneratorsOfSemigroup(S), x);
true
gap> T := FullTransformationMonoid(10);
<full transformation monoid of degree 10>
gap> IndecomposableElements(T);
[  ]
gap> B := ZeroSemigroup(IsBipartitionSemigroup, 3);
<commutative non-regular bipartition semigroup of size 3, degree 4
 with 2 generators>
gap> IndecomposableElements(B);
[ <bipartition: [ 1, 2, 3, -1 ], [ 4, -2 ], [ -3 ], [ -4 ]>,
  <bipartition: [ 1, 2, 4, -1 ], [ 3, -2 ], [ -3 ], [ -4 ]> ]
```

## 11.8 Minimal ideals and multiplicative zeros

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

### 11.8.1 MinimalIdeal

▷ `MinimalIdeal(S)` (attribute)

**Returns:** The minimal ideal of a semigroup.

The minimal ideal of a semigroup is the least ideal with respect to containment.

It is significantly easier to find the minimal $\mathscr{D}$-class of a semigroup, than to find its $\mathscr{D}$-classes.

See also RepresentativeOfMinimalIdeal (11.8.2), PartialOrderOfDClasses (10.1.10), IsGreensLessThanOrEqual (**Reference: IsGreensLessThanOrEqual**), and MinimalDClass (10.1.6).

Example

```
gap> S := Semigroup(
> Transformation([3, 4, 1, 3, 6, 3, 4, 6, 10, 1]),
> Transformation([8, 2, 3, 8, 4, 1, 3, 4, 9, 7]));;
gap> MinimalIdeal(S);
<simple transformation semigroup ideal of degree 10 with 1 generator>
gap> Elements(MinimalIdeal(S));
[ Transformation( [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ),
  Transformation( [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ] ),
  Transformation( [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ] ),
  Transformation( [ 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ] ),
  Transformation( [ 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 ] ) ]
gap> x := Transformation([8, 8, 8, 8, 8, 8, 8, 8, 8, 8]);;
gap> D := DClass(S, x);;
gap> ForAll(GreensDClasses(S), x -> IsGreensLessThanOrEqual(D, x));
true
gap> MinimalIdeal(POI(10));
<partial perm group of rank 0>
gap> MinimalIdeal(BrauerMonoid(6));
<simple bipartition *-semigroup ideal of degree 6 with 1 generator>
```

### 11.8.2 RepresentativeOfMinimalIdeal

▷ RepresentativeOfMinimalIdeal(*S*) (attribute)

▷ RepresentativeOfMinimalDClass(*S*) (attribute)

**Returns:** An element of the minimal ideal of a semigroup.

The minimal ideal of a semigroup is the least ideal with respect to containment.

This method returns a representative element of the minimal ideal of $S$ without having to create the minimal ideal itself. In general, beyond being a member of the minimal ideal, the returned element is not guaranteed to have any special properties. However, the element will coincide with the zero element of $S$ if one exists.

This method works particularly well if $S$ is a semigroup of transformations or partial permutations.

See also MinimalIdeal (11.8.1) and MinimalDClass (10.1.6).

Example

```
gap> S := SymmetricInverseSemigroup(10);;
gap> RepresentativeOfMinimalIdeal(S);
<empty partial perm>
gap> B := Semigroup([
> Bipartition([[1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
> Bipartition([[1, -1], [2], [3], [4, -3], [5, 6, -5, -6],
>   [-2, -4]])]);;
gap> RepresentativeOfMinimalIdeal(B);
<bipartition: [ 1, 2 ], [ 3, 6 ], [ 4, 5 ], [ -1, -5, -6 ],
 [ -2, -4 ], [ -3 ]>
gap> S := Semigroup(Transformation([5, 1, 6, 2, 2, 4]),
>                   Transformation([3, 5, 5, 1, 6, 2]));;
gap> RepresentativeOfMinimalDClass(S);
Transformation( [ 5, 6, 6, 3, 3, 5 ] )
```

```
gap> MinimalDClass(S);
<Green's D-class: Transformation( [ 5, 6, 6, 3, 3, 5 ] )>
```

### 11.8.3 MultiplicativeZero

▷ `MultiplicativeZero(`*S*`)` (attribute)

**Returns:** The zero element of a semigroup.

`MultiplicativeZero` returns the zero element of the semigroup *S* if it exists and `fail` if it does not. See also `MultiplicativeZero` (**Reference: MultiplicativeZero**).

Example

```
gap> S := Semigroup(Transformation([1, 4, 2, 6, 6, 5, 2]),
>                   Transformation([1, 6, 3, 6, 2, 1, 6]));;
gap> MultiplicativeZero(S);
Transformation( [ 1, 1, 1, 1, 1, 1, 1 ] )
gap> S := Semigroup(Transformation([2, 8, 3, 7, 1, 5, 2, 6]),
>                   Transformation([3, 5, 7, 2, 5, 6, 3, 8]),
>                   Transformation([6, 7, 4, 1, 4, 1, 6, 2]),
>                   Transformation([8, 8, 5, 1, 7, 5, 2, 8]));;
gap> MultiplicativeZero(S);
fail
gap> S := InverseSemigroup(
> PartialPerm([1, 3, 4], [5, 3, 1]),
> PartialPerm([1, 2, 3, 4], [4, 3, 1, 2]),
> PartialPerm([1, 3, 4, 5], [2, 4, 5, 3]));;
gap> MultiplicativeZero(S);
<empty partial perm>
gap> S := PartitionMonoid(6);
<regular bipartition *-monoid of size 4213597, degree 6 with 4
 generators>
gap> MultiplicativeZero(S);
fail
gap> S := DualSymmetricInverseMonoid(6);
<inverse block bijection monoid of degree 6 with 3 generators>
gap> MultiplicativeZero(S);
<block bijection: [ 1, 2, 3, 4, 5, 6, -1, -2, -3, -4, -5, -6 ]>
```

### 11.8.4 UnderlyingSemigroupOfSemigroupWithAdjoinedZero

▷ `UnderlyingSemigroupOfSemigroupWithAdjoinedZero(`*S*`)` (attribute)

**Returns:** A semigroup, or `fail`.

If *S* is a semigroup for which the property `IsSemigroupWithAdjoinedZero` (12.1.21) is true, (i.e. *S* has a `MultiplicativeZero` (11.8.3) and the set $S \setminus \{0\}$ is a subsemigroup of *S*), then this method returns the semigroup $S \setminus \{0\}$.

Otherwise, if *S* is a semigroup for which the property `IsSemigroupWithAdjoinedZero` (12.1.21) 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
```

## 11.9 Group of units and identity elements

### 11.9.1 GroupOfUnits

▷ `GroupOfUnits(`*S*`)` (attribute)

**Returns:** The group of units of a semigroup or `fail`.

`GroupOfUnits` returns the group of units of the semigroup *S* as a subsemigroup of *S* if it exists and returns `fail` if it does not. Use `IsomorphismPermGroup` (6.5.5) if you require a permutation representation of the group of units.

If a semigroup *S* has an identity `e`, then the *group of units* of *S* is the set of those `s` in *S* such that there exists `t` in *S* where `s*t=t*s=e`. Equivalently, the group of units is the $\mathscr{H}$-class of the identity of *S*.

See also `GreensHClassOfElement` (**Reference: GreensHClassOfElement**), `IsMonoidAsSemigroup` (12.1.13), 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"
```

## 11.10 Idempotents

### 11.10.1 Idempotents

▷ `Idempotents(obj[, n])` (attribute)

**Returns:** A list of idempotents.

The argument *obj* should be a semigroup, $\mathscr{D}$-class, $\mathscr{H}$-class, $\mathscr{L}$-class, or $\mathscr{R}$-class.

If the optional second argument *n* is present and *obj* is a semigroup, then a list of the idempotents in *obj* of rank *n* is returned. If you are only interested in the idempotents of a given rank, then the second version of the function will probably be faster. However, if the optional second argument is present, then nothing is stored in *obj* and so every time the function is called the computation must be repeated.

This function produces essentially the same output as the GAP library function with the same name; see `Idempotents` (**Reference: Idempotents**). The main difference is that this function can be applied to a wider class of objects as described above.

See also `IsRegularDClass` (**Reference: IsRegularDClass**), `IsRegularGreensClass` (10.3.2), `IsGroupHClass` (**Reference: IsGroupHClass**), `NrIdempotents` (11.10.2), and `GroupHClass` (10.4.1).

Example

```
gap> S := Semigroup(Transformation([2, 3, 4, 1]),
>                   Transformation([3, 3, 1, 1]));;
gap> Idempotents(S, 1);
[ ]
gap> AsSet(Idempotents(S, 2));
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 3, 3, 1 ] ),
  Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> AsSet(Idempotents(S));
[ Transformation( [ 1, 1, 3, 3 ] ), IdentityTransformation,
  Transformation( [ 1, 3, 3, 1 ] ), Transformation( [ 2, 2, 4, 4 ] ),
  Transformation( [ 4, 2, 2, 4 ] ) ]
gap> x := Transformation([2, 2, 4, 4]);;
gap> R := GreensRClassOfElement(S, x);;
gap> Idempotents(R);
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 2, 2, 4, 4 ] ) ]
gap> x := Transformation([4, 2, 2, 4]);;
gap> L := GreensLClassOfElement(S, x);;
gap> AsSet(Idempotents(L));
[ Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> D := DClassOfLClass(L);;
```

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

### 11.10.2 NrIdempotents

▷ `NrIdempotents(`*obj*`)` (attribute)

**Returns:** A positive integer.

This function returns the number of idempotents in *obj* where *obj* can be a semigroup, $\mathscr{D}$-, $\mathscr{L}$-, $\mathscr{H}$-, or $\mathscr{R}$-class. If the actual idempotents are not required, then it is more efficient to use `NrIdempotents(obj)` than `Length(Idempotents(obj))` since the idempotents themselves are not created when `NrIdempotents` is called.

See also `Idempotents` (**Reference: Idempotents**) and `Idempotents` (11.10.1), `IsRegularDClass` (**Reference: IsRegularDClass**), `IsRegularGreensClass` (10.3.2) `IsGroupHClass` (**Reference: IsGroupHClass**), and `GroupHClass` (10.4.1).

Example

```
gap> S := Semigroup(Transformation([2, 3, 4, 1]),
>                   Transformation([3, 3, 1, 1]));;
gap> NrIdempotents(S);
5
gap> f := Transformation([2, 2, 4, 4]);;
gap> R := GreensRClassOfElement(S, f);;
gap> NrIdempotents(R);
2
gap> f := Transformation([4, 2, 2, 4]);;
gap> L := GreensLClassOfElement(S, f);;
gap> NrIdempotents(L);
2
gap> D := DClassOfLClass(L);;
gap> NrIdempotents(D);
4
gap> L := GreensLClassOfElement(S, Transformation([3, 1, 1, 3]));;
gap> NrIdempotents(L);
2
gap> H := GroupHClass(D);;
gap> NrIdempotents(H);
1
```

```
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 5, 7, 9, 10],
>             [6, 7, 2, 9, 1, 5, 3]),
> PartialPerm([1, 2, 3, 5, 6, 7, 9, 10],
>            [8, 1, 9, 4, 10, 5, 6, 7]));;
gap> NrIdempotents(S);
236
gap> f := PartialPerm([2, 3, 7, 9, 10],
>                     [7, 2, 1, 5, 3]);;
gap> D := DClassNC(S, f);;
gap> NrIdempotents(D);
13
```

### 11.10.3 IdempotentGeneratedSubsemigroup

▷ `IdempotentGeneratedSubsemigroup(`*S*`)` (attribute)

**Returns:** A semigroup.

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

See also `Idempotents` (11.10.1) and `SmallGeneratingSet` (11.7.2).

Example

```
gap> S := Semigroup(Transformation([1, 1]),
>                   Transformation([2, 1]),
>                   Transformation([1, 2, 2]),
>                   Transformation([1, 2, 3, 4, 5, 1]),
>                   Transformation([1, 2, 3, 4, 5, 5]),
>                   Transformation([1, 2, 3, 4, 6, 5]),
>                   Transformation([1, 2, 3, 5, 4]),
>                   Transformation([1, 2, 3, 7, 4, 5, 7]),
>                   Transformation([1, 2, 4, 8, 8, 3, 8, 7]),
>                   Transformation([1, 2, 8, 4, 5, 6, 7, 8]),
>                   Transformation([7, 7, 7, 4, 5, 6, 1]));;
gap> IdempotentGeneratedSubsemigroup(S) =
> Monoid(Transformation([1, 1]),
>        Transformation([1, 2, 1]),
>        Transformation([1, 2, 2]),
>        Transformation([1, 2, 3, 1]),
>        Transformation([1, 2, 3, 2]),
>        Transformation([1, 2, 3, 4, 1]),
>        Transformation([1, 2, 3, 4, 2]),
>        Transformation([1, 2, 3, 4, 4]),
>        Transformation([1, 2, 3, 4, 5, 1]),
>        Transformation([1, 2, 3, 4, 5, 2]),
>        Transformation([1, 2, 3, 4, 5, 5]),
>        Transformation([1, 2, 3, 4, 5, 7, 7]),
>        Transformation([1, 2, 3, 4, 7, 6, 7]),
>        Transformation([1, 2, 3, 6, 5, 6]),
>        Transformation([1, 2, 3, 7, 5, 6, 7]),
>        Transformation([1, 2, 8, 4, 5, 6, 7, 8]),
>        Transformation([2, 2]));
true
```

```
gap> S := SymmetricInverseSemigroup(5);
<symmetric inverse monoid of degree 5>
gap> IdempotentGeneratedSubsemigroup(S);
<inverse partial perm monoid of rank 5 with 5 generators>
gap> S := DualSymmetricInverseSemigroup(5);
<inverse block bijection monoid of degree 5 with 3 generators>
gap> IdempotentGeneratedSubsemigroup(S);
<inverse block bijection monoid of degree 5 with 10 generators>
gap> IsSemilattice(last);
true
```

## 11.11 Maximal subsemigroups

The Semigroups package provides methods to calculate the maximal subsemigroups of a finite semigroup, subject to various conditions. A *maximal subsemigroup* of a semigroup is a proper subsemigroup that is contained in no other proper subsemigroup of the semigroup.

When computing the maximal subsemigroups of a regular Rees (0-)matrix semigroup over a group, additional functionality is available. As described in [GGR68], a maximal subsemigroup of a finite regular Rees (0-)matrix semigroup over a group is one of 6 possible types. Using the Semigroups package, it is possible to search for only those maximal subsemigroups of certain types.

A maximal subsemigroup of such a Rees (0-)matrix semigroup `R` over a group `G` is either:

1. `{0}`;
2. formed by removing `0`;
3. formed by removing a column (a non-zero $\mathscr{L}$-class);
4. formed by removing a row (a non-zero $\mathscr{R}$-class);
5. formed by removing a set of both rows and columns;
6. isomorphic to a Rees (0-)matrix semigroup of the same dimensions over a maximal subgroup of `G` (in particular, the maximal subsemigroup intersects every $\mathscr{H}$-class of `R`).

Note that if `R` is a Rees matrix semigroup then it has no maximal subsemigroups of types 1, 2, or 5. Only types 3, 4, and 6 are relevant to a Rees matrix semigroup.

### 11.11.1 MaximalSubsemigroups (for a finite semigroup)

▷ `MaximalSubsemigroups(`*S*`)` (attribute)

▷ `MaximalSubsemigroups(`*S*`, `*opts*`)` (operation)

**Returns:** The maximal subsemigroups of *S*.

If *S* is a finite semigroup, then the attribute `MaximalSubsemigroups` returns a list of the non-empty maximal subsemigroups of *S*. The methods used by `MaximalSubsemigroups` are based on [GGR68], and are described in [DMW18].

It is computationally expensive to search for the maximal subsemigroups of a semigroup, and so computations involving `MaximalSubsemigroups` may be very lengthy. A substantial amount of information on the progress of `MaximalSubsemigroups` is provided through the info class `InfoSemigroups` (2.5.1), with increasingly detailed information given at levels 1, 2, and 3.

The behaviour of MaximalSubsemigroups can be altered via the second argument *opts*, which should be a record. The optional components of *opts* are:

**gens (a boolean)**

If *opts*.gens is false or unspecified, then the maximal subsemigroups themselves are returned and not just generating sets for these subsemigroups.

It can be more computationally expensive to return the generating sets for the maximal subsemigroups, than to return the maximal subsemigroups themselves.

**contain (a list)**

If *opts*.contain is a duplicate-free list of elements of *S*, then MaximalSubsemigroups will search for the maximal subsemigroups of *S* which contain those elements.

**D (a $\mathscr{D}$-class)**

For a maximal subsemigroup M of a finite semigroup *S*, there exists a unique $\mathscr{D}$-class which contains the complement of M in *S*. In other words, the elements of *S* which M lacks are contained in a unique $\mathscr{D}$-class.

If *opts*.D is a $\mathscr{D}$-class of *S*, then MaximalSubsemigroups will search exclusively for those maximal subsemigroups of *S* whose complement is contained in *opts*.D.

**types (a list)**

*This option is relevant only if S is a regular Rees (0-)matrix semigroup over a group.*

As described at the start of this subsection, 11.11, a maximal subsemigroup of a regular Rees (0-)matrix semigroup over a group is one of 6 possible types.

If *S* is a regular Rees (0-)matrix semigroup over a group and *opts*.types is a subset of [1 .. 6], then MaximalSubsemigroups will search for those maximal subsemigroups of *S* of the types enumerated by *opts*.types.

The default value for this option is [1 .. 6] (i.e. no restriction).

Example

```
gap> S := FullTransformationSemigroup(3);
<full transformation monoid of degree 3>
gap> MaximalSubsemigroups(S);
[ <transformation semigroup of degree 3 with 7 generators>,
  <transformation semigroup of degree 3 with 7 generators>,
  <transformation semigroup of degree 3 with 7 generators>,
  <transformation semigroup of degree 3 with 7 generators>,
  <transformation monoid of degree 3 with 5 generators> ]
gap> MaximalSubsemigroups(S,
> rec(gens := true, D := DClass(S, Transformation([2, 2, 3]))));
[ [ Transformation( [ 1, 1, 1 ] ), Transformation( [ 3, 3, 3 ] ),
      Transformation( [ 2, 2, 2 ] ), IdentityTransformation,
      Transformation( [ 2, 3, 1 ] ), Transformation( [ 2, 1 ] ) ] ]
gap> MaximalSubsemigroups(S,
> rec(contain := [Transformation([2, 3, 1])]));
[ <transformation semigroup of degree 3 with 7 generators>,
  <transformation monoid of degree 3 with 5 generators> ]
gap> R := PrincipalFactor(
> DClass(FullTransformationMonoid(4), Transformation([2, 2])));
<Rees 0-matrix semigroup 6x4 over Group([ (2,3,4), (2,4) ])>
```

```
gap> MaximalSubsemigroups(R, rec(types := [5],
>   contain := [RMSElement(R, 1, (), 1),
>               RMSElement(R, 1, (2, 3), 2)]));
[ <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators>,
  <subsemigroup of 6x4 Rees 0-matrix semigroup with 10 generators> ]
```

### 11.11.2 NrMaximalSubsemigroups

▷ `NrMaximalSubsemigroups(`*S*`)` (attribute)

**Returns:** The number of maximal subsemigroups of *S*.

If *S* is a finite semigroup, then `NrMaximalSubsemigroups` returns the number of non-empty maximal subsemigroups of *S*. The methods used by `MaximalSubsemigroups` are based on [GGR68], and are described in [DMW18].

It can be significantly faster to find the number of maximal subsemigroups of a semigroup than to find the maximal subsemigroups themselves.

Unless the maximal subsemigroups of *S* are already known, the command `NrMaximalSubsemigroups(`*S*`)` simply calls the command `MaximalSubsemigroups(`*S*`, rec(number := true))`.

For more information about searching for maximal subsemigroups of a finite semigroup in the Semigroups package, and for information about the options available to alter the search, see `MaximalSubsemigroups` (11.11.1). By supplying the additional option *opts*`.number := true`, the number of maximal subsemigroups will be returned rather than the subsemigroups themselves.

Example

```
gap> S := FullTransformationSemigroup(3);
<full transformation monoid of degree 3>
gap> NrMaximalSubsemigroups(S);
5
gap> S := RectangularBand(3, 4);;
gap> NrMaximalSubsemigroups(S);
7
gap> R := PrincipalFactor(
> DClass(FullTransformationMonoid(4), Transformation([2, 2])));
<Rees 0-matrix semigroup 6x4 over Group([ (2,3,4), (2,4) ])>
gap> MaximalSubsemigroups(R, rec(number := true, types := [3, 4]));
10
```

### 11.11.3 IsMaximalSubsemigroup

▷ `IsMaximalSubsemigroup(`*S*`, `*T*`)` (operation)

**Returns:** `true` or `false`.

If *S* and *T* are semigroups, then `IsMaximalSubsemigroup` returns `true` if and only if *T* is a maximal subsemigroup of *S*.

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

Example

```
gap> S := ZeroSemigroup(2);;
gap> IsMaximalSubsemigroup(S, Semigroup(MultiplicativeZero(S)));
```

```
true
gap> S := FullTransformationSemigroup(4);
<full transformation monoid of degree 4>
gap> T := Semigroup(Transformation([3, 4, 1, 2]),
>                   Transformation([1, 4, 2, 3]),
>                   Transformation([2, 1, 1, 3]));
<transformation semigroup of degree 4 with 3 generators>
gap> IsMaximalSubsemigroup(S, T);
true
gap> R := Semigroup(Transformation([3, 4, 1, 2]),
>                   Transformation([1, 4, 2, 2]),
>                   Transformation([2, 1, 1, 3]));
<transformation semigroup of degree 4 with 3 generators>
gap> IsMaximalSubsemigroup(S, R);
false
```

## 11.12 Attributes of transformations and transformation semigroups

### 11.12.1 ComponentRepsOfTransformationSemigroup

▷ `ComponentRepsOfTransformationSemigroup(`*S*`)` (attribute)

**Returns:** The representatives of components of a transformation semigroup.

This function returns the representatives of the components of the action of the transformation semigroup *S* on the set of positive integers not greater than the degree of *S*.

The representatives are the least set of points such that every point can be reached from some representative under the action of *S*.

Example

```
gap> S := Semigroup(
> Transformation([11, 11, 9, 6, 4, 1, 4, 1, 6, 7, 12, 5]),
> Transformation([12, 10, 7, 10, 4, 1, 12, 9, 11, 9, 1, 12]));;
gap> ComponentRepsOfTransformationSemigroup(S);
[ 2, 3, 8 ]
```

### 11.12.2 ComponentsOfTransformationSemigroup

▷ `ComponentsOfTransformationSemigroup(`*S*`)` (attribute)

**Returns:** The components of a transformation semigroup.

This function returns the components of the action of the transformation semigroup *S* on the set of positive integers not greater than the degree of *S*; the components of *S* partition this set.

Example

```
gap> S := Semigroup(
> Transformation([11, 11, 9, 6, 4, 1, 4, 1, 6, 7, 12, 5]),
> Transformation([12, 10, 7, 10, 4, 1, 12, 9, 11, 9, 1, 12]));;
gap> ComponentsOfTransformationSemigroup(S);
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ] ]
```

### 11.12.3 CyclesOfTransformationSemigroup

▷ `CyclesOfTransformationSemigroup(`*S*`)` (attribute)

**Returns:** The cycles of a transformation semigroup.

This function returns the cycles, or strongly connected components, of the action of the transformation semigroup *S* on the set of positive integers not greater than the degree of *S*.

Example

```
gap> S := Semigroup(
> Transformation([11, 11, 9, 6, 4, 1, 4, 1, 6, 7, 12, 5]),
> Transformation([12, 10, 7, 10, 4, 1, 12, 9, 11, 9, 1, 12]));;
gap> CyclesOfTransformationSemigroup(S);
[ [ 1, 11, 12, 5, 4, 6, 10, 7, 9 ], [ 2 ], [ 3 ], [ 8 ] ]
```

### 11.12.4 DigraphOfAction (for a transformation semigroup, list, and action)

▷ `DigraphOfAction(`*S*`, `*list*`, `*act*`)` (operation)

**Returns:** A digraph, or `fail`.

If *S* is a transformation semigroup and *list* is a list such that *S* acts on the items in *list* via the function *act*, then `DigraphOfAction` returns a digraph representing the action of *S* on the items in *list* and any further items output by *act*`(`*list*`[i], `*S*`.j)`.

If *act*`(`*list*`[i], `*S*`.j)` is the `k`-th item in *list*, then in the output digraph there is an edge from the vertex `i` to the vertex `k` labelled by `j`.

The values in *list* and the additional values generated are stored in the vertex labels of the output digraph; see `DigraphVertexLabels` (**Digraphs: DigraphVertexLabels**), and the edge labels are stored in the `DigraphEdgeLabels` (**Digraphs: DigraphEdgeLabels**)

The digraph returned by `DigraphOfAction` has no multiple edges; see `IsMultiDigraph` (**Digraphs: IsMultiDigraph**).

Example

```
gap> S := Semigroup(Transformation([2, 4, 3, 4, 7, 1, 6]),
>                   Transformation([3, 3, 2, 3, 5, 1, 5]));
<transformation semigroup of degree 7 with 2 generators>
gap> list := Concatenation(List([1 .. 7], x -> [x]),
>                          Combinations([1 .. 7], 2));
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7 ], [ 1, 2 ],
  [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 1, 7 ], [ 2, 3 ],
  [ 2, 4 ], [ 2, 5 ], [ 2, 6 ], [ 2, 7 ], [ 3, 4 ], [ 3, 5 ],
  [ 3, 6 ], [ 3, 7 ], [ 4, 5 ], [ 4, 6 ], [ 4, 7 ], [ 5, 6 ],
  [ 5, 7 ], [ 6, 7 ] ]
gap> D := DigraphOfAction(S, list, OnSets);
<immutable digraph with 28 vertices, 54 edges>
gap> OnSets([2, 5], S.1);
[ 4, 7 ]
gap> Position(DigraphVertexLabels(D), [2, 5]);
16
gap> DigraphVertexLabel(D, 25);
[ 4, 7 ]
gap> DigraphEdgeLabel(D, 16, 25);
1
```

### 11.12.5 DigraphOfActionOnPoints (for a transformation semigroup)

▷ DigraphOfActionOnPoints(*S*) (attribute)
▷ DigraphOfActionOnPoints(*S*, *n*) (attribute)

**Returns:** A digraph.

If *S* is a transformation semigroup and *n* is a non-negative integer, then DigraphOfActionOnPoints(*S*, *n*) returns a digraph representing the OnPoints (**Reference: OnPoints**) action of *S* on the set [1 .. *n*].

If the optional argument *n* is not specified, then by default the degree of *S* will be chosen for *n*; see DegreeOfTransformationSemigroup (**Reference: DegreeOfTransformationSemigroup**).

The digraph returned by DigraphOfActionOnPoints has *n* vertices, where the vertex i corresponds to the point i. For each point i in [1 .. *n*], and for each generator f in GeneratorsOfSemigroup(*S*), there is an edge from the vertex i to the vertex i ^ f. See GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**) for further information.

Example

```
gap> S := Semigroup(Transformation([2, 4, 2, 4, 7, 1, 6]),
>                   Transformation([3, 3, 2, 3, 5, 1, 5]));
<transformation semigroup of degree 7 with 2 generators>
gap> D1 := DigraphOfActionOnPoints(S);
<immutable digraph with 7 vertices, 12 edges>
gap> OnPoints(2, S.1);
4
gap> D2 := DigraphOfActionOnPoints(S, 4);
<immutable digraph with 4 vertices, 7 edges>
gap> D2 = InducedSubdigraph(D1, [1 .. 4]);
true
gap> DigraphOfActionOnPoints(S, 5);
<immutable digraph with 5 vertices, 8 edges>
```

### 11.12.6 FixedPointsOfTransformationSemigroup (for a transformation semigroup)

▷ FixedPointsOfTransformationSemigroup(*S*) (attribute)

**Returns:** A set of positive integers.

If *S* is a transformation semigroup, then FixedPointsOfTransformationSemigroup(*S*) returns the set of points i in [1 .. DegreeOfTransformationSemigroup(*S*)] such that i ^ f = i for all f in *S*.

Example

```
gap> f := Transformation([1, 4, 2, 4, 3, 7, 7]);
Transformation( [ 1, 4, 2, 4, 3, 7, 7 ] )
gap> S := Semigroup(f);
<commutative transformation semigroup of degree 7 with 1 generator>
gap> FixedPointsOfTransformationSemigroup(S);
[ 1, 4, 7 ]
```

### 11.12.7 IsTransitive (for a transformation semigroup and a set)

▷ IsTransitive(*S*[, *X*]) (property)
▷ IsTransitive(*S*[, *n*]) (property)

**Returns:** true or false.

A transformation semigroup `S` is *transitive* or *strongly connected* on the set `X` if for every `i, j` in `X` there is an element `s` in `S` such that `i ^ s = j`.

If the optional second argument is a positive integer `n`, then `IsTransitive` returns `true` if `S` is transitive on `[1 .. n]`, and `false` if it is not.

If the optional second argument is not provided, then the degree of `S` is used by default; see `DegreeOfTransformationSemigroup` (**Reference: DegreeOfTransformationSemigroup**).

Example

```
gap> S := Semigroup([
>  Bipartition([
>    [1, 2], [3, 6, -2], [4, 5, -3, -4], [-1, -6], [-5]]),
>  Bipartition([
>    [1, -4], [2, 3, 4, 5], [6], [-1, -6], [-2, -3], [-5]])]);
<bipartition semigroup of degree 6 with 2 generators>
gap> AsSemigroup(IsTransformationSemigroup, S);
<transformation semigroup of size 11, degree 12 with 2 generators>
gap> IsTransitive(last);
false
gap> IsTransitive(AsSemigroup(Group((1, 2, 3))));
true
```

### 11.12.8 SmallestElementSemigroup

▷ `SmallestElementSemigroup(S)` (attribute)

▷ `LargestElementSemigroup(S)` (attribute)

**Returns:** A transformation.

These attributes return the smallest and largest element of the transformation semigroup `S`, respectively. Smallest means the first element in the sorted set of elements of `S` and largest means the last element in the set of elements.

It is not necessary to find the elements of the semigroup to determine the smallest or largest element, and this function has considerably 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 ] )
```

### 11.12.9 CanonicalTransformation

▷ `CanonicalTransformation(trans[, n])` (function)

**Returns:** A transformation.

If `trans` is a transformation, and `n` is a non-negative integer such that the restriction of `trans` to `[1 .. n]` defines a transformation of `[1 .. n]`, then `CanonicalTransformation` returns a canonical representative of the transformation `trans` restricted to `[1 .. n]`.

More specifically, let C(n) be a class of transformations of degree *n* such that AsDigraph returns isomorphic digraphs for every pair of elements in C(n). Recall that for a transformation *trans* and integer *n* the function AsDigraph returns a digraph with *n* vertices and an edge with source x and range x^trans for every x in [1 .. n]. See AsDigraph (**Digraphs: AsDigraph for a binary relation**). Then CanonicalTransformation returns a canonical representative of the class C(n) that contains *trans*.

Example

```
gap> x := Transformation([5, 1, 4, 1, 1]);
Transformation( [ 5, 1, 4, 1, 1 ] )
gap> y := Transformation([3, 3, 2, 3, 1]);
Transformation( [ 3, 3, 2, 3, 1 ] )
gap> CanonicalTransformation(x);
Transformation( [ 3, 5, 2, 2, 2 ] )
gap> CanonicalTransformation(y);
Transformation( [ 3, 5, 2, 2, 2 ] )
```

### 11.12.10 IsConnectedTransformationSemigroup (for a transformation semigroup)

▷ IsConnectedTransformationSemigroup(*S*) (property)

**Returns:** true or false.

A transformation semigroup *S* is connected if the digraph returned by the function DigraphOfActionOnPoints is connected. See IsConnectedDigraph (**Digraphs: IsConnectedDigraph**) and DigraphOfActionOnPoints (11.12.5). The function IsConnectedTransformationSemigroup returns true if the semigroup *S* is connected and false otherwise.

Example

```
gap> S := Semigroup([
>  Transformation([2, 4, 3, 4]),
>  Transformation([3, 3, 2, 3, 3])]);
<transformation semigroup of degree 5 with 2 generators>
gap> IsConnectedTransformationSemigroup(S);
true
```

## 11.13 Attributes of partial perms and partial perm semigroups

### 11.13.1 ComponentRepsOfPartialPermSemigroup

▷ ComponentRepsOfPartialPermSemigroup(*S*) (attribute)

**Returns:** The representatives of components of a partial perm semigroup.

This function returns the representatives of the components of the action of the partial perm semigroup *S* on the set of positive integers where it is defined.

The representatives are the least set of points such that every point can be reached from some representative under the action of *S*.

Example

```
gap> S := Semigroup([
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19],
>             [9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 16, 18, 19, 20],
>             [13, 1, 8, 5, 4, 14, 11, 12, 9, 20, 2, 18, 7, 3, 19])]);;
```

```
gap> ComponentRepsOfPartialPermSemigroup(S);
[ 6, 10, 15, 17 ]
```

### 11.13.2 ComponentsOfPartialPermSemigroup

▷ ComponentsOfPartialPermSemigroup(*S*) (attribute)

**Returns:** The components of a partial perm semigroup.

This function returns the components of the action of the partial perm semigroup *S* on the set of positive integers where it is defined; the components of *S* partition this set.

Example

```
gap> S := Semigroup([
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19],
>             [9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 16, 18, 19, 20],
>             [13, 1, 8, 5, 4, 14, 11, 12, 9, 20, 2, 18, 7, 3, 19])]);;
gap> ComponentsOfPartialPermSemigroup(S);
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20 ],
  [ 15 ], [ 17 ] ]
```

### 11.13.3 CyclesOfPartialPerm

▷ CyclesOfPartialPerm(*x*) (attribute)

**Returns:** The cycles of a partial perm.

This function returns the cycles, or strongly connected components, of the action of the partial perm *x* on the set of positive integers where it is defined.

Example

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

### 11.13.4 CyclesOfPartialPermSemigroup

▷ CyclesOfPartialPermSemigroup(*S*) (attribute)

**Returns:** The cycles of a partial perm semigroup.

This function returns the cycles, or strongly connected components, of the action of the partial perm semigroup *S* on the set of positive integers where it is defined.

Example

```
gap> S := Semigroup([
> PartialPerm([1, 2, 3, 5, 6, 7, 8, 11, 12, 16, 19],
>             [9, 18, 20, 11, 5, 16, 8, 19, 14, 13, 1]),
> PartialPerm([1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 16, 18, 19, 20],
>             [13, 1, 8, 5, 4, 14, 11, 12, 9, 20, 2, 18, 7, 3, 19])]);;
gap> CyclesOfPartialPermSemigroup(S);
[ [ 18, 7, 16 ], [ 1, 9, 12, 14, 2, 20, 19, 3, 8, 11 ], [ 4, 5 ] ]
```

### 11.13.5 UniformRandomPartialPerm

▷ `UniformRandomPartialPerm(n)` (operation)

**Returns:** A partial perm of degree `n` chosen uniformly at random among all partial permutations of that degree.

This function returns a partial perm of degree `n` chosen uniformly at random among all partial permutations of that degree. This function differs from `RandomPartialPerm` (**Reference: RandomPartialPerm**) in that the returned partial perm is chosen *uniformly* at random, whereas `RandomPartialPerm` (**Reference: RandomPartialPerm**) does not use a uniform distribution.

The downside to `UniformRandomPartialPerm` is that it can use a significant amount of memory, and is rather slower than `RandomPartialPerm` (**Reference: RandomPartialPerm**).

Example

```
gap> UniformRandomPartialPerm(1000);
<partial perm on 971 pts with degree 1000, codegree 1000>
```

## 11.14 Attributes of Rees (0-)matrix semigroups

### 11.14.1 RZMSDigraph

▷ `RZMSDigraph(R)` (attribute)

**Returns:** A digraph.

If `R` is an $n$ by $m$ Rees 0-matrix semigroup $M^0[I,T,\Lambda;P]$ (so that $I = \{1,2,\ldots,n\}$ and $\Lambda = \{1,2,\ldots,m\}$) then `RZMSDigraph` returns a symmetric bipartite digraph with $n+m$ vertices. An index $i \in I$ corresponds to the vertex $i$ and an index $j \in \Lambda$ corresponds to the vertex $j+n$.

Two vertices $v$ and $w$ in `RZMSDigraph(R)` are adjacent if and only if $v \in I$, $w-n \in \Lambda$, and `P[w - n][v]` $\neq 0$.

This digraph is commonly called the *Graham-Houghton graph* of `R`.

Example

```
gap> R := PrincipalFactor(
> DClass(FullTransformationMonoid(5),
>        Transformation([2, 4, 1, 5, 5])));
<Rees 0-matrix semigroup 10x5 over Group([ (1,2,3,4), (1,2) ])>
gap> gr := RZMSDigraph(R);
<immutable bipartite digraph with bicomponent sizes 10 and 5>
gap> e := DigraphEdges(gr)[1];
[ 1, 11 ]
gap> Matrix(R)[e[2] - 10][e[1]] <> 0;
true
```

### 11.14.2 RZMSConnectedComponents

▷ `RZMSConnectedComponents(R)` (attribute)

**Returns:** The connected components of a Rees 0-matrix semigroup.

If `R` is an $n$ by $m$ Rees 0-matrix semigroup $M^0[I,T,\Lambda;P]$ (so that $I = \{1,2,\ldots,n\}$ and $\Lambda = \{1,2,\ldots,m\}$) then `RZMSConnectedComponents` returns the connected components of `R`.

*Connectedness* is an equivalence relation on the indices of `R`: the equivalence classes of the relation are called the *connected components* of `R`, and two indices in $I \cup \Lambda$ are connected if and only if their

corresponding vertices in RZMSDigraph(*R*) are connected (see RZMSDigraph (11.14.1)). If *R* has *n* connected components, then RZMSConnectedComponents will return a list of pairs:

[ [ $I_1$,$\Lambda_1$ ], ..., [ $I_k$,$\Lambda_k$ ] ]

where $I = I_1 \sqcup \cdots \sqcup I_k$, $\Lambda = \Lambda_1 \sqcup \cdots \sqcup \Lambda_k$, and for each $l$ the set $I_l \cup \Lambda_l$ is a connected component of *R*. Note that at most one of $I_l$ and $\Lambda_l$ is possibly empty. The ordering of the connected components in the result in unspecified.

Example

```
gap> R := ReesZeroMatrixSemigroup(SymmetricGroup(5),
> [[(), 0, (1, 3), (4, 5), 0],
>  [0, (), 0, 0, (1, 3, 4, 5)],
>  [0, 0, (1, 5)(2, 3), 0, 0],
>  [0, (2, 3)(1, 4), 0, 0, 0]]);
<Rees 0-matrix semigroup 5x4 over Sym( [ 1 .. 5 ] )>
gap> RZMSConnectedComponents(R);
[ [ [ 1, 3, 4 ], [ 1, 3 ] ], [ [ 2, 5 ], [ 2, 4 ] ] ]
```

## 11.15 Attributes of inverse semigroups

### 11.15.1 NaturalLeqInverseSemigroup

▷ NaturalLeqInverseSemigroup(*S*) (attribute)

**Returns:** A function.

NaturalLeqInverseSemigroup returns a function that, when given two elements x, y of the inverse semigroup *S*, returns true if x is less than or equal to y in the natural partial order on *S*.

Example

```
gap> S := Monoid(Transformation([1, 3, 4, 4]),
>                Transformation([1, 4, 2, 4]));
<transformation monoid of degree 4 with 2 generators>
gap> IsInverseSemigroup(S);
true
gap> Size(S);
6
gap> NaturalPartialOrder(S);
[ [ 2, 5, 6 ], [ 6 ], [ 6 ], [ 6 ], [ 6 ], [  ] ]
```

### 11.15.2 JoinIrreducibleDClasses

▷ JoinIrreducibleDClasses(*S*) (attribute)

**Returns:** A list of $\mathscr{D}$-classes.

JoinIrreducibleDClasses returns a list of the join irreducible $\mathscr{D}$-classes of the inverse semigroup of partial permutations, block bijections or partial permutation bipartitions *S*.

A *join irreducible $\mathscr{D}$-class* is a $\mathscr{D}$-class containing only join irreducible elements. See IsJoinIrreducible (12.2.7). If a $\mathscr{D}$-class contains one join irreducible element, then all of the elements in the $\mathscr{D}$-class are join irreducible.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> JoinIrreducibleDClasses(S);
[ <Green's D-class: <identity partial perm on [ 2 ]>> ]
```

```
gap> T := InverseSemigroup([
>   PartialPerm([1, 2, 4, 3]),
>   PartialPerm([1]),
>   PartialPerm([0, 2])]);
<inverse partial perm semigroup of rank 4 with 3 generators>
gap> JoinIrreducibleDClasses(T);
[ <Green's D-class: <identity partial perm on [ 1, 2, 3, 4 ]>>,
  <Green's D-class: <identity partial perm on [ 1 ]>>,
  <Green's D-class: <identity partial perm on [ 2 ]>> ]
gap> D := DualSymmetricInverseSemigroup(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> JoinIrreducibleDClasses(D);
[ <Green's D-class: <block bijection: [ 1, 2, -1, -2 ], [ 3, -3 ]>> ]
```

### 11.15.3 MajorantClosure

▷ `MajorantClosure(`*S*`, `*T*`)` (operation)

**Returns:** A majorantly closed list of elements.

`MajorantClosure` returns a majorantly closed subset of an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions, *S*, as a list. See `IsMajorantlyClosed` (12.2.8).

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
```

### 11.15.4 Minorants

▷ `Minorants(S, f)` (operation)

**Returns:** A list of elements.

`Minorants` takes an element `f` from an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions `S`, and returns a list of the minorants of `f` in `S`.

A *minorant* of `f` is an element of `S` which is strictly less than `f` in the natural partial order of `S`. See `NaturalLeqPartialPerm` (**Reference: NaturalLeqPartialPerm**).

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> x := Elements(S)[13];
[1,3](2)
gap> Minorants(S, x);
[ <empty partial perm>, [1,3], <identity partial perm on [ 2 ]> ]
gap> x := PartialPerm([3, 2, 4, 0]);
[1,3,4](2)
gap> S := InverseSemigroup(x);
<inverse partial perm semigroup of rank 4 with 1 generator>
gap> Minorants(S, x);
[ <identity partial perm on [ 2 ]>, [1,3](2), [3,4](2) ]
```

### 11.15.5 PrimitiveIdempotents

▷ `PrimitiveIdempotents(S)` (attribute)

**Returns:** A list of elements.

An idempotent in an inverse semigroup `S` is *primitive* if it is non-zero and minimal with respect to the `NaturalPartialOrder` (**Reference: NaturalPartialOrder**) on `S`. `PrimitiveIdempotents` returns the list of primitive idempotents in the inverse semigroup `S`.

Example

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

```
    <block bijection: [ 1, 3, -1, -3 ], [ 2, 4, -2, -4 ]>,
    <block bijection: [ 1, 4, -1, -4 ], [ 2, 3, -2, -3 ]>,
    <block bijection: [ 1, -1 ], [ 2, 3, 4, -2, -3, -4 ]> ]
```

### 11.15.6 RightCosetsOfInverseSemigroup

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

**Returns:** A list of lists of elements.

`RightCosetsOfInverseSemigroup` takes a majorantly closed inverse subsemigroup $T$ of an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions $S$. See `IsMajorantlyClosed` (12.2.8). The result is a list of the right cosets of $T$ in $S$.

For $s \in S$, the right coset $\overline{T}s$ 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` (11.15.3). Distinct cosets are disjoint but do not necessarily partition $S$.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> T := InverseSemigroup(MajorantClosure(S, [PartialPerm([1])]));
<inverse partial perm monoid of rank 3 with 6 generators>
gap> IsMajorantlyClosed(S, T);
true
gap> RC := RightCosetsOfInverseSemigroup(S, T);
[ [ <identity partial perm on [ 1 ]>,
      <identity partial perm on [ 1, 2 ]>, [2,3](1), [3,2](1),
      <identity partial perm on [ 1, 3 ]>,
      <identity partial perm on [ 1, 2, 3 ]>, (1)(2,3) ],
  [ [1,3], [2,1,3], [1,3](2), (1,3), [1,3,2], (1,3,2), (1,3)(2) ],
  [ [1,2], (1,2), [1,2,3], [3,1,2], [1,2](3), (1,2)(3), (1,2,3) ] ]
```

### 11.15.7 SameMinorantsSubgroup

▷ `SameMinorantsSubgroup(H)` (attribute)

**Returns:** A list of elements of the group $\mathscr{H}$-class $H$.

Given a group $\mathscr{H}$-class $H$ in an inverse semigroup of partial permutations, block bijections or partial permutation bipartitions `S`, `SameMinorantsSubgroup` returns a list of the elements of $H$ which have the same strict minorants as the identity element of $H$. A *strict minorant* of `x` in $H$ is an element of `S` which is less than `x` (with respect to the natural partial order), but is not equal to `x`.

The returned list of elements of $H$ describe a subgroup of $H$.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> H := GroupHClass(DClass(S, PartialPerm([1, 2, 3])));
<Green's H-class: <identity partial perm on [ 1, 2, 3 ]>>
gap> Elements(H);
[ <identity partial perm on [ 1, 2, 3 ]>, (1)(2,3), (1,2)(3),
  (1,2,3), (1,3,2), (1,3)(2) ]
gap> SameMinorantsSubgroup(H);
[ <identity partial perm on [ 1, 2, 3 ]> ]
gap> T := InverseSemigroup(
> PartialPerm([1, 2, 3, 4], [1, 2, 4, 3]),
```

```
> PartialPerm([1], [1]), PartialPerm([2], [2]));
<inverse partial perm semigroup of rank 4 with 3 generators>
gap> Elements(T);
[ <empty partial perm>, <identity partial perm on [ 1 ]>,
  <identity partial perm on [ 2 ]>,
  <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
gap> x := GroupHClass(DClass(T, PartialPerm([1, 2, 3, 4])));
<Green's H-class: <identity partial perm on [ 1, 2, 3, 4 ]>>
gap> Elements(x);
[ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
gap> AsSet(SameMinorantsSubgroup(x));
[ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4) ]
```

### 11.15.8 SmallerDegreePartialPermRepresentation

▷ SmallerDegreePartialPermRepresentation(*S*) (attribute)

**Returns:** An isomorphism.

SmallerDegreePartialPermRepresentation attempts to find an isomorphism from the inverse semigroup $S$ to an inverse semigroup of partial permutations with small degree. If $S$ is already a partial permutation semigroup, and the function cannot reduce the degree, the identity mapping is returned.

There is no guarantee that the smallest possible degree representation is returned. For more information see [Sch92].

Example

```
gap> S := InverseSemigroup(PartialPerm([2, 1, 4, 3, 6, 5, 8, 7]));
<partial perm group of rank 8 with 1 generator>
gap> Elements(S);
[ <identity partial perm on [ 1, 2, 3, 4, 5, 6, 7, 8 ]>,
  (1,2)(3,4)(5,6)(7,8) ]
gap> iso := SmallerDegreePartialPermRepresentation(S);;
gap> Source(iso) = S;
true
gap> R := Range(iso);
<partial perm group of rank 2 with 1 generator>
gap> Elements(R);
[ <identity partial perm on [ 1, 2 ]>, (1,2) ]
gap> S := DualSymmetricInverseMonoid(5);;
gap> T := Range(IsomorphismPartialPermSemigroup(S));
<inverse partial perm monoid of size 6721, rank 6721 with 3
 generators>
gap> SmallerDegreePartialPermRepresentation(T);
<inverse partial perm monoid of size 6721, rank 6721 with 3
  generators> -> <inverse partial perm monoid of rank 30 with 3
  generators>
```

### 11.15.9 VagnerPrestonRepresentation

▷ VagnerPrestonRepresentation(*S*) (attribute)

**Returns:** An isomorphism to an inverse semigroup of partial permutations.

VagnerPrestonRepresentation returns an isomorphism from an inverse semigroup $S$ where the elements of $S$ have a unique semigroup inverse accessible via Inverse (**Reference: Inverse**), to the inverse semigroup of partial permutations $T$ of degree equal to the size of $S$, which is obtained using the Vagner-Preston Representation Theorem.

More precisely, if $f : S \to T$ is the isomorphism returned by VagnerPrestonRepresentation($S$) and $x$ is in $S$, then $f(x)$ is the partial permutation with domain $Sx^{-1}$ and range $Sx^{-1}x$ defined by $f(x) : sx^{-1} \mapsto sx^{-1}x$.

In many cases, it is possible to find a smaller degree representation than that provided by VagnerPrestonRepresentation using IsomorphismPartialPermSemigroup (**Reference: IsomorphismPartialPermSemigroup**) or SmallerDegreePartialPermRepresentation (11.15.8).

Example

```
gap> S := SymmetricInverseSemigroup(2);
<symmetric inverse monoid of degree 2>
gap> Size(S);
7
gap> iso := VagnerPrestonRepresentation(S);
<symmetric inverse monoid of degree 2> ->
<inverse partial perm monoid of rank 7 with 2 generators>
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);
<inverse bipartition semigroup of size 394, degree 5 with 3
  generators> -> <inverse partial perm semigroup of rank 394 with 5
  generators>
```

### 11.15.10 CharacterTableOfInverseSemigroup

▷ CharacterTableOfInverseSemigroup($S$) (attribute)

**Returns:** The character table of the inverse semigroup $S$ and a list of conjugacy class representatives of $S$.

Returns a list with two entries: the first entry being the character table of the inverse semigroup $S$ as a matrix, while the second entry is a list of conjugacy class representatives of $S$.

The order of the columns in the character table matrix follows the order of the conjugacy class representatives list. The conjugacy class representatives are grouped by $\mathscr{D}$-class and then sorted by rank. Also, as is typical of character tables, the rows of the matrix correspond to the irreducible characters and the columns correspond to the conjugacy classes.

This function was contributed by Jhevon Smith and Ben Steinberg.

Example

```
gap> S := InverseMonoid([
>   PartialPerm([1, 2], [3, 1]),
```

```
>  PartialPerm([1, 2, 3], [1, 3, 4]),
>  PartialPerm([1, 2, 3], [2, 4, 1]),
>  PartialPerm([1, 3, 4], [3, 4, 1])]);;
gap> CharacterTableOfInverseSemigroup(S);
[ [ [ 1, 0, 0, 0, 0, 0, 0, 0 ], [ 3, 1, 1, 1, 0, 0, 0, 0 ],
      [ 3, 1, E(3), E(3)^2, 0, 0, 0, 0 ],
      [ 3, 1, E(3)^2, E(3), 0, 0, 0, 0 ], [ 6, 3, 0, 0, 1, -1, 0, 0 ],
      [ 6, 3, 0, 0, 1, 1, 0, 0 ], [ 4, 3, 0, 0, 2, 0, 1, 0 ],
      [ 1, 1, 1, 1, 1, 1, 1, 1 ] ],
  [ <identity partial perm on [ 1, 2, 3, 4 ]>,
      <identity partial perm on [ 1, 3, 4 ]>, (1,3,4), (1,4,3),
      <identity partial perm on [ 1, 3 ]>, (1,3),
      <identity partial perm on [ 3 ]>, <empty partial perm> ] ]
gap> S := SymmetricInverseMonoid(4);;
gap> CharacterTableOfInverseSemigroup(S);
[ [ [ 1, -1, 1, 1, -1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 3, -1, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 2, 0, -1, 2, 0, 0, 0, 0, 0, 0, 0, 0 ],
      [ 3, 1, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 4, -2, 1, 0, 0, 1, -1, 1, 0, 0, 0, 0 ],
      [ 8, 0, -1, 0, 0, 2, 0, -1, 0, 0, 0, 0 ],
      [ 4, 2, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0 ],
      [ 6, 0, 0, -2, 0, 3, -1, 0, 1, -1, 0, 0 ],
      [ 6, 2, 0, 2, 0, 3, 1, 0, 1, 1, 0, 0 ],
      [ 4, 2, 1, 0, 0, 3, 1, 0, 2, 0, 1, 0 ],
      [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ],
  [ <identity partial perm on [ 1, 2, 3, 4 ]>, (1)(2)(3,4),
      (1)(2,3,4), (1,2)(3,4), (1,2,3,4),
      <identity partial perm on [ 1, 2, 3 ]>, (1)(2,3), (1,2,3),
      <identity partial perm on [ 2, 3 ]>, (2,3),
      <identity partial perm on [ 1 ]>, <empty partial perm> ] ]
```

### 11.15.11 EUnitaryInverseCover

▷ `EUnitaryInverseCover(`*S*`)` (attribute)

**Returns:** A homomorphism between semigroups.

If the argument *S* is an inverse semigroup then this function returns a finite E-unitary inverse cover of *S*. A finite E-unitary cover of *S* is a surjective idempotent separating homomorphism from a finite semigroup satisfying `IsEUnitaryInverseSemigroup` (12.2.3) to *S*. A semigroup homomorphism is said to be idempotent separating if no two idempotents are mapped to the same element of the image.

Example

```
gap> S := InverseSemigroup([PartialPermNC([1, 2], [2, 1]),
> PartialPermNC([1], [1])]);
<inverse partial perm semigroup of rank 2 with 2 generators>
gap> cov := EUnitaryInverseCover(S);
<inverse partial perm semigroup of rank 4 with 2 generators> ->
<inverse partial perm semigroup of rank 2 with 2 generators>
gap> IsEUnitaryInverseSemigroup(Source(cov));
true
gap> S = Range(cov);
```

```
true
```

## 11.16 Nambooripad partial order

### 11.16.1 NambooripadLeqRegularSemigroup

▷ NambooripadLeqRegularSemigroup(*S*) (attribute)

**Returns:** A function.

NambooripadLeqRegularSemigroup returns a function that, when given two elements x, y of the regular semigroup *S*, returns true if x is less than or equal to y in the Nambooripad partial order on *S*. See also NambooripadPartialOrder (11.16.2).

Example

```
gap> S := BrauerMonoid(3);
<regular bipartition *-monoid of degree 3 with 3 generators>
gap> IsRegularSemigroup(S);
true
gap> Size(S);
15
gap> NambooripadPartialOrder(S);
[ [  ], [  ], [  ], [  ], [  ], [  ], [  ], [  ], [  ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ]
gap> NambooripadLeqRegularSemigroup(S)(Elements(S)[3], Elements(S)[9]);
false
gap> NambooripadLeqRegularSemigroup(S)(Elements(S)[2], Elements(S)[15]);
true
```

### 11.16.2 NambooripadPartialOrder

▷ NambooripadPartialOrder(*S*) (attribute)

**Returns:** The Nambooripad partial order on a regular semigroup.

The *Nambooripad partial order* $\leq$ on a regular semigroup *S* is defined by s$\leq$t if the principal right ideal of S generated by s is contained in the principal right ideal of S generated by t and there is an idempotent e in the $\mathscr{R}$-class of s such that s=et. The Nambooripad partial order coincides with the natural partial order when considering inverse semigroups NaturalPartialOrder (**Reference: NaturalPartialOrder**).

NambooripadPartialOrder returns the Nambooripad partial order on the regular semigroup *S* as a list of sets of positive integers where entry i in NambooripadPartialOrder(*S*) is the set of positions in Elements(*S*) of elements which are less than Elements(*S*)[i]. See also NambooripadLeqRegularSemigroup (11.16.1).

Example

```
gap> S := BrauerMonoid(3);
<regular bipartition *-monoid of degree 3 with 3 generators>
gap> IsRegularSemigroup(S);
true
gap> Size(S);
15
gap> NambooripadPartialOrder(S);
```

```
[ [  ], [  ], [  ], [  ], [  ], [  ], [  ], [  ], [  ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ]
gap> NambooripadLeqRegularSemigroup(S)(Elements(S)[3], Elements(S)[9]);
false
gap> NambooripadLeqRegularSemigroup(S)(Elements(S)[2], Elements(S)[15]);
true
```

## 11.17 Monoid character table and Cartan matrix

In this section we describe some operations in Semigroups for computing character tables and Cartan matrices of monoids. These functions are currently implemented for finite monoids over a splitting field of the monoid in characteristic zero. The case for computing character tables and Cartan matrices for finite monoids over a splitting field of the monoid in characteristic zero originates from [Cha25] by Balthazar Charles. An example of a splitting field for a finite monoid is the cyclotomic field with $g$th roots where $g$ is the least common multiple of the exponents of every maximal subgroup of the finite monoid. In what follows cyclotomic field with $g$th roots can be assumed to be what is meant by a splitting field of the monoid.

### 11.17.1 GeneralizedConjugacyClasses (for a semigroup)

▷ `GeneralizedConjugacyClasses(`$S$`)` (attribute)

▷ `GeneralizedConjugacyClassesRepresentatives(`$S$`)` (attribute)

**Returns:** A complete list of generalized conjugacy classes.

`GeneralizedConjugacyClasses` returns a list of the generalized conjugacy classes of the semigroup $S$. Let $R$ be a relation on $S$ such that it is the union of the set $\{(mn,nm)|m,n \in S\}$ and the set $\{(m,m^{(w+1)})|m \in S\}$ where $w$ is `SmallestIdempotentPower` (11.4.2) of `m`. A generalized conjugacy class is an equivalence class of the smallest equivalence relation that contains $R$. `GeneralizedConjugacyClassesRepresentatives` returns a list of the representatives of the generalized conjugacy classes in the semigroup $S$. In a group generalized conjugacy classes coincide with conjugacy classes.

Note that each generalized conjugacy class as currently implemented does not store a list of elements in the conjugacy class, only the representative with no way to compute all the elements in the conjugacy class. This means that `GeneralizedConjugacyClasses` as currently implemented is only as effective as `GeneralizedConjugacyClassesRepresentatives`.

Example

```
gap> S := FullTransformationMonoid(6);;
gap> GeneralizedConjugacyClassesRepresentatives(S);
[ IdentityTransformation, Transformation( [ 1, 2, 3, 4, 6, 5 ] ),
  Transformation( [ 1, 2, 3, 5, 6, 4 ] ),
  Transformation( [ 1, 2, 4, 3, 6, 5 ] ),
  Transformation( [ 1, 2, 4, 5, 6, 3 ] ),
  Transformation( [ 1, 3, 2, 5, 6, 4 ] ),
  Transformation( [ 1, 3, 4, 5, 6, 2 ] ),
  Transformation( [ 2, 1, 4, 3, 6, 5 ] ),
  Transformation( [ 2, 1, 4, 5, 6, 3 ] ),
  Transformation( [ 2, 3, 1, 5, 6, 4 ] ),
```

```
  Transformation( [ 2, 3, 4, 5, 6, 1 ] ),
  Transformation( [ 1, 2, 3, 4, 5, 1 ] ),
  Transformation( [ 1, 2, 3, 5, 4, 1 ] ),
  Transformation( [ 1, 2, 4, 5, 3, 1 ] ),
  Transformation( [ 1, 3, 2, 5, 4, 1 ] ),
  Transformation( [ 1, 3, 4, 5, 2, 1 ] ),
  Transformation( [ 2, 1, 4, 5, 3, 2 ] ),
  Transformation( [ 2, 3, 4, 5, 1, 2 ] ),
  Transformation( [ 2, 2, 3, 4, 5, 2 ] ),
  Transformation( [ 2, 2, 3, 5, 4, 2 ] ),
  Transformation( [ 2, 2, 4, 5, 3, 2 ] ),
  Transformation( [ 3, 3, 2, 5, 4, 3 ] ),
  Transformation( [ 3, 3, 4, 5, 2, 3 ] ),
  Transformation( [ 1, 1, 3, 4, 3, 1 ] ),
  Transformation( [ 1, 1, 4, 3, 4, 1 ] ),
  Transformation( [ 3, 3, 4, 1, 4, 3 ] ),
  Transformation( [ 1, 2, 2, 1, 2, 1 ] ),
  Transformation( [ 2, 1, 1, 2, 1, 2 ] ),
  Transformation( [ 1, 1, 1, 1, 1, 1 ] ) ]
```

### 11.17.2 DClassBicharacter (for a D-class)

▷ `DClassBicharacter(`*D*`)` (attribute)

**Returns:** A matrix of non-negative integers.

`DClassBicharacter` returns a matrix whose `i`th row and `j`th column entry is the size of the set $\{m \in D | hmk = m\}$ where `h` is an element in the `i`th generalized conjugacy class of the `Parent` (**Reference: Parent**) monoid of *D* and `k` is an element in the `j`th generalized conjugacy class of the `Parent` (**Reference: Parent**) monoid of *D*. The order of the generalized conjugacy classes of the `Parent` (**Reference: Parent**) monoid of *D* is determined by the list returned by the attribute `GeneralizedConjugacyClasses` (11.17.1) of the `Parent` (**Reference: Parent**) monoid of *D*.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> D := DClasses(S)[1];;
gap> DClassBicharacter(D);
[ [ 6, 0, 0, 0, 0, 0 ], [ 0, 2, 0, 0, 0, 0 ], [ 0, 0, 3, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0 ] ]
```

### 11.17.3 RegularRepresentationBicharacter (for a semigroup)

▷ `RegularRepresentationBicharacter(`*S*`)` (attribute)

**Returns:** A matrix of non-negative integers.

`RegularRepresentationBicharacter` returns a matrix whose `i`th row and `j`th column entry is the size of the set $\{m \in S | hmk = m\}$ where $h$ is an element in the `i`th generalized conjugacy class of $S$ and $k$ is an element in the `j`th generalized conjugacy class of $S$. The order of the generalized conjugacy classes of $S$ is determined by the list returned by the attribute `GeneralizedConjugacyClasses` (11.17.1) of $S$.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> RegularRepresentationBicharacter(S);
```

```
[ [ 27, 1, 0, 8, 0, 1 ], [ 9, 3, 0, 4, 0, 1 ], [ 3, 1, 3, 2, 0, 1 ],
  [ 9, 1, 0, 4, 0, 1 ], [ 3, 3, 0, 2, 2, 1 ], [ 3, 1, 0, 2, 0, 1 ] ]
```

### 11.17.4 RClassBicharacterOfGroupHClass (for a group H-class)

▷ `RClassBicharacterOfGroupHClass(H)` (attribute)

**Returns:** A matrix of non-negative integers.

`RClassBicharacterOfGroupHClass` returns a matrix whose `i`th row and `j`th column entry is the size of the set $\{m \in R | hmk = m\}$ where `R` is the `RClassOfHClass` (10.1.1) of $H$. Furthermore, $h$ is an element in the `i`th generalized conjugacy class of $H$ as a group and $k$ is an element in the `j`th generalized conjugacy class of the `Parent` (**Reference: Parent**) semigroup of $H$. The order of the generalized conjugacy classes of the `Parent` (**Reference: Parent**) semigroup of $H$ is determined by the list returned by the attribute `GeneralizedConjugacyClasses` (11.17.1) of the `Parent` (**Reference: Parent**) semigroup of $H$. The order of the generalized conjugacy classes of $H$ is determined by the list returned by the attribute `ConjugacyClasses` (**Reference: ConjugacyClasses attribute**) of `OrdinaryCharacterTable` (**Reference: OrdinaryCharacterTable**) of `Range` (**Reference: range**) of `IsomorphismPermGroup` (6.5.5) of $H$.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> D := RegularDClasses(S)[1];;
gap> H := GroupHClass(D);;
gap> RClassBicharacterOfGroupHClass(H);
[ [ 6, 0, 0, 0, 0, 0 ], [ 0, 2, 0, 0, 0, 0 ], [ 0, 0, 3, 0, 0, 0 ] ]
```

### 11.17.5 BlockDiagonalMatrixOfCharacterTables (for a semigroup)

▷ `BlockDiagonalMatrixOfCharacterTables(S)` (attribute)

**Returns:** A matrix.

`BlockDiagonalMatrixOfCharacterTables` returns a block diagonal matrix which has a matrix block for each regular $\mathscr{D}$-class of $S$. Each diagonal block is the character table of one (arbitrary) group $\mathscr{H}$-class in each regular $\mathscr{D}$-class of $S$. The character tables are determined by `OrdinaryCharacterTable` (**Reference: OrdinaryCharacterTable**) of `Range` (**Reference: range**) of `IsomorphismPermGroup` (6.5.5) of `GroupHClass` (10.4.1) of `RegularDClasses` (10.1.8) of $S$.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> BlockDiagonalMatrixOfCharacterTables(S);
[ [ 1, -1, 1, 0, 0, 0 ], [ 2, 0, -1, 0, 0, 0 ], [ 1, 1, 1, 0, 0, 0 ],
  [ 0, 0, 0, 1, -1, 0 ], [ 0, 0, 0, 1, 1, 0 ], [ 0, 0, 0, 0, 0, 1 ] ]
```

### 11.17.6 MonoidCharacterTable

▷ `MonoidCharacterTable(M[, F])` (attribute)

**Returns:** The character table object for $M$ over $F$.

Called with a finite monoid $M$ and optionally a field $F$, `MonoidCharacterTable` returns the character table of the monoid which is defined as the matrix $Trace(X(m))$, where $X$ runs over the simple $FM$-modules and $m$ runs over the generalized conjugacy class representatives of $M$.

To get the character table of a monoid to display like the example below, the irreducible characters need to be computed first using Irr (11.17.7).

If $F$ is not given, then MonoidCharacterTable returns the character table of $M$ over a characteristic zero splitting field of $M$.

At the moment, methods are available for the following cases: if $F$ is not given (i.e. it defaults to a splitting field of $M$ over the rationals) and $M$ is a finite monoid.

For other cases no methods are implemented yet.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> ct := MonoidCharacterTable(S);;
gap> Irr(ct);;
gap> Display(ct);
     c.1 c.2 c.3 c.4 c.5 c.6

X.1   1  -1   1   .   .   .
X.2   2   .  -1   .   .   .
X.3   1   1   1   .   .   .
X.4   2   .  -1   1  -1   .
X.5   3   1   .   1   1   .
X.6   1   1   1   1   1   1
```

### 11.17.7 Irr (for a monoid character table)

▷ Irr($T$) (attribute)

**Returns:** A list of monoid characters.

Irr returns a list of the characters of the irreducible representations of the Parent (**Reference: Parent**) monoid of the monoid character table $T$.

Example

```
gap> M := FullBooleanMatMonoid(2);;
gap> ct := MonoidCharacterTable(M);;
gap> Irr(ct);
[ MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , [ 1, 1, 1, 1, 1 ] ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , [ 0, 1, 2, 3, 1 ] ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , [ 0, 0, 1, 2, 0 ] ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , [ 0, 0, 0, 1, -1 ] ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , [ 0, 0, 0, 1, 1 ] ) ]
```

### 11.17.8 MonoidCartanMatrix

▷ `MonoidCartanMatrix(M[, F])` (attribute)

**Returns:** An object.

Called with a finite monoid `M` and a field `F`, `MonoidCartanMatrix` returns the Cartan matrix of the monoid algebra $FM$ which is defined as the matrix $dimHom(P,Q)/dimEnd(P/rad(FM))$, where $P$ and $Q$ run over the right indecomposable projective modules of FM.

To get the Cartan matrix of a monoid to display like the example below, the projective indecomposable modules need to be computed first using `Pims` (11.17.9).

If `M` is the only argument then `MonoidCartanMatrix` returns the Cartan matrix of the monoid algebra $FM$, where $F$ is a splitting field of `M` over the rationals.

At the moment, methods are available for the following cases: if $F$ is not given (i.e. it defaults to a splitting field of `M` over the rationals) and `M` is a finite monoid.

For other cases no methods are implemented yet.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> cm := MonoidCartanMatrix(S);;
gap> Pims(cm);;
gap> Display(cm);
    X.1 X.2 X.3 X.4 X.5 X.6

P.1   1   .   .   .   .   .
P.2   .   1   .   .   .   .
P.3   1   .   1   1   .   .
P.4   1   .   .   1   .   .
P.5   .   .   .   .   1   .
P.6   .   .   .   1   .   1
```

### 11.17.9 Pims (for a monoid cartan matrix)

▷ `Pims(T)` (attribute)

**Returns:** A list of monoid characters.

`Pims` returns a list of the characters of the projective indecomposable modules of the `Parent` (**Reference: Parent**) monoid of the monoid character table `T`.

Example

```
gap> M := FullBooleanMatMonoid(2);;
gap> cm := MonoidCartanMatrix(M);;
gap> Pims(cm);
[ MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , Projective Cover Of MonoidCharacter( Monoi\
dCharacterTable( Monoid( [ Matrix(IsBooleanMat, [ [ false, true ], [ t\
rue, false ] ]), Matrix(IsBooleanMat, [ [ true, false ], [ true, true \
] ]), Matrix(IsBooleanMat, [ [ true, false ], [ false, false ] ]) ] ) \
) , [ 1, 1, 1, 1, 1 ] ) ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , Projective Cover Of MonoidCharacter( Monoi\
```

```
dCharacterTable( Monoid( [ Matrix(IsBooleanMat, [ [ false, true ], [ t\
rue, false ] ]), Matrix(IsBooleanMat, [ [ true, false ], [ true, true \
] ]), Matrix(IsBooleanMat, [ [ true, false ], [ false, false ] ]) ] ) \
) , [ 0, 1, 2, 3, 1 ] ) ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , Projective Cover Of MonoidCharacter( Monoi\
dCharacterTable( Monoid( [ Matrix(IsBooleanMat, [ [ false, true ], [ t\
rue, false ] ]), Matrix(IsBooleanMat, [ [ true, false ], [ true, true \
] ]), Matrix(IsBooleanMat, [ [ true, false ], [ false, false ] ]) ] ) \
) , [ 0, 0, 1, 2, 0 ] ) ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , Projective Cover Of MonoidCharacter( Monoi\
dCharacterTable( Monoid( [ Matrix(IsBooleanMat, [ [ false, true ], [ t\
rue, false ] ]), Matrix(IsBooleanMat, [ [ true, false ], [ true, true \
] ]), Matrix(IsBooleanMat, [ [ true, false ], [ false, false ] ]) ] ) \
) , [ 0, 0, 0, 1, -1 ] ) ),
  MonoidCharacter( MonoidCharacterTable( Monoid( [ Matrix(IsBooleanMat\
, [ [ false, true ], [ true, false ] ]), Matrix(IsBooleanMat, [ [ true\
, false ], [ true, true ] ]), Matrix(IsBooleanMat, [ [ true, false ], \
[ false, false ] ]) ] ) ) , Projective Cover Of MonoidCharacter( Monoi\
dCharacterTable( Monoid( [ Matrix(IsBooleanMat, [ [ false, true ], [ t\
rue, false ] ]), Matrix(IsBooleanMat, [ [ true, false ], [ true, true \
] ]), Matrix(IsBooleanMat, [ [ true, false ], [ false, false ] ]) ] ) \
) , [ 0, 0, 0, 1, 1 ] ) ) ]
```

# Chapter 12

# Properties of semigroups

In this chapter we describe the methods that are available in Semigroups for determining various properties of a semigroup or monoid.

## 12.1 Arbitrary semigroups

In this section we describe the properties of an arbitrary semigroup or monoid that can be determined using the Semigroups package.

### 12.1.1 IsBand

▷ `IsBand(`*S*`)` (property)

**Returns:** `true` or `false`.

`IsBand` returns `true` if every element of the semigroup *S* is an idempotent and `false` if it is not. An inverse semigroup is a band if and only if it is a semilattice; see `IsSemilattice` (12.1.24).

Example

```
gap> S := Semigroup(
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 1]),
> Transformation([2, 2, 2, 5, 5, 5, 8, 8, 8, 2]),
> Transformation([3, 3, 3, 6, 6, 6, 9, 9, 9, 3]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 4]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 7]));;
gap> IsBand(S);
true
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 3, 4, 8, 9], [5, 8, 7, 6, 9, 1]),
> PartialPerm([1, 3, 4, 7, 8, 9, 10], [2, 3, 8, 7, 10, 6, 1]));;
gap> IsBand(S);
false
gap> IsBand(IdempotentGeneratedSubsemigroup(S));
true
gap> S := PartitionMonoid(4);
<regular bipartition *-monoid of size 4140, degree 4 with 4
 generators>
gap> M := MinimalIdeal(S);
<simple bipartition *-semigroup ideal of degree 4 with 1 generator>
```

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

### 12.1.2 IsBlockGroup

▷ `IsBlockGroup(`*S*`)` (property)

**Returns:** `true` or `false`.

`IsBlockGroup` returns `true` if the semigroup *S* is a block group and `false` if it is not.

A semigroup *S* is a *block group* if every $\mathscr{L}$-class and every $\mathscr{R}$-class of *S* contains at most one idempotent. Every semigroup of partial permutations is a block group.

Example

```
gap> S := Semigroup(Transformation([5, 6, 7, 3, 1, 4, 2, 8]),
>                   Transformation([3, 6, 8, 5, 7, 4, 2, 8]));;
gap> IsBlockGroup(S);
true
gap> S := Semigroup(
> Transformation([2, 1, 10, 4, 5, 9, 7, 4, 8, 4]),
> Transformation([10, 7, 5, 6, 1, 3, 9, 7, 10, 2]));;
gap> IsBlockGroup(S);
false
gap> S := Semigroup(PartialPerm([1, 2], [5, 4]),
>                   PartialPerm([1, 2, 3], [1, 2, 5]),
>                   PartialPerm([1, 2, 3], [2, 1, 5]),
>                   PartialPerm([1, 3, 4], [3, 1, 2]),
>                   PartialPerm([1, 3, 4, 5], [5, 4, 3, 2]));;
gap> T := AsSemigroup(IsBlockBijectionSemigroup, S);
<block bijection semigroup of degree 6 with 5 generators>
gap> IsBlockGroup(T);
true
gap> IsBlockGroup(AsSemigroup(IsBipartitionSemigroup, S));
true
gap> S := Semigroup(
> Bipartition([[1, -2], [2, -3], [3, -4], [4, -1]]),
> Bipartition([[1, -2], [2, -1], [3, -3], [4, -4]]),
> Bipartition([[1, 2, -3], [3, -1, -2], [4, -4]]),
> Bipartition([[1, -1], [2, -2], [3, -3], [4, -4]]));;
gap> IsBlockGroup(S);
true
```

### 12.1.3 IsCommutativeSemigroup

▷ `IsCommutativeSemigroup(`*S*`)` (property)

**Returns:** `true` or `false`.

`IsCommutativeSemigroup` returns `true` if the semigroup *S* is commutative and `false` if it is not. The function `IsCommutative` (**Reference: IsCommutative**) can also be used to test if a semigroup is commutative.

A semigroup *S* is *commutative* if `x * y = y * x` for all `x, y` in *S*.

Example

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

```
gap> IsCommutativeSemigroup(S);
true
gap> IsCommutative(S);
true
gap> S := InverseSemigroup(
>  PartialPerm([1, 2, 3, 4, 5, 6], [2, 5, 1, 3, 9, 6]),
>  PartialPerm([1, 2, 3, 4, 6, 8], [8, 5, 7, 6, 2, 1]));;
gap> IsCommutativeSemigroup(S);
false
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 6, 7, -1, -4, -6],
>              [4, 5, 8, -2, -3, -5, -7, -8]]),
>  Bipartition([[1, 2, -3, -4], [3, -5], [4, -6], [5, -7],
>               [6, -8], [7, -1], [8, -2]]));;
gap> IsCommutativeSemigroup(S);
true
```

### 12.1.4 IsCompletelyRegularSemigroup

▷ `IsCompletelyRegularSemigroup(`*S*`)` (property)

**Returns:** `true` or `false`.

`IsCompletelyRegularSemigroup` returns `true` if every element of the semigroup *S* is contained in a subgroup of *S*.

An inverse semigroup is completely regular if and only if it is a Clifford semigroup; see `IsCliffordSemigroup` (12.2.1).

Example

```
gap> S := Semigroup(Transformation([1, 2, 4, 3, 6, 5, 4]),
>                   Transformation([1, 2, 5, 6, 3, 4, 5]),
>                   Transformation([2, 1, 2, 2, 2, 2, 2]));;
gap> IsCompletelyRegularSemigroup(S);
true
gap> IsInverseSemigroup(S);
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsCompletelyRegularSemigroup(T);
true
gap> IsCliffordSemigroup(T);
true
gap> S := Semigroup(
> Bipartition([[1, 3, -4], [2, 4, -1, -2], [-3]]),
> Bipartition([[1, -1], [2, 3, 4, -3], [-2, -4]]));;
gap> IsCompletelyRegularSemigroup(S);
false
```

### 12.1.5 IsCongruenceFreeSemigroup

▷ `IsCongruenceFreeSemigroup(`*S*`)` (property)

**Returns:** `true` or `false`.

`IsCongruenceFreeSemigroup` returns `true` if the semigroup *S* is a congruence-free semigroup and `false` if it is not.

A semigroup $S$ is *congruence-free* if it has no non-trivial proper congruences.

A semigroup with zero is congruence-free if and only if it is isomorphic to a regular Rees 0-matrix semigroup R whose underlying semigroup is the trivial group, no two rows of the matrix of R are identical, and no two columns are identical; see Theorem 3.7.1 in [How95].

A semigroup without zero is congruence-free if and only if it is a simple group or has order 2; see Theorem 3.7.2 in [How95].

Example

```
gap> S := Semigroup(Transformation([4, 2, 3, 3, 4]));;
gap> IsCongruenceFreeSemigroup(S);
true
gap> S := Semigroup(Transformation([2, 2, 4, 4]),
>                   Transformation([5, 3, 4, 4, 6, 6]));;
gap> IsCongruenceFreeSemigroup(S);
false
```

### 12.1.6 IsCryptoGroup

▷ IsCryptoGroup(*S*) (property)

**Returns:** true or false.

A semigroup $S$ is a *crypto group* if it is completely regular and Green's $\mathscr{H}$-relation is a congruence. IsCryptoGroup(*S*) returns true if the semigroup $S$ is a crypto group, and false if it is not.

Example

```
gap> G := SymmetricGroup(5);;
gap> M := [[(1, 2), (2, 4)], [(1, 4), (1, 2, 3, 4, 5)]];;
gap> R := ReesMatrixSemigroup(G, M);;
gap> IsCryptoGroup(R);
true
gap> S := FullTransformationMonoid(2);;
gap> IsCryptoGroup(S);
false
gap> G1 := AlternatingGroup(4);;
gap> G2 := SymmetricGroup(3);;
gap> G3 := AlternatingGroup(5);;
gap> gr := Digraph([[1, 3], [2, 3], [3]]);;
gap> sgn := function(x)
> if SignPerm(x) = 1 then
> return ();
> fi;
> return (1, 2);
> end;;
gap> hom13 := GroupHomomorphismByFunction(G1, G3, sgn);;
gap> hom23 := GroupHomomorphismByFunction(G2, G3, sgn);;
gap> S := AsSemigroup(IsPartialPermSemigroup,
> gr,
> [G1, G2, G3], [[1, 3, hom13], [2, 3, hom23]]);;
gap> IsCryptoGroup(S);
true
```

### 12.1.7 IsGroupAsSemigroup

▷ IsGroupAsSemigroup(*S*) (property)

**Returns:** true or false.

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

Example

```
gap> S := Semigroup(Transformation([2, 4, 5, 3, 7, 8, 6, 9, 1]),
>                   Transformation([3, 5, 6, 7, 8, 1, 9, 2, 4]));;
gap> IsGroupAsSemigroup(S);
true
gap> G := SymmetricGroup(5);;
gap> IsGroupAsSemigroup(G);
true
gap> S := AsSemigroup(IsPartialPermSemigroup, G);
<partial perm group of size 120, rank 5 with 2 generators>
gap> IsGroupAsSemigroup(S);
true
gap> G := SymmetricGroup([1, 2, 10]);;
gap> T := AsSemigroup(IsBlockBijectionSemigroup, G);
<inverse block bijection semigroup of size 6, degree 11 with 2
 generators>
gap> IsGroupAsSemigroup(T);
true
```

### 12.1.8 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 (11.10.1) and IdempotentGeneratedSubsemigroup (11.10.3).

An inverse semigroup is idempotent-generated if and only if it is a semilattice; see IsSemilattice (12.1.24).

The terms semiband and idempotent-generated are synonymous in this context.

Example

```
gap> S := SingularTransformationSemigroup(4);
<regular transformation semigroup ideal of degree 4 with 1 generator>
gap> IsIdempotentGenerated(S);
true
gap> S := SingularBrauerMonoid(5);
<regular bipartition *-semigroup ideal of degree 5 with 1 generator>
gap> IsIdempotentGenerated(S);
true
```

### 12.1.9 IsLeftSimple

▷ IsLeftSimple(*S*) (property)

▷ IsRightSimple(*S*) (property)

**Returns:** true or false.

IsLeftSimple and IsRightSimple return 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 (12.1.7).

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
```

### 12.1.10 IsLeftZeroSemigroup

▷ IsLeftZeroSemigroup(*S*) (property)

**Returns:** true or false.

IsLeftZeroSemigroup returns true if the semigroup $S$ is a left zero semigroup and false if it is not.

A semigroup is a *left zero semigroup* if x*y=x for all x,y. An inverse semigroup is a left zero semigroup if and only if it is trivial.

Example

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

```
gap> IsLeftZeroSemigroup(S);
true
```

### 12.1.11 IsMonogenicMonoid

▷ IsMonogenicMonoid(*S*) (property)

**Returns:** true or false.

IsMonogenicMonoid returns true if the monoid *S* is a monogenic monoid and it returns false if it is not.

A monoid is *monogenic* if it is generated as a monoid by a single element. See also IsMonogenicSemigroup (12.1.12) and IsMonogenicInverseMonoid (12.2.10).

Example

```
gap> x := PartialPerm([1, 2, 3, 6, 8, 10], [2, 6, 7, 9, 1, 5]);;
gap> S := Monoid(x, x ^ 2, x ^ 3);;
gap> IsMonogenicSemigroup(S);
false
gap> IsMonogenicMonoid(S);
true
gap> S := FullTransformationMonoid(5);;
gap> IsMonogenicMonoid(S);
false
```

### 12.1.12 IsMonogenicSemigroup

▷ IsMonogenicSemigroup(*S*) (property)

**Returns:** true or false.

IsMonogenicSemigroup returns true if the semigroup *S* is monogenic and it returns false if it is not.

A semigroup is *monogenic* if it is generated by a single element. See also IsMonogenicMonoid (12.1.11), IsMonogenicInverseSemigroup (12.2.9), and IsMonogenicInverseMonoid (12.2.10).

Example

```
gap> S := Semigroup(
> Transformation(
>  [2, 2, 2, 11, 10, 8, 10, 11, 2, 11, 10, 2, 11, 11, 10]),
> Transformation(
>  [2, 2, 2, 8, 11, 15, 11, 10, 2, 10, 11, 2, 10, 4, 7]),
> Transformation(
>  [2, 2, 2, 11, 10, 8, 10, 11, 2, 11, 10, 2, 11, 11, 10]),
> Transformation(
>  [2, 2, 12, 7, 8, 14, 8, 11, 2, 11, 10, 2, 11, 15, 4]));;
gap> IsMonogenicSemigroup(S);
true
gap> S := Semigroup(
> Bipartition([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -2, -5, -7, -9],
>              [-1, -10], [-3, -4, -6, -8]]),
>  Bipartition([[1, 4, 7, 8, -2], [2, 3, 5, 10, -5],
>               [6, 9, -7, -9], [-1, -10], [-3, -4, -6, -8]]));;
gap> IsMonogenicSemigroup(S);
true
gap> S := FullTransformationSemigroup(5);;
```

```
gap> IsMonogenicSemigroup(S);
false
```

### 12.1.13 IsMonoidAsSemigroup

▷ IsMonoidAsSemigroup(*S*) (property)

**Returns:** true or false.

IsMonoidAsSemigroup returns true if and only if the semigroup *S* is mathematically a monoid, i.e. if and only if it contains a MultiplicativeNeutralElement (**Reference: MultiplicativeNeutralElement**).

It is possible that a semigroup which satisfies IsMonoidAsSemigroup is not in the GAP category IsMonoid (**Reference: IsMonoid**). This is possible if the MultiplicativeNeutralElement (**Reference: MultiplicativeNeutralElement**) of *S* is not equal to the One (**Reference: One**) of any element in *S*. Therefore a semigroup satisfying IsMonoidAsSemigroup may not possess the attributes of a monoid (such as, GeneratorsOfMonoid (**Reference: GeneratorsOfMonoid**)).

See also One (**Reference: One**), IsInverseMonoid (**Reference: IsInverseMonoid**) and IsomorphismTransformationMonoid (**Reference: IsomorphismTransformationMonoid**).

Example

```
gap> S := Semigroup(Transformation([1, 4, 6, 2, 5, 3, 7, 8, 9, 9]),
>                   Transformation([6, 3, 2, 7, 5, 1, 8, 8, 9, 9]));;
gap> IsMonoidAsSemigroup(S);
true
gap> IsMonoid(S);
false
gap> MultiplicativeNeutralElement(S);
Transformation( [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 9 ] )
gap> T := AsSemigroup(IsBipartitionSemigroup, S);;
gap> IsMonoidAsSemigroup(T);
true
gap> IsMonoid(T);
false
gap> One(T);
fail
gap> S := Monoid(Transformation([8, 2, 8, 9, 10, 6, 2, 8, 7, 8]),
>                Transformation([9, 2, 6, 3, 6, 4, 5, 5, 3, 2]));;
gap> IsMonoidAsSemigroup(S);
true
```

### 12.1.14 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 (12.1.18) and IsRegularSemigroup (**Reference: IsRegularSemigroup**).

Example

```
gap> S := Semigroup(Transformation([1, 1, 1, 4, 5, 4]),
>                   Transformation([1, 2, 3, 1, 1, 2]),
>                   Transformation([1, 2, 3, 1, 1, 3]),
>                   Transformation([5, 5, 5, 5, 5, 5]));;
gap> IsOrthodoxSemigroup(S);
true
gap> S := DualSymmetricInverseMonoid(5);;
gap> S := Semigroup(GeneratorsOfSemigroup(S));;
gap> IsOrthodoxSemigroup(S);
true
```

### 12.1.15 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 (12.1.24).

A semigroup is *aperiodic* if it 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
```

### 12.1.16 IsRectangularBand

▷ IsRectangularBand(*S*) (property)

**Returns:** true or false.

IsRectangularBand returns true if the semigroup $S$ is a rectangular band and false if it is not.

A semigroup $S$ is a *rectangular band* if for all $x, y, z$ in $S$ we have that $x^2 = x$ and $xyz = xz$.

Equivalently, $S$ is a *rectangular band* if $S$ is isomorphic to a semigroup of the form $I \times \Lambda$ with multiplication $(i,\lambda)(j,\mu) = (i,\mu)$. In this case, $S$ is called an $|I| \times |\Lambda|$ *rectangular band*.

An inverse semigroup is a rectangular band if and only if it is a group.

Example

```
gap> S := Semigroup(
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 1]),
> Transformation([2, 2, 2, 5, 5, 5, 8, 8, 8, 2]),
> Transformation([3, 3, 3, 6, 6, 6, 9, 9, 9, 3]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 4]),
> Transformation([1, 1, 1, 4, 4, 4, 7, 7, 7, 7]));;
gap> IsRectangularBand(S);
true
gap> IsRectangularBand(MinimalIdeal(PartitionMonoid(4)));
true
```

### 12.1.17 IsRectangularGroup

▷ `IsRectangularGroup(`$S$`)` (property)

**Returns:** `true` or `false`.

A semigroup is *rectangular group* if it is the direct product of a group and a rectangular band. Or equivalently, if it is orthodox and simple.

Example

```
gap> G := AsSemigroup(IsTransformationSemigroup, MathieuGroup(11));
<transformation group of size 7920, degree 11 with 2 generators>
gap> R := RectangularBand(3, 2);
<regular transformation semigroup of size 6, degree 6 with 3
 generators>
gap> S := DirectProduct(G, R);;
gap> IsRectangularGroup(R);
true
gap> IsRectangularGroup(G);
true
gap> IsRectangularGroup(S);
true
gap> IsRectangularGroup(JonesMonoid(3));
false
```

### 12.1.18 IsRegularSemigroup

▷ `IsRegularSemigroup(`$S$`)` (property)

**Returns:** `true` or `false`.

`IsRegularSemigroup` returns `true` if the semigroup $S$ is regular and `false` if it is not.

A semigroup `S` is *regular* if for all `x` in `S` there exists `y` in `S` such that `x * y * x = x`. Every inverse semigroup is regular, and a semigroup of partial permutations is regular if and only if it is an inverse semigroup.

See also `IsRegularDClass` (**Reference: IsRegularDClass**), `IsRegularGreensClass` (10.3.2), and `IsRegularSemigroupElement` (**Reference: IsRegularSemigroupElement**).

Example

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

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

### 12.1.19 IsRightZeroSemigroup

▷ IsRightZeroSemigroup(*S*) (property)

**Returns:** true or false.

IsRightZeroSemigroup returns true if *S* is a right zero semigroup and false if it is not.

A semigroup S is a *right zero semigroup* if x * y = y for all x, y in S. An inverse semigroup is a right zero semigroup if and only if it is trivial.

Example

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

### 12.1.20 IsSelfDualSemigroup

▷ IsSelfDualSemigroup(*S*) (property)

**Returns:** true or false.

Returns true if the semigroup *S* is self dual and false otherwise.

A semigroup is *self dual* if it is isomorphic to its dual, that is, the semigroup T with multiplication * defined by x*y=yx where yx denotes the product in *S*.

Example

```
gap> F := FreeSemigroup("a", "b");
<free semigroup on the generators [ a, b ]>
gap> AssignGeneratorVariables(F);
gap> R := [[a ^ 3, a], [b ^ 2, b], [(a * b) ^ 2, a]];
[ [ a^3, a ], [ b^2, b ], [ (a*b)^2, a ] ]
gap> S := F / R;
<fp semigroup with 2 generators and 3 relations of length 14>
gap> IsSelfDualSemigroup(S);
false
gap> IsSelfDualSemigroup(FreeBand(3));
true
gap> S := DualSymmetricInverseMonoid(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> IsSelfDualSemigroup(S);
true
```

### 12.1.21 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 (11.8.3) of $S$).

If this is not the case, then either $S$ lacks a multiplicative zero, or the set $S \setminus \{0\}$ is not a subsemigroup of $S$, and so IsSemigroupWithAdjoinedZero returns false.

Example

```
gap> S := Semigroup(Transformation([2, 3, 4, 5, 1, 6]),
>                   Transformation([2, 1, 3, 4, 5, 6]),
>                   Transformation([6, 6, 6, 6, 6, 6]));
<transformation semigroup of degree 6 with 3 generators>
gap> IsZeroGroup(S);
true
gap> IsSemigroupWithAdjoinedZero(S);
true
gap> S := FullTransformationMonoid(4);;
gap> IsSemigroupWithAdjoinedZero(S);
false
```

### 12.1.22 IsSemigroupWithClosedIdempotents

▷ IsSemigroupWithClosedIdempotents(*S*) (property)

**Returns:** true or false.

IsSemigroupWithClosedIdempotents returns true if the idempotents of the semigroup $S$ form a subsemigroup of $S$, and false if they do not.

Example

```
gap> x := Transformation([2, 2]);;
gap> y := Transformation([1, 3, 3]);;
gap> S := Semigroup(x, y);
<transformation semigroup of degree 3 with 2 generators>
gap> IsSemigroupWithClosedIdempotents(S);
false
gap> S := FullTransformationMonoid(4);
<full transformation monoid of degree 4>
gap> IsSemigroupWithClosedIdempotents(S);
false
```

### 12.1.23 IsSemigroupWithCommutingIdempotents

▷ IsSemigroupWithCommutingIdempotents(*S*) (property)

**Returns:** true or false.

IsSemigroupWithCommutingIdempotents returns true if the idempotents of the semigroup $S$ commute, and false if they do not. Note that such a semigroup is also a block group; see IsBlockGroup (12.1.2).

Note that the idempotents commute in any inverse semigroup, in any semigroup of partial permutations, and in any semigroup of block bijections.

Example

```
gap> x := Transformation([2, 2]);;
gap> y := Transformation([1, 3, 3]);;
gap> S := Semigroup(x, y);
<transformation semigroup of degree 3 with 2 generators>
gap> IsSemigroupWithCommutingIdempotents(S);
```

```
false
gap> S := FullTransformationMonoid(4);
<full transformation monoid of degree 4>
gap> IsSemigroupWithCommutingIdempotents(S);
false
gap> W := Semigroup([
>  Bipartition([[1, 2, 3, -1, -2, -4], [4, -3]]),
>  Bipartition([[1, 2, -4], [3, -3], [4, -1, -2]])]);
<block bijection semigroup of degree 4 with 2 generators>
gap> IsSemigroupWithCommutingIdempotents(S);
false
```

### 12.1.24 IsSemilattice

▷ `IsSemilattice(`*S*`)` (property)

**Returns:** `true` or `false`.

`IsSemilattice` returns `true` if the semigroup *S* is a semilattice and `false` if it is not.

A semigroup is a *semilattice* if it is commutative and every element is an idempotent. The idempotents of an inverse semigroup form a semilattice.

Example

```
gap> S := Semigroup(Transformation([2, 5, 1, 7, 3, 7, 7]),
>                   Transformation([3, 6, 5, 7, 2, 1, 7]));;
gap> Size(S);
631
gap> IsInverseSemigroup(S);
true
gap> A := Semigroup(Idempotents(S));
<transformation semigroup of degree 7 with 32 generators>
gap> IsSemilattice(A);
true
gap> S := FactorisableDualSymmetricInverseMonoid(5);;
gap> S := IdempotentGeneratedSubsemigroup(S);;
gap> IsSemilattice(S);
true
```

### 12.1.25 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(())>
```

### 12.1.26 IsSurjectiveSemigroup

▷ `IsSurjectiveSemigroup(`*S*`)` (property)

**Returns:** `true` or `false`.

A semigroup is *surjective* if each of its elements can be written as a product of two elements in the semigroup. `IsSurjectiveSemigroup(`*S*`)` returns `true` if the semigroup *S* is surjective, and `false` if it is not.

See also `IndecomposableElements` (11.7.6).

Note that every monoid, and every regular semigroup, is surjective.

Example

```
gap> S := FullTransformationMonoid(100);
<full transformation monoid of degree 100>
gap> IsSurjectiveSemigroup(S);
true
gap> F := FreeSemigroup(3);;
gap> P := F / [[F.1, F.2 * F.1], [F.3 ^ 3, F.3]];
<fp semigroup with 3 generators and 2 relations of length 10>
gap> IsSurjectiveSemigroup(P);
false
gap> I := SingularTransformationMonoid(5);
<regular transformation semigroup ideal of degree 5 with 1 generator>
gap> IsSurjectiveSemigroup(I);
true
gap> M := MonogenicSemigroup(IsBipartitionSemigroup, 3, 2);
<commutative non-regular block bijection semigroup of size 4,
 degree 6 with 1 generator>
gap> IsSurjectiveSemigroup(M);
false
```

### 12.1.27 IsSynchronizingSemigroup (for a transformation semigroup)

▷ `IsSynchronizingSemigroup(`*S*`)` (property)

**Returns:** `true` or `false`.

For a positive integer *n*, `IsSynchronizingSemigroup` returns `true` if the semigroup of transformations *S* contains a transformation with constant value on `[1 .. `*n*`]` where `n` is the degree of the semigroup. See also `ConstantTransformation` (**Reference: ConstantTransformation**).

Note that the semigroup consisting of the identity transformation is the unique transformation semigroup with degree `0`. In this special case, the function `IsSynchronizingSemigroup` will return `false`.

Example

```
gap> S := Semigroup(
> Transformation([1, 1, 8, 7, 6, 6, 4, 1, 8, 9]),
> Transformation([5, 8, 7, 6, 10, 8, 7, 6, 9, 7]));;
gap> IsSynchronizingSemigroup(S);
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);
false
gap> Representative(MinimalIdeal(S));
Transformation( [ 8, 7, 7, 7, 8, 8, 7, 8, 8, 8 ] )
```

### 12.1.28 IsUnitRegularMonoid

▷ IsUnitRegularMonoid(*S*) (property)

**Returns:** `true` if the semigroup *S* is unit regular and `false` if it is not.

A monoid is *unit regular* if and only if for every `x` in *S* there exists an element `y` in the group of units of *S* such that `x*y*x=x`.

Example

```
gap> IsUnitRegularMonoid(FullTransformationMonoid(3));
true
```

### 12.1.29 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
```

### 12.1.30 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 (12.1.32) and IsHTrivial (12.1.15). An inverse semigroup is a 0-rectangular band if and only if it is a 0-group; see IsZeroGroup (12.1.29).

Example

```
gap> S := Semigroup(
>  Transformation([1, 3, 7, 9, 1, 12, 13, 1, 15, 9, 1, 18, 1, 1, 13,
>                  1, 1, 21, 1, 1, 1, 1, 1, 25, 26, 1]),
> Transformation([1, 5, 1, 5, 11, 1, 1, 14, 1, 16, 17, 1, 1, 19, 1,
>                 11, 1, 1, 1, 23, 1, 16, 19, 1, 1, 1]),
> Transformation([1, 4, 8, 1, 10, 1, 8, 1, 1, 1, 10, 1, 8, 10, 1, 1,
>                 20, 1, 22, 1, 8, 1, 1, 1, 1, 1]),
> Transformation([1, 6, 6, 1, 1, 1, 6, 1, 1, 1, 1, 1, 6, 1, 6, 1, 1,
>                 6, 1, 1, 24, 1, 1, 1, 1, 6]));;
gap> D := DClass(S,
> Transformation([1, 8, 1, 1, 8, 1, 1, 1, 1, 1, 8, 1, 1, 8, 1, 1, 1,
>                 1, 1, 1, 1, 1, 1, 1, 1, 1]));;
gap> IsZeroRectangularBand(Semigroup(D));
true
gap> IsZeroRectangularBand(Semigroup(GreensDClasses(S)[1]));
false
```

### 12.1.31 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 ] )
```

### 12.1.32 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 (11.8.3). An inverse semigroup is 0-simple if and only if it is a Brandt semigroup; see IsBrandtSemigroup (12.2.2).

Example

```
gap> S := Semigroup(
>  Transformation([1, 17, 17, 17, 17, 17, 17, 17, 17, 17, 5, 17,
>                  17, 17, 17, 17, 17]),
>  Transformation([1, 17, 17, 17, 11, 17, 17, 17, 17, 17, 17, 17,
>                  17, 17, 17, 17, 17]),
>  Transformation([1, 17, 17, 17, 17, 17, 17, 17, 17, 17, 4, 17,
>                  17, 17, 17, 17, 17]),
>  Transformation([1, 17, 17, 5, 17, 17, 17, 17, 17, 17, 17, 17,
>                  17, 17, 17, 17, 17]));;
gap> IsZeroSimpleSemigroup(S);
true
gap> S := Semigroup(
> Transformation([2, 3, 4, 5, 1, 8, 7, 6, 2, 7]),
> Transformation([2, 3, 4, 5, 6, 8, 7, 1, 2, 2]));;
gap> IsZeroSimpleSemigroup(S);
false
```

## 12.2 Inverse semigroups

In this section we describe the properties of an inverse semigroup or monoid that can be determined using the Semigroups package.

### 12.2.1 IsCliffordSemigroup

▷ IsCliffordSemigroup(*S*) (property)

**Returns:** true or false.

IsCliffordSemigroup returns true if the semigroup *S* is regular and its idempotents are central, and false if it is not.

Example

```
gap> S := Semigroup(Transformation([1, 2, 4, 5, 6, 3, 7, 8]),
>                   Transformation([3, 3, 4, 5, 6, 2, 7, 8]),
>                   Transformation([1, 2, 5, 3, 6, 8, 4, 4]));;
gap> IsCliffordSemigroup(S);
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsCliffordSemigroup(S);
true
gap> S := DualSymmetricInverseMonoid(5);;
gap> T := IdempotentGeneratedSubsemigroup(S);;
gap> IsCliffordSemigroup(T);
true
```

### 12.2.2 IsBrandtSemigroup

▷ IsBrandtSemigroup(*S*) (property)

**Returns:** true or false.

IsBrandtSemigroup returns true if the semigroup $S$ is a finite 0-simple inverse semigroup, and false if it is not. See also IsZeroSimpleSemigroup (12.1.32) and IsInverseSemigroup (**Reference: IsInverseSemigroup**).

Example

```
gap> S := Semigroup(
> Transformation([2, 8, 8, 8, 8, 8, 8, 8]),
> Transformation([5, 8, 8, 8, 8, 8, 8, 8]),
> Transformation([8, 3, 8, 8, 8, 8, 8, 8]),
> Transformation([8, 6, 8, 8, 8, 8, 8, 8]),
> Transformation([8, 8, 1, 8, 8, 8, 8, 8]),
> Transformation([8, 8, 8, 1, 8, 8, 8, 8]),
> Transformation([8, 8, 8, 8, 4, 8, 8, 8]),
> Transformation([8, 8, 8, 8, 8, 7, 8, 8]),
> Transformation([8, 8, 8, 8, 8, 8, 2, 8]));;
gap> IsBrandtSemigroup(S);
true
gap> T := Range(IsomorphismPartialPermSemigroup(S));;
gap> IsBrandtSemigroup(T);
true
gap> S := DualSymmetricInverseMonoid(4);;
gap> D := DClass(S,
>                Bipartition([[1, 2, 3, -1, -2, -3], [4, -4]]));;
gap> R := InjectionPrincipalFactor(D);;
gap> S := Semigroup(PreImages(R, GeneratorsOfSemigroup(Range(R))));;
gap> IsBrandtSemigroup(S);
true
```

### 12.2.3 IsEUnitaryInverseSemigroup

▷ IsEUnitaryInverseSemigroup($S$) (property)

**Returns:** true or false.

As described in Section 5.9 of [How95], an inverse semigroup $S$ with semilattice of idempotents $E$ is $E$-*unitary* if for

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

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

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

This condition is equivalent to $E$ being majorantly closed in $S$. See IdempotentGeneratedSubsemigroup (11.10.3) and IsMajorantlyClosed (12.2.8). 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"
```

### 12.2.4 IsFInverseSemigroup

▷ `IsFInverseSemigroup(`*S*`)` (property)

**Returns:** `true` or `false`.

This function determines whether a given semigroup is an F-inverse semigroup. An F-inverse semigroup is a semigroup which satisfies `IsEUnitaryInverseSemigroup` (12.2.3) as well as being isomorphic to some `McAlisterTripleSemigroup` (8.4.2) where the `McAlisterTripleSemigroupPartialOrder` (8.4.4) satisfies `IsJoinSemilatticeDigraph` (**Digraphs: IsJoinSemilatticeDigraph**). McAlister triple semigroups are a representation of E-unitary inverse semigroups and more can be read about them in Chapter 8.4.

Example

```
gap> S := InverseMonoid([PartialPermNC([1, 2], [1, 2]),
> PartialPermNC([1, 2, 3], [1, 2, 3]),
> PartialPermNC([1, 2, 4], [1, 2, 4]),
> PartialPermNC([1, 2], [2, 1]), PartialPermNC([1, 2, 3], [2, 1, 3]),
> PartialPermNC([1, 2, 4], [2, 1, 4])]);;
gap> IsEUnitaryInverseSemigroup(S);
true
gap> IsFInverseSemigroup(S);
false
gap> IsFInverseSemigroup(IdempotentGeneratedSubsemigroup(S));
true
```

### 12.2.5 IsFInverseMonoid

▷ `IsFInverseMonoid(`*S*`)` (property)

**Returns:** `true` or `false`.

This function determines whether a given semigroup is an F-inverse monoid. A semigroup is an F-inverse monoid when it satisfies `IsMonoid` (**Reference: IsMonoid**) and `IsFInverseSemigroup` (12.2.4).

### 12.2.6 IsFactorisableInverseMonoid

▷ IsFactorisableInverseMonoid(*S*) (property)

**Returns:** true or false.

An inverse monoid is *factorisable* if every element is the product of an element of the group of units and an idempotent; see also GroupOfUnits (11.9.1) and Idempotents (11.10.1). Hence an inverse semigroup of partial permutations is factorisable if and only if each of its generators is the restriction of some element in the group of units.

Example

```
gap> S := InverseSemigroup(
> PartialPerm([1, 2, 4], [3, 1, 4]),
> PartialPerm([1, 2, 3, 5], [4, 1, 5, 2]));;
gap> IsFactorisableInverseMonoid(S);
false
gap> IsFactorisableInverseMonoid(SymmetricInverseSemigroup(5));
true
gap> IsFactorisableInverseMonoid(DualSymmetricInverseMonoid(5));
false
gap> S := FactorisableDualSymmetricInverseMonoid(5);;
gap> IsFactorisableInverseMonoid(S);
true
```

### 12.2.7 IsJoinIrreducible

▷ IsJoinIrreducible(*S*, *x*) (operation)

**Returns:** true or false.

IsJoinIrreducible determines whether an element *x* of an inverse semigroup *S* of partial permutations, block bijections or partial permutation bipartitions is join irreducible.

An element *x* is *join irreducible* when it is not the least upper bound (with respect to the natural partial order NaturalLeqPartialPerm (**Reference: NaturalLeqPartialPerm**)) of any subset of *S* not containing *x*.

Example

```
gap> S := SymmetricInverseSemigroup(3);
<symmetric inverse monoid of degree 3>
gap> x := PartialPerm([1, 2, 3]);
<identity partial perm on [ 1, 2, 3 ]>
gap> IsJoinIrreducible(S, x);
false
gap> T := InverseSemigroup([
>  PartialPerm([1, 2, 4, 3]),
>  PartialPerm([1]),
>  PartialPerm([0, 2])]);
<inverse partial perm semigroup of rank 4 with 3 generators>
gap> y := PartialPerm([1, 2, 3, 4]);
<identity partial perm on [ 1, 2, 3, 4 ]>
gap> IsJoinIrreducible(T, y);
true
gap> B := InverseSemigroup([
>  Bipartition([
>    [1, -5], [2, -2], [3, 5, 6, 7, -1, -4, -6, -7], [4, -3]]),
>  Bipartition([
```

```
>    [1, -1], [2, -3], [3, -4], [4, 5, 7, -2, -6, -7], [6, -5]]),
>  Bipartition([
>    [1, -2], [2, -4], [3, -6], [4, -1], [5, 7, -3, -7], [6, -5]]),
>  Bipartition([
>    [1, -5], [2, -1], [3, -6], [4, 5, 7, -2, -4, -7], [6, -3]])]);
<inverse block bijection semigroup of degree 7 with 4 generators>
gap> x := Bipartition([
>   [1, 2, 3, 5, 6, 7, -2, -3, -4, -5, -6, -7], [4, -1]]);
<block bijection: [ 1, 2, 3, 5, 6, 7, -2, -3, -4, -5, -6, -7 ],
 [ 4, -1 ]>
gap> IsJoinIrreducible(B, x);
true
gap> IsJoinIrreducible(B, B.1);
false
```

### 12.2.8 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` (11.15.3).

We say that *T* is *majorantly closed* in *S* if it contains all elements of *S* which are greater than or equal to any element of *T*, with respect to the natural partial order. See `NaturalLeqPartialPerm` (**Reference: NaturalLeqPartialPerm**).

Note that *T* can be a subset of *S* or a subsemigroup of *S*.

Example

```
gap> S := SymmetricInverseSemigroup(2);
<symmetric inverse monoid of degree 2>
gap> T := [Elements(S)[2]];
[ <identity partial perm on [ 1 ]> ]
gap> IsMajorantlyClosed(S, T);
false
gap> U := [Elements(S)[2], Elements(S)[6]];
[ <identity partial perm on [ 1 ]>, <identity partial perm on [ 1, 2 ]
    > ]
gap> IsMajorantlyClosed(S, U);
true
gap> D := DualSymmetricInverseSemigroup(3);
<inverse block bijection monoid of degree 3 with 3 generators>
gap> x := Bipartition([[1, -2], [2, -3], [3, -1]]);;
gap> IsMajorantlyClosed(D, [x]);
true
gap> y := Bipartition([[1, 2, -1, -2], [3, -3]]);;
gap> IsMajorantlyClosed(D, [x, y]);
false
```

### 12.2.9 IsMonogenicInverseSemigroup

▷ IsMonogenicInverseSemigroup(*S*) (property)

**Returns:** true or false.

IsMonogenicInverseSemigroup returns true if the semigroup *S* is a monogenic inverse semigroup and it returns false if it is not.

An inverse semigroup is *monogenic* if it is generated as an inverse semigroup by a single element. See also IsMonogenicSemigroup (12.1.12) and IsMonogenicInverseMonoid (12.2.10).

Example

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

### 12.2.10 IsMonogenicInverseMonoid

▷ IsMonogenicInverseMonoid(*S*) (property)

**Returns:** true or false.

IsMonogenicInverseMonoid returns true if the monoid *S* is a monogenic inverse monoid and it returns false if it is not.

An inverse monoid is *monogenic* if it is generated as an inverse monoid by a single element. See also IsMonogenicInverseSemigroup (12.2.9) and IsMonogenicMonoid (12.1.11).

Example

```
gap> x := PartialPerm([1, 2, 3, 6, 8, 10], [2, 6, 7, 9, 1, 5]);;
gap> S := InverseMonoid(x, x ^ 2, x ^ 3);;
gap> IsMonogenicMonoid(S);
false
gap> IsMonogenicInverseSemigroup(S);
false
gap> IsMonogenicInverseMonoid(S);
true
gap> x := RandomBlockBijection(100);;
gap> S := InverseMonoid(x, x ^ 2, x ^ 20);;
gap> IsMonogenicInverseMonoid(S);
true
gap> S := SymmetricInverseMonoid(5);;
gap> IsMonogenicInverseMonoid(S);
false
```

# Chapter 13

# 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
- Word graph -- only for congruences created via `IteratorOfLeftCongruences` (13.4.15) or `IteratorOfRightCongruences` (13.4.15)
- Wang pairs -- only for graph inverse semigroup

The operation `SemigroupCongruence` (13.2.1) can be used to create any of these, interpreting the arguments in a smart way. The usual way of specifying a congruence will be by giving a set of generating pairs, but a user with an ideal could instead create a Rees congruence or universal congruence.

If a congruence is specified by generating pairs on a simple, 0-simple, or inverse semigroup, then the congruence may be converted automatically to one of the last two items in the above list, to reduce the complexity of any calculations to be performed. The user need not manually specify, or even be aware of, the congruence's linked triple or kernel and trace.

We can also create left congruences and right congruences, using the `LeftSemigroupCongruence` (13.2.2) and `RightSemigroupCongruence` (13.2.3) functions.

Please note that congruence objects made in GAP before loading the Semigroups package may not behave correctly after Semigroups is loaded. If Semigroups is loaded at the beginning of the session, or before any congruence work is done, then the objects should behave correctly.

## 13.1 Semigroup congruence objects

### 13.1.1 IsSemigroupCongruence

▷ `IsSemigroupCongruence(`*obj*`)` (property)

A semigroup congruence cong is an equivalence relation on a semigroup S which respects left and right multiplication.

That is, if $(a,b)$ is a pair in cong, and $x$ is an element of S, then $(ax,bx)$ and $(xa,xb)$ are both in cong.

The simplest way of creating a congruence in Semigroups is by a set of *generating pairs*. See SemigroupCongruence (13.2.1).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := SemigroupCongruence(S, [pair1, pair2]);
<semigroup congruence over <simple transformation semigroup of
 degree 5 with 4 generators> with linked triple (2,4,1)>
gap> IsSemigroupCongruence(cong);
true
```

### 13.1.2 IsLeftSemigroupCongruence

▷ IsLeftSemigroupCongruence(*obj*) (property)

A left semigroup congruence cong is an equivalence relation on a semigroup S which respects left multiplication.

That is, if $(a,b)$ is a pair in cong, and $x$ is an element of S, then $(xa,xb)$ is also in cong.

The simplest way of creating a left congruence in Semigroups is by a set of *generating pairs*. See LeftSemigroupCongruence (13.2.2).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := LeftSemigroupCongruence(S, [pair1, pair2]);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
gap> IsLeftSemigroupCongruence(cong);
true
```

### 13.1.3 IsRightSemigroupCongruence

▷ IsRightSemigroupCongruence(*obj*) (property)

A right semigroup congruence cong is an equivalence relation on a semigroup S which respects right multiplication.

That is, if $(a,b)$ is a pair in cong, and $x$ is an element of S, then $(ax,bx)$ is also in cong.

The simplest way of creating a right congruence in Semigroups is by a set of *generating pairs*. See RightSemigroupCongruence (13.2.3).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := RightSemigroupCongruence(S, [pair1, pair2]);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
gap> IsRightSemigroupCongruence(cong);
true
```

## 13.2 Creating congruences

### 13.2.1 SemigroupCongruence

▷ SemigroupCongruence(*S*, *pairs*) (function)

**Returns:** A semigroup congruence.

This function returns a semigroup congruence over the semigroup *S*.

If *pairs* is a list of lists of size 2 with elements from *S*, then this function will return the semigroup congruence defined by these generating pairs. The individual pairs may instead be given as separate arguments.

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> SemigroupCongruence(S, [pair1, pair2]);
<semigroup congruence over <simple transformation semigroup of
 degree 5 with 4 generators> with linked triple (2,4,1)>
gap> SemigroupCongruence(S, pair1, pair2);
```

```
<semigroup congruence over <simple transformation semigroup of
 degree 5 with 4 generators> with linked triple (2,4,1)>
```

### 13.2.2 LeftSemigroupCongruence

▷ `LeftSemigroupCongruence(`*S*`, `*pairs*`)` (function)

**Returns:** A left semigroup congruence.

This function returns a left semigroup congruence over the semigroup *S*.

If *pairs* is a list of lists of size 2 with elements from *S*, then this function will return the least left semigroup congruence on *S* which contains these generating pairs. The individual pairs may instead be given as separate arguments.

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> LeftSemigroupCongruence(S, [pair1, pair2]);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
gap> LeftSemigroupCongruence(S, pair1, pair2);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
```

### 13.2.3 RightSemigroupCongruence

▷ `RightSemigroupCongruence(`*S*`, `*pairs*`)` (function)

**Returns:** A right semigroup congruence.

This function returns a right semigroup congruence over the semigroup *S*.

If *pairs* is a list of lists of size 2 with elements from *S*, then this function will return the least right semigroup congruence on *S* which contains these generating pairs. The individual pairs may instead be given as separate arguments.

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> RightSemigroupCongruence(S, [pair1, pair2]);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
```

```
gap> RightSemigroupCongruence(S, pair1, pair2);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
```

## 13.3 Congruence classes

The main operations and attributes for congruences in the GAP library are:

- `EquivalenceClasses` (**Reference: EquivalenceClasses attribute**)
- `NrEquivalenceClasses`
- `EquivalenceClassOfElement` (**Reference: EquivalenceClassOfElement**)

### 13.3.1 IsCongruenceClass

▷ `IsCongruenceClass(`*obj*`)` (category)

This category contains any object which is an equivalence class of a semigroup congruence (see `IsSemigroupCongruence` (13.1.1)). An object will only be in this category if the relation is known to be a semigroup congruence when the congruence class is created.

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S, [Transformation([1, 2, 1]),
>                                    Transformation([2, 1, 2])]);;
gap> class := EquivalenceClassOfElement(cong,
>                                       Transformation([3, 1, 1]));
<2-sided congruence class of Transformation( [ 3, 1, 1 ] )>
gap> IsCongruenceClass(class);
true
```

### 13.3.2 IsLeftCongruenceClass

▷ `IsLeftCongruenceClass(`*obj*`)` (category)

This category contains any object which is an equivalence class of a left semigroup congruence (see `IsLeftSemigroupCongruence` (13.1.2)). An object will only be in this category if the relation is known to be a left semigroup congruence when the class is created.

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> pairs := [Transformation([1, 2, 1]),
>              Transformation([2, 1, 2])];;
gap> cong := LeftSemigroupCongruence(S, pairs);;
gap> class := EquivalenceClassOfElement(cong,
>                                       Transformation([3, 1, 1]));
<left congruence class of Transformation( [ 3, 1, 1 ] )>
gap> IsLeftCongruenceClass(class);
true
```

### 13.3.3 IsRightCongruenceClass

▷ IsRightCongruenceClass(*obj*) (category)

This category contains any object which is an equivalence class of a right semigroup congruence (see IsRightSemigroupCongruence (13.1.3)). An object will only be in this category if the relation is known to be a right semigroup congruence when the class is created.

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> pairs := [Transformation([1, 2, 1]),
>              Transformation([2, 1, 2])];;
gap> cong := RightSemigroupCongruence(S, pairs);;
gap> class := EquivalenceClassOfElement(cong,
>                                       Transformation([3, 1, 1]));
<right congruence class of Transformation( [ 3, 1, 1 ] )>
gap> IsRightCongruenceClass(class);
true
```

### 13.3.4 NonTrivialEquivalenceClasses

▷ NonTrivialEquivalenceClasses(*eq*) (attribute)

**Returns:** A list of equivalence classes.

If *eq* is an equivalence relation, then this attribute returns a list of all equivalence classes of *eq* which contain more than one element.

Example

```
gap> S := Monoid([Transformation([1, 2, 2]),
>                 Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S, [Transformation([1, 2, 1]),
>                                    Transformation([2, 1, 2])]);;
gap> classes := NonTrivialEquivalenceClasses(cong);;
gap> Set(classes);
[ <2-sided congruence class of Transformation( [ 3, 3, 3 ] )>,
  <2-sided congruence class of Transformation( [ 2, 1, 2 ] )>,
  <2-sided congruence class of Transformation( [ 1, 2, 2 ] )>,
  <2-sided congruence class of Transformation( [ 3, 1, 3 ] )>,
  <2-sided congruence class of Transformation( [ 3, 1, 1 ] )> ]
gap> cong := RightSemigroupCongruence(S, [Transformation([1, 2, 1]),
>                                         Transformation([2, 1, 2])]);;
gap> classes := NonTrivialEquivalenceClasses(cong);;
gap> Set(classes);
[ <right congruence class of Transformation( [ 2, 1, 2 ] )>,
  <right congruence class of Transformation( [ 3, 1, 3 ] )> ]
```

### 13.3.5 EquivalenceRelationLookup (for an equivalence relation over a finite semigroup)

▷ EquivalenceRelationLookup(*equiv*) (attribute)

**Returns:** A list.

This attribute describes the equivalence relation *equiv*, defined over a finite semigroup, as a list of positive integers of length equal to the size of the finite semigroup over which *equiv* is defined.

Each position in the list corresponds to an element of the semigroup (in a consistent canonical order) and the integer at that position is a unique identifier for that element's equivalence class under *equiv*. Two elements of the semigroup on which the equivalence is defined are related in the equivalence if and only if they have the same number at their respective positions in the lookup.

Note that the order in which numbers appear in the list is non-deterministic, and two equivalence relations describing the same mathematical relation might therefore have different lookups. Note also that the maximum value of the list may not be the number of classes of *equiv*, and that any integer might not be included. However, see EquivalenceRelationCanonicalLookup (13.3.6).

See also EquivalenceRelationPartition (**Reference: EquivalenceRelationPartition**).

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S,
> [Transformation([1, 2, 1]), Transformation([2, 1, 2])]);;
gap> lookup := EquivalenceRelationLookup(cong);;
gap> lookup[3] = lookup[8];
true
gap> lookup[2] = lookup[9];
false
```

### 13.3.6 EquivalenceRelationCanonicalLookup (for an equivalence relation over a finite semigroup)

▷ EquivalenceRelationCanonicalLookup(*equiv*) (attribute)

**Returns:** A list.

This attribute describes the equivalence relation *equiv*, defined over a finite semigroup, as a list of positive integers of length equal to the size of the semigroup.

Each position in the list corresponds to an element of the semigroup (in a consistent canonical order as defined by PositionCanonical (11.1.2)) and the integer at that position is a unique identifier for that element's equivalence class under *equiv*. The value of EquivalenceRelationCanonicalLookup has the property that the first appearance of the value i is strictly later than the first appearance of i-1, and that all entries in the list will be from the range [1 .. NrEquivalenceClasses(*equiv*)]. As such, two equivalence relations on a given semigroup are equal if and only if their canonical lookups are equal.

Two elements of the semigroup on which the equivalence relation is defined are related in the equivalence relation if and only if they have the same number at their respective positions in the lookup.

See also EquivalenceRelationLookup (13.3.5) and EquivalenceRelationPartition (**Reference: EquivalenceRelationPartition**).

Example

```
gap> S := Monoid([
>  Transformation([1, 2, 2]), Transformation([3, 1, 3])]);;
gap> cong := SemigroupCongruence(S,
> [Transformation([1, 2, 1]), Transformation([2, 1, 2])]);;
gap> EquivalenceRelationCanonicalLookup(cong);
[ 1, 2, 3, 4, 5, 6, 2, 3, 6, 4, 5, 6 ]
```

### 13.3.7 EquivalenceRelationCanonicalPartition

▷ EquivalenceRelationCanonicalPartition(*cong*) (attribute)

**Returns:** A list of lists.

This attribute returns a list of lists of elements of the underlying set of the semigroup congruence *cong*. These lists are precisely the nontrivial equivalence classes of *cong*. The order in which the classes appear is deterministic, and the order of the elements inside each class is also deterministic. Hence, two congruence objects have the same EquivalenceRelationCanonicalPartition if and only if they describe the same relation.

See also EquivalenceRelationPartition (**Reference: EquivalenceRelationPartition**), a similar attribute which does not have canonical ordering, but which is likely to be faster.

Example

```
gap> S := Semigroup(Transformation([1, 4, 3, 3]),
>                   Transformation([2, 4, 3, 3]));;
gap> cong := SemigroupCongruence(S, [Transformation([1, 4, 3, 3]),
>                                    Transformation([1, 3, 3, 3])]);;
gap> EquivalenceRelationCanonicalPartition(cong);
[ [ Transformation( [ 1, 4, 3, 3 ] ),
      Transformation( [ 1, 3, 3, 3 ] ) ],
  [ Transformation( [ 4, 3, 3, 3 ] ),
      Transformation( [ 3, 3, 3, 3 ] ) ] ]
```

### 13.3.8 OnLeftCongruenceClasses

▷ OnLeftCongruenceClasses(*class*, *elm*) (operation)

**Returns:** A left congruence class.

If *class* is an equivalence class of the left semigroup congruence cong on the semigroup S, and *elm* is an element of S, then this operation returns the equivalence class of cong containing the element *elm* * x, where x is any element of *class*. The result is well-defined by the definition of a left congruence.

See IsLeftSemigroupCongruence (13.1.2) and IsLeftCongruenceClass (13.3.2).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := LeftSemigroupCongruence(S, [pair1, pair2]);
<left semigroup congruence over <transformation semigroup of degree 5
 with 4 generators> with 2 generating pairs>
gap> x := Transformation([3, 4, 3, 4, 3]);;
gap> class := EquivalenceClassOfElement(cong, x);
<left congruence class of Transformation( [ 3, 4, 3, 4, 3 ] )>
gap> elm := Transformation([1, 2, 2, 1, 2]);;
gap> OnLeftCongruenceClasses(class, elm);
<left congruence class of Transformation( [ 3, 4, 4, 3, 4 ] )>
```

### 13.3.9 OnRightCongruenceClasses

▷ OnRightCongruenceClasses(*class, elm*) (operation)

**Returns:** A right congruence class.

If *class* is an equivalence class of the right semigroup congruence cong on the semigroup S, and *elm* is an element of S, then this operation returns the equivalence class of cong containing the element x * *elm*, where x is any element of *class*. The result is well-defined by the definition of a right congruence.

See IsRightSemigroupCongruence (13.1.3) and IsRightCongruenceClass (13.3.3).

Example

```
gap> S := Semigroup([
>   Transformation([2, 1, 1, 2, 1]),
>   Transformation([3, 4, 3, 4, 4]),
>   Transformation([3, 4, 3, 4, 3]),
>   Transformation([4, 3, 3, 4, 4])]);;
gap> pair1 := [Transformation([3, 4, 3, 4, 3]),
>              Transformation([1, 2, 1, 2, 1])];;
gap> pair2 := [Transformation([4, 3, 4, 3, 4]),
>              Transformation([3, 4, 3, 4, 3])];;
gap> cong := RightSemigroupCongruence(S, [pair1, pair2]);
<right semigroup congruence over <transformation semigroup of
 degree 5 with 4 generators> with 2 generating pairs>
gap> x := Transformation([3, 4, 3, 4, 3]);;
gap> class := EquivalenceClassOfElement(cong, x);
<right congruence class of Transformation( [ 3, 4, 3, 4, 3 ] )>
gap> elm := Transformation([1, 2, 2, 1, 2]);;
gap> OnRightCongruenceClasses(class, elm);
<right congruence class of Transformation( [ 2, 1, 2, 1, 2 ] )>
```

## 13.4 Finding the congruences of a semigroup

### 13.4.1 CongruencesOfSemigroup (for a semigroup)

▷ CongruencesOfSemigroup(*S*) (attribute)
▷ LeftCongruencesOfSemigroup(*S*) (attribute)
▷ RightCongruencesOfSemigroup(*S*) (attribute)
▷ CongruencesOfSemigroup(*S, restriction*) (operation)
▷ LeftCongruencesOfSemigroup(*S, restriction*) (operation)
▷ RightCongruencesOfSemigroup(*S, restriction*) (operation)

**Returns:** The congruences of a semigroup.

This attribute gives a list of the left, right, or 2-sided congruences of the semigroup *S*.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See also LatticeOfCongruences (13.4.5).

Example

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

```
gap> Length(congs);
4
gap> Set(congs, NrEquivalenceClasses);
[ 1, 5, 9, 25 ]
gap> pos := Position(congs, UniversalSemigroupCongruence(S));;
gap> congs[pos];
<universal semigroup congruence over
<Rees 0-matrix semigroup 2x2 over Sym( [ 1 .. 3 ] )>>
```

### 13.4.2 MinimalCongruencesOfSemigroup (for a semigroup)

▷ MinimalCongruencesOfSemigroup(*S*) (attribute)
▷ MinimalLeftCongruencesOfSemigroup(*S*) (attribute)
▷ MinimalRightCongruencesOfSemigroup(*S*) (attribute)
▷ MinimalCongruencesOfSemigroup(*S*, *restriction*) (operation)
▷ MinimalLeftCongruencesOfSemigroup(*S*, *restriction*) (operation)
▷ MinimalRightCongruencesOfSemigroup(*S*, *restriction*) (operation)

**Returns:** The congruences of a semigroup.

If *S* is a semigroup, then the attribute MinimalCongruencesOfSemigroup gives a list of all the congruences on *S* which are *minimal*. A congruence is minimal iff it is non-trivial and contains no other congruences as subrelations (apart from the trivial congruence).

MinimalLeftCongruencesOfSemigroup and MinimalRightCongruencesOfSemigroup do the same thing, but for left congruences and right congruences respectively. Note that any congruence is also a left congruence, but that a minimal congruence may not be a minimal left congruence.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See also CongruencesOfSemigroup (13.4.1) and PrincipalCongruencesOfSemigroup (13.4.3).

Example

```
gap> S := Semigroup(Transformation([1, 3, 2]),
>                   Transformation([3, 1, 3]));;
gap> min := MinimalCongruencesOfSemigroup(S);
[ <2-sided semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>
 ]
gap> minl := MinimalLeftCongruencesOfSemigroup(S);
[ <left semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <left semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <left semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>
 ]
```

### 13.4.3 PrincipalCongruencesOfSemigroup (for a semigroup)

▷ PrincipalCongruencesOfSemigroup(*S*) (attribute)
▷ PrincipalLeftCongruencesOfSemigroup(*S*) (attribute)

▷ PrincipalRightCongruencesOfSemigroup(*S*) (attribute)
▷ PrincipalCongruencesOfSemigroup(*S*, *restriction*) (operation)
▷ PrincipalLeftCongruencesOfSemigroup(*S*, *restriction*) (operation)
▷ PrincipalRightCongruencesOfSemigroup(*S*, *restriction*) (operation)

**Returns:** A list.

If *S* is a semigroup, then the attribute PrincipalCongruencesOfSemigroup gives a list of all the congruences on *S* which are *principal*. A congruence is principal if and only if it is non-trivial and can be defined by a single generating pair.

PrincipalLeftCongruencesOfSemigroup and PrincipalRightCongruencesOfSemigroup do the same thing, but for left congruences and right congruences respectively. Note that any congruence is a left congruence and a right congruence, but that a principal congruence may not be a principal left congruence or a principal right congruence.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See also CongruencesOfSemigroup (13.4.1) and MinimalCongruencesOfSemigroup (13.4.2).

Example

```
gap> S := Semigroup(Transformation([1, 3, 2]),
>                   Transformation([3, 1, 3]));;
gap> congs := PrincipalCongruencesOfSemigroup(S);
[ <universal semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators>>,
  <2-sided semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <2-sided semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <2-sided semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>,
  <2-sided semigroup congruence over <transformation semigroup
     of size 13, degree 3 with 2 generators> with 1 generating pairs>
 ]
```

### 13.4.4 IsCongruencePoset

▷ IsCongruencePoset(*poset*) (category)
▷ IsCayleyDigraphOfCongruences(*poset*) (category)

**Returns:** true or false.

This category contains all congruence posets. A *congruence poset* is a partially ordered set of congruences over a specific semigroup, where the ordering is defined by containment according to IsSubrelation (13.5.1): given two congruences cong1 and cong2, we say that cong1 < cong2 if and only if cong1 is a subrelation (a refinement) of cong2. The congruences in a congruence poset can be left, right, or two-sided.

A congruence poset is a digraph (see IsDigraph (**Digraphs: IsDigraph**)) with a vertex for each congruence, and an edge from vertex i to vertex j only if the congruence numbered i is a subrelation of the congruence numbered j. To avoid using an unnecessarily large amount of memory in some cases, a congruence poset does not necessarily belong to IsPartialOrderDigraph (**Digraphs: IsPartialOrderDigraph**). In other words, although every congruence poset represents a partial order it

is not necessarily the case that there is an edge from vertex `i` to vertex `j` if and only if the congruence numbered `i` is a subrelation of the congruence numbered `j`.

The list of congruences can be obtained using `CongruencesOfPoset` (13.4.8); and the underlying semigroup of the poset can be obtained using `UnderlyingSemigroupOfCongruencePoset` (13.4.9).

Congruence posets can be created using any of:

- `PosetOfCongruences` (13.4.10),
- `JoinSemilatticeOfCongruences` (13.4.11)
- `LatticeOfCongruences` (13.4.5), `LatticeOfLeftCongruences` (13.4.5), or `LatticeOfRightCongruences` (13.4.5)
- `CayleyDigraphOfCongruences` (13.4.6), `CayleyDigraphOfLeftCongruences` (13.4.6), or `CayleyDigraphOfRightCongruences` (13.4.6).

`IsCayleyDigraphOfCongruences` only applies to the output of `JoinSemilatticeOfCongruences` (13.4.11), `CayleyDigraphOfCongruences` (13.4.6), `CayleyDigraphOfLeftCongruences` (13.4.6), and `CayleyDigraphOfRightCongruences` (13.4.6). The congruences used as the generating set for these operations can be obtained using `GeneratingCongruencesOfJoinSemilattice` (13.4.12).

Example

```
gap> S := SymmetricInverseMonoid(2);;
gap> poset := LatticeOfCongruences(S);
<lattice of 4 two-sided congruences over
 <symmetric inverse monoid of degree 2>>
gap> IsCongruencePoset(poset);
true
gap> IsDigraph(poset);
true
gap> IsIsomorphicDigraph(poset,
> Digraph([[1, 2, 3, 4], [2], [2, 3], [2, 3, 4]]));
true
gap> T := FullTransformationMonoid(3);;
gap> congs := PrincipalCongruencesOfSemigroup(T);;
gap> poset := JoinSemilatticeOfCongruences(PosetOfCongruences(congs));
<lattice of 6 two-sided congruences over
 <full transformation monoid of degree 3>>
gap> IsCayleyDigraphOfCongruences(poset);
false
gap> IsCongruencePoset(poset);
true
gap> DigraphNrVertices(poset);
6
gap> poset := CayleyDigraphOfCongruences(T);
<poset of 7 two-sided congruences over
 <full transformation monoid of degree 3>>
gap> IsCayleyDigraphOfCongruences(poset);
true
```

### 13.4.5 LatticeOfCongruences (for a semigroup)

▷ `LatticeOfCongruences(`*S*`)` (attribute)

▷ `LatticeOfLeftCongruences(`*S*`)` (attribute)

▷ LatticeOfRightCongruences(*S*) (attribute)
▷ LatticeOfCongruences(*S*, *restriction*) (operation)
▷ LatticeOfLeftCongruences(*S*, *restriction*) (operation)
▷ LatticeOfRightCongruences(*S*, *restriction*) (operation)

**Returns:** A lattice digraph.

If *S* is a semigroup, then LatticeOfCongruences returns a congruence poset object containing all the congruences of *S* and information about how they are contained in each other. See IsCongruencePoset (13.4.4) for more details.

LatticeOfLeftCongruences and LatticeOfRightCongruences do the same thing for left and right congruences, respectively.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

See CongruencesOfSemigroup (13.4.1).

Example

```
gap> S := OrderEndomorphisms(2);;
gap> LatticeOfCongruences(S);
<lattice of 3 two-sided congruences over <regular transformation
 monoid of size 3, degree 2 with 2 generators>>
gap> LatticeOfLeftCongruences(S);
<lattice of 3 left congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> LatticeOfRightCongruences(S);
<lattice of 5 right congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> IsIsomorphicDigraph(LatticeOfRightCongruences(S),
> Digraph([[1, 2, 3, 4, 5], [2], [2, 3], [2, 4], [2, 5]]));
true
gap> S := FullTransformationMonoid(4);;
gap> restriction := [Transformation([1, 1, 1, 1]),
>                    Transformation([1, 1, 1, 2]),
>                    Transformation([1, 1, 1, 3])];;
gap> latt := LatticeOfCongruences(S, Combinations(restriction, 2));
<lattice of 2 two-sided congruences over
 <full transformation monoid of degree 4>>
```

### 13.4.6 CayleyDigraphOfCongruences (for a semigroup)

▷ CayleyDigraphOfCongruences(*S*) (attribute)
▷ CayleyDigraphOfLeftCongruences(*S*) (attribute)
▷ CayleyDigraphOfRightCongruences(*S*) (attribute)
▷ CayleyDigraphOfCongruences(*S*, *restriction*) (operation)
▷ CayleyDigraphOfLeftCongruences(*S*, *restriction*) (operation)
▷ CayleyDigraphOfRightCongruences(*S*, *restriction*) (operation)

**Returns:** A digraph.

If *S* is a semigroup, then CayleyDigraphOfCongruences returns the right Cayley graph of the semilattice of congruences of *S* with respect to the generating set consisting of the principal congruences in the congruence poset. See IsCayleyDigraphOfCongruences (13.4.4) for more details.

CayleyDigraphOfLeftCongruences and CayleyDigraphOfRightCongruences do the same thing for left and right congruences, respectively.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include congruences generated by pairs of elements from *restriction*. Otherwise, all congruences will be calculated.

Note that LatticeOfCongruences (13.4.5), and its analogues for right and left congruences, return the reflexive transitive closure of the digraph returned by this function (with any multiple edges removed). If there are a large number of congruences, then it might be the case that forming the reflexive transitive closure takes a significant amount of time, and so it might be desirable to use this function instead.

See CongruencesOfSemigroup (13.4.1).

Example

```
gap> S := OrderEndomorphisms(2);;
gap> CayleyDigraphOfCongruences(S);
<poset of 3 two-sided congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> CayleyDigraphOfLeftCongruences(S);
<poset of 3 left congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> CayleyDigraphOfRightCongruences(S);
<poset of 5 right congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> IsIsomorphicDigraph(CayleyDigraphOfRightCongruences(S),
> Digraph([[2, 3, 4], [2, 5, 5], [5, 3, 5], [5, 5, 4], [5, 5, 5]]));
true
gap> S := FullTransformationMonoid(4);;
gap> restriction := [Transformation([1, 1, 1, 1]),
>                    Transformation([1, 1, 1, 2]),
>                    Transformation([1, 1, 1, 3])];;
gap> CayleyDigraphOfCongruences(S, Combinations(restriction, 2));
<poset of 2 two-sided congruences over
 <full transformation monoid of degree 4>>
```

### 13.4.7 PosetOfPrincipalCongruences (for a semigroup)

▷ PosetOfPrincipalCongruences(*S*) (attribute)
▷ PosetOfPrincipalLeftCongruences(*S*) (attribute)
▷ PosetOfPrincipalRightCongruences(*S*) (attribute)
▷ PosetOfPrincipalCongruences(*S*, *restriction*) (operation)
▷ PosetOfPrincipalLeftCongruences(*S*, *restriction*) (operation)
▷ PosetOfPrincipalRightCongruences(*S*, *restriction*) (operation)

**Returns:** A congruence poset.

If *S* is a semigroup, then PosetOfPrincipalCongruences returns a congruence poset object which contains all the principal congruences of *S*, ordered by containment according to IsSubrelation (13.5.1). A congruence is *principal* if it can be defined by a single generating pair. PosetOfPrincipalLeftCongruences and PosetOfPrincipalRightCongruences do the same thing for left and right congruences respectively.

If *restriction* is specified and is a collection of elements from *S*, then the result will only include principal congruences generated by pairs of elements from *restriction*. Otherwise, all

principal congruences will be calculated.

See also LatticeOfCongruences (13.4.5) and PrincipalCongruencesOfSemigroup (13.4.3).

Example

```
gap> S := Semigroup(Transformation([1, 3, 1]),
>                   Transformation([2, 3, 3]));;
gap> PosetOfPrincipalLeftCongruences(S);
<poset of 12 left congruences over <transformation semigroup
 of size 11, degree 3 with 2 generators>>
gap> PosetOfPrincipalCongruences(S);
<lattice of 3 two-sided congruences over <transformation semigroup
 of size 11, degree 3 with 2 generators>>
gap> restriction := [Transformation([3, 2, 3]),
>                    Transformation([3, 1, 3]),
>                    Transformation([2, 2, 2])];;
gap> poset := PosetOfPrincipalRightCongruences(S,
> Combinations(restriction, 2));
<poset of 3 right congruences over <transformation semigroup
 of size 11, degree 3 with 2 generators>>
```

### 13.4.8 CongruencesOfPoset

▷ CongruencesOfPoset(*poset*) (attribute)

**Returns:** A list.

If *poset* is a congruence poset object, then this attribute returns a list of all the congruence objects in the poset (these may be left, right, or two-sided). The order of this list corresponds to the order of the entries in the poset.

See also LatticeOfCongruences (13.4.5) and CongruencesOfSemigroup (13.4.1).

Example

```
gap> S := OrderEndomorphisms(2);;
gap> latt := LatticeOfRightCongruences(S);
<lattice of 5 right congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> CongruencesOfPoset(latt);
[ <2-sided semigroup congruence over <regular transformation monoid
     of size 3, degree 2 with 2 generators> with 0 generating pairs>,
  <right semigroup congruence over <regular transformation monoid
     of size 3, degree 2 with 2 generators> with 1 generating pairs>,
  <right semigroup congruence over <regular transformation monoid
     of size 3, degree 2 with 2 generators> with 1 generating pairs>,
  <right semigroup congruence over <regular transformation monoid
     of size 3, degree 2 with 2 generators> with 1 generating pairs>,
  <right semigroup congruence over <regular transformation monoid
     of size 3, degree 2 with 2 generators> with 2 generating pairs> ]
```

### 13.4.9 UnderlyingSemigroupOfCongruencePoset

▷ UnderlyingSemigroupOfCongruencePoset(*poset*) (attribute)

**Returns:** A semigroup.

If *poset* is a congruence poset object, then this attribute returns the semigroup on which all its congruences are defined.

Example

```
gap> S := OrderEndomorphisms(2);
<regular transformation monoid of degree 2 with 2 generators>
gap> latt := LatticeOfRightCongruences(S);
<lattice of 5 right congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> UnderlyingSemigroupOfCongruencePoset(latt) = S;
true
```

### 13.4.10 PosetOfCongruences

▷ PosetOfCongruences(*coll*) (operation)

**Returns:** A congruence poset.

If *coll* is a list or collection of semigroup congruences (which may be left, right, or two-sided) then this operation returns the congruence poset formed by these congruences partially ordered by containment.

This operation does not create any new congruences or take any joins. See also JoinSemilatticeOfCongruences (13.4.11), IsCongruencePoset (13.4.4), and LatticeOfCongruences (13.4.5).

Example

```
gap> S := OrderEndomorphisms(2);;
gap> pair1 := [Transformation([1, 1]), IdentityTransformation];;
gap> pair2 := [IdentityTransformation, Transformation([2, 2])];;
gap> coll := [RightSemigroupCongruence(S, pair1),
>             RightSemigroupCongruence(S, pair2),
>             RightSemigroupCongruence(S, [])];;
gap> poset := PosetOfCongruences(coll);
<poset of 3 right congruences over <regular transformation monoid
 of size 3, degree 2 with 2 generators>>
gap> OutNeighbours(poset);
[ [ 1 ], [ 2 ], [ 1, 2, 3 ] ]
```

### 13.4.11 JoinSemilatticeOfCongruences

▷ JoinSemilatticeOfCongruences(*poset*) (attribute)

**Returns:** A congruence poset.

If *poset* is a congruence poset (i.e. it satisfies IsCongruencePoset (13.4.4)), then this function returns the congruence poset formed by these congruences partially ordered by containment, along with all their joins. This includes the empty join which equals the trivial congruence.

The digraph returned by this function represents the Cayley graph of the semilattice generated by CongruencesOfPoset (13.4.8) with identity adjoined. The reflexive transitive closure of this digraph is a join semilattice in the sense of IsJoinSemilatticeDigraph (**Digraphs: IsJoinSemilatticeDigraph**).

See also IsCongruencePoset (13.4.4) and PosetOfCongruences (13.4.10).

Example

```
gap> S := SymmetricInverseMonoid(2);;
gap> pair1 := [PartialPerm([1], [1]), PartialPerm([2], [1])];;
gap> pair2 := [PartialPerm([1], [1]), PartialPerm([1, 2], [1, 2])];;
gap> pair3 := [PartialPerm([1, 2], [1, 2]),
```

```
>              PartialPerm([1, 2], [2, 1])];;
gap> coll := [RightSemigroupCongruence(S, pair1),
>             RightSemigroupCongruence(S, pair2),
>             RightSemigroupCongruence(S, pair3)];;
gap> D := JoinSemilatticeOfCongruences(PosetOfCongruences(coll));
<poset of 4 right congruences over
 <symmetric inverse monoid of degree 2>>
gap> IsJoinSemilatticeDigraph(DigraphReflexiveTransitiveClosure(D));
true
```

### 13.4.12 GeneratingCongruencesOfJoinSemilattice

▷ GeneratingCongruencesOfJoinSemilattice(*poset*) (attribute)

**Returns:** A list of congruences.

If *poset* satisfies IsCayleyDigraphOfCongruences (13.4.4), then this attribute holds the generating set for the semilattice of congruences (where the operation is join).

Example

```
gap> S := OrderEndomorphisms(3);;
gap> D := CayleyDigraphOfCongruences(S);
<poset of 4 two-sided congruences over <regular transformation monoid
 of size 10, degree 3 with 3 generators>>
gap> GeneratingCongruencesOfJoinSemilattice(D);
[ <universal semigroup congruence over <regular transformation monoid
     of size 10, degree 3 with 3 generators>>,
  <2-sided semigroup congruence over <regular transformation monoid
     of size 10, degree 3 with 3 generators> with 1 generating pairs>,
  <2-sided semigroup congruence over <regular transformation monoid
     of size 10, degree 3 with 3 generators> with 1 generating pairs>
 ]
```

### 13.4.13 MinimalCongruences (for a list or collection)

▷ MinimalCongruences(*coll*) (attribute)

▷ MinimalCongruences(*poset*) (attribute)

**Returns:** A list.

If *coll* is a list or collection of semigroup congruences (which may be left, right, or two-sided) then this attribute returns a list of all the congruences from *coll* which do not contain any of the others as subrelations.

Alternatively, a congruence poset *poset* can be specified; in this case, the congruences contained in *poset* will be used in place of *coll*, and information already known about their containments will be used.

This function should not be confused with MinimalCongruencesOfSemigroup (13.4.2). See also IsCongruencePoset (13.4.4) and PosetOfCongruences (13.4.10).

Example

```
gap> S := SymmetricInverseMonoid(2);;
gap> pair1 := [PartialPerm([1], [1]), PartialPerm([2], [1])];;
gap> pair2 := [PartialPerm([1], [1]), PartialPerm([1, 2], [1, 2])];;
gap> pair3 := [PartialPerm([1, 2], [1, 2]),
>              PartialPerm([1, 2], [2, 1])];;
```

```
gap> coll := [RightSemigroupCongruence(S, pair1),
>             RightSemigroupCongruence(S, pair2),
>             RightSemigroupCongruence(S, pair3)];;
gap> MinimalCongruences(PosetOfCongruences(coll));
[ <right semigroup congruence over <symmetric inverse monoid of degree\
 2> with 1 generating pairs>,
  <right semigroup congruence over <symmetric inverse monoid of degree\
 2> with 1 generating pairs> ]
gap> poset := LatticeOfCongruences(S);
<lattice of 4 two-sided congruences over
 <symmetric inverse monoid of degree 2>>
gap> MinimalCongruences(poset);
[ <2-sided semigroup congruence over <symmetric inverse monoid of degr\
ee 2> with 0 generating pairs> ]
```

### 13.4.14 NumberOfRightCongruences (for a semigroup, positive integer, and list or collection)

▷ `NumberOfRightCongruences(`*S*`, `*n*`, `*extra*`)` (operation)
▷ `NumberOfLeftCongruences(`*S*`, `*n*`, `*extra*`)` (operation)
▷ `NumberOfRightCongruences(`*S*`, `*n*`)` (operation)
▷ `NumberOfLeftCongruences(`*S*`, `*n*`)` (operation)
▷ `NumberOfRightCongruences(`*S*`)` (attribute)
▷ `NumberOfLeftCongruences(`*S*`)` (operation)

**Returns:** A non-negative integer.

`NumberOfRightCongruences` returns the number of right congruences of the semigroup *S* with at most *n* classes that contain the pairs in *extra*; `NumberOfLeftCongruences` is defined dually for left congruences rather than right congruences.

If the optional third argument *extra* is not present, then `NumberOfRightCongruences` returns the number of right congruences of *S* with at most *n* classes.

If the optional second argument *n* is not present, then `NumberOfRightCongruences` returns the number of right congruences of *S*.

Note that the 2 and 3 argument variants of this function can be applied to infinite semigroups, but the 1 argument variant cannot.

If the lattice of right or left congruences of *S* is known, then that is used by `NumberOfRightCongruences`. If this lattice is not known, then Sim's low index congruence algorithm is used.

See `IteratorOfRightCongruences` (13.4.15) to actually obtain the congruences counted by this function.

Example

```
gap> S := PartitionMonoid(2);
<regular bipartition *-monoid of size 15, degree 2 with 3 generators>
gap> NumberOfRightCongruences(S, 10);
86
gap> NumberOfLeftCongruences(S, 10);
86
gap> NumberOfRightCongruences(S, Size(S), [[S.1, S.2], [S.1, S.3]]);
1
```

```
gap> NumberOfLeftCongruences(S, Size(S), [[S.1, S.2], [S.1, S.3]]);
1
```

### 13.4.15 IteratorOfRightCongruences (for a semigroup, positive integer, and list or collection)

▷ IteratorOfRightCongruences(*S*, *n*, *extra*) (operation)
▷ IteratorOfLeftCongruences(*S*, *n*, *extra*) (operation)
▷ IteratorOfRightCongruences(*S*, *n*) (operation)
▷ IteratorOfLeftCongruences(*S*, *n*) (operation)
▷ IteratorOfRightCongruences(*S*) (attribute)
▷ IteratorOfLeftCongruences(*S*) (operation)

**Returns:** An iterator.

`IteratorOfRightCongruences` returns an iterator where calling `NextIterator` (**Reference: NextIterator**) returns the next right congruence of the semigroup *S* with at most *n* classes that contain the pairs in *extra*; `IteratorOfLeftCongruences` is defined dually for left congruences rather than right congruences.

If the optional third argument *extra* is not present, then `IteratorOfRightCongruences` uses an empty list by default.

If the optional second argument *n* is not present, then `IteratorOfRightCongruences` uses `Size(`*S*`)` by default.

Note that the 2 and 3 argument variants of this function can be applied to infinite semigroups, but the 1 argument variant cannot.

If the lattice of right or left congruences of *S* is known, then that is used by `IteratorOfRightCongruences`. If this lattice is not known, then Sim's low index congruence algorithm is used.

Example

```
gap> F := FreeMonoidAndAssignGeneratorVars("a", "b");
<free monoid on the generators [ a, b ]>
gap> R := [[a ^ 3, a], [b ^ 2, b], [(a * b) ^ 2, a]];
[ [ a^3, a ], [ b^2, b ], [ (a*b)^2, a ] ]
gap> S := F / R;
<fp monoid with 2 generators and 3 relations of length 14>
gap> NumberOfRightCongruences(S);
6
gap> it := IteratorOfRightCongruences(S);
<iterator>
gap> OutNeighbours(WordGraph(NextIterator(it)));
[ [ 1, 1 ] ]
gap> OutNeighbours(WordGraph(NextIterator(it)));
[ [ 2, 1 ], [ 2, 2 ] ]
gap> OutNeighbours(WordGraph(NextIterator(it)));
[ [ 2, 2 ], [ 2, 2 ] ]
gap> OutNeighbours(WordGraph(NextIterator(it)));
[ [ 2, 3 ], [ 2, 2 ], [ 2, 3 ] ]
gap> OutNeighbours(WordGraph(NextIterator(it)));
[ [ 2, 3 ], [ 2, 2 ], [ 3, 3 ] ]
gap> OutNeighbours(WordGraph(NextIterator(it)));
[ [ 2, 3 ], [ 2, 2 ], [ 4, 3 ], [ 4, 4 ] ]
```

```
gap> NextIterator(it);
fail
```

## 13.5 Comparing congruences

### 13.5.1 IsSubrelation

▷ IsSubrelation(*cong1*, *cong2*) (operation)

**Returns:** True or false.

If *cong1* and *cong2* are congruences over the same semigroup, then this operation returns whether *cong2* is a refinement of *cong1*, i.e. whether every pair in *cong2* is contained in *cong1*.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                     RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> IsSubrelation(cong1, cong2);
true
gap> IsSubrelation(cong2, cong1);
false
```

### 13.5.2 IsSuperrelation

▷ IsSuperrelation(*cong1*, *cong2*) (operation)

**Returns:** True or false.

If *cong1* and *cong2* are congruences over the same semigroup, then this operation returns whether *cong1* is a refinement of *cong2*, i.e. whether every pair in *cong1* is contained in *cong2*.

See IsSubrelation (13.5.1).

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                     RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> IsSuperrelation(cong1, cong2);
false
gap> IsSuperrelation(cong2, cong1);
true
```

### 13.5.3 MeetSemigroupCongruences

▷ MeetSemigroupCongruences(*c1*, *c2*) (operation)

▷ MeetLeftSemigroupCongruences(*c1*, *c2*) (operation)

▷ MeetRightSemigroupCongruences(*c1*, *c2*) (operation)

**Returns:** A semigroup congruence.

This operation returns the *meet* of the two semigroup congruences *c1* and *c2* -- that is, the largest semigroup congruence contained in both *c1* and *c2*.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                     RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> MeetSemigroupCongruences(cong1, cong2);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (1,2,2)>
```

### 13.5.4 JoinSemigroupCongruences

▷ `JoinSemigroupCongruences(c1, c2)` (operation)
▷ `JoinLeftSemigroupCongruences(c1, c2)` (operation)
▷ `JoinRightSemigroupCongruences(c1, c2)` (operation)

**Returns:** A semigroup congruence.

This operation returns the *join* of the two semigroup congruences `c1` and `c2` -- that is, the smallest semigroup congruence containing all the relations in both `c1` and `c2`.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> cong1 := SemigroupCongruence(S, [RMSElement(S, 1, (1, 2, 3), 1),
>                                     RMSElement(S, 1, (), 1)]);;
gap> cong2 := SemigroupCongruence(S, []);;
gap> JoinSemigroupCongruences(cong1, cong2);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (3,2,2)>
```

## 13.6 Congruences on Rees matrix semigroups

This section describes the implementation of congruences of simple and 0-simple semigroups in the Semigroups package, and the functions associated with them. This code and this part of the manual were written by Michael Young. 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, $\Lambda$ and $I$ are non-empty sets, and $P$ is regular in the sense that it has no rows or columns consisting solely of zeroes.

The congruences of a Rees 0-matrix semigroup are in 1-1 correspondence with the *linked triple*, which is a triple of the form `[N, S, T]` where:

- `N` is a normal subgroup of the underlying group `G`,
- `S` is an equivalence relation on the columns of `P`,
- `T` is an equivalence relation on the rows of `P`,

satisfying the following conditions:

- a pair of S-related columns must contain zeroes in precisely the same rows,
- a pair of T-related rows must contain zeroes in precisely the same columns,
- if i and j are S-related, k and l are T-related and the matrix entries $p_{k,i}, p_{k,j}, p_{l,i}, p_{l,j} \neq 0$, then $q_{k,l,i,j} \in N$, where

$$q_{k,l,i,j} = p_{k,i} p_{l,i}^{-1} p_{l,j} p_{k,j}^{-1}.$$

By Theorem 3.5.9 in [How95], for any finite 0-simple Rees 0-matrix semigroup, there is a bijection between its non-universal congruences and its linked triples. In this way, we can internally represent any congruence of such a semigroup by storing its associated linked triple instead of a set of generating pairs, and thus perform many calculations on it more efficiently.

If a congruence is defined by a linked triple `(N, S, T)`, then a single class of that congruence can be defined by a triple `(Nx, i / S, k / S)`, where `Nx` is a right coset of `N`, `i / S` is the equivalence class of `i` in `S`, and `k / S` is the equivalence class of `k` in `T`. Thus we can internally represent any class of such a congruence as a triple simply consisting of a right coset and two positive integers.

An analogous condition exists for finite simple Rees matrix semigroups without zero.

### 13.6.1 IsRMSCongruenceByLinkedTriple

▷ `IsRMSCongruenceByLinkedTriple(`*obj*`)` (category)
▷ `IsRZMSCongruenceByLinkedTriple(`*obj*`)` (category)

**Returns:** `true` or `false`.

These categories describe a type of semigroup congruence over a Rees matrix or 0-matrix semigroup. Externally, an object of this type may be used in the same way as any other object in the category `IsSemigroupCongruence` (**Reference: IsSemigroupCongruence**) but it is represented internally by its *linked triple*, and certain functions may take advantage of this information to reduce computation times.

An object of this type may be constructed with `RMSCongruenceByLinkedTriple` or `RZMSCongruenceByLinkedTriple`, or this representation may be selected automatically by `SemigroupCongruence` (13.2.1).

Example

```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);;
gap> IsRZMSCongruenceByLinkedTriple(cong);
true
```

### 13.6.2 RMSCongruenceByLinkedTriple

▷ RMSCongruenceByLinkedTriple(*S, N, colBlocks, rowBlocks*) (function)
▷ RZMSCongruenceByLinkedTriple(*S, N, colBlocks, rowBlocks*) (function)

**Returns:** A Rees matrix or 0-matrix semigroup congruence by linked triple.

This function returns a semigroup congruence over the Rees matrix or 0-matrix semigroup *S* corresponding to the linked triple (*N*, *colBlocks*, *rowBlocks*). The argument *N* should be a normal subgroup of the underlying group 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);;
```

### 13.6.3 IsRMSCongruenceClassByLinkedTriple

▷ IsRMSCongruenceClassByLinkedTriple(*obj*) (category)
▷ IsRZMSCongruenceClassByLinkedTriple(*obj*) (category)

**Returns:** `true` or `false`.

These categories contain the congruence classes of a semigroup congruence of the categories `IsRMSCongruenceByLinkedTriple` (13.6.1) and `IsRZMSCongruenceByLinkedTriple` (13.6.1) respectively.

An object of one of these types may be used in the same way as any other object in the category `IsCongruenceClass` (13.3.1), but the class is represented internally by information related to the congruence's *linked triple*, and certain functions may take advantage of this information to reduce computation times.

Example

```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);;
gap> classes := EquivalenceClasses(cong);;
```

```
gap> IsRZMSCongruenceClassByLinkedTriple(classes[1]);
true
```

### 13.6.4 RMSCongruenceClassByLinkedTriple

▷ RMSCongruenceClassByLinkedTriple(*cong*, *nCoset*, *colClass*, *rowClass*) (operation)
▷ RZMSCongruenceClassByLinkedTriple(*cong*, *nCoset*, *colClass*, *rowClass*) (operation)

**Returns:** A Rees matrix or 0-matrix semigroup congruence class by linked triple.

This operation returns one congruence class of the congruence *cong*, as defined by the other three parameters.

The argument *cong* must be a Rees matrix or 0-matrix semigroup congruence by linked triple. If the linked triple consists of the three parameters N, colBlocks and rowBlocks, then *nCoset* must be a right coset of N, *colClass* must be a positive integer corresponding to a position in the list colBlocks, and *rowClass* must be a positive integer corresponding to a position in the list rowBlocks.

If the arguments are valid, an IsRMSCongruenceClassByLinkedTriple or IsRZMSCongruenceClassByLinkedTriple object is returned, which can be used like any other equivalence class in GAP. Otherwise, an error is returned.

Example

```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> cong := RZMSCongruenceByLinkedTriple(S, N, colBlocks, rowBlocks);;
gap> class := RZMSCongruenceClassByLinkedTriple(cong,
> RightCoset(N, (1, 5)), 2, 3);
<2-sided congruence class of (2,(3,4),3)>
```

### 13.6.5 IsLinkedTriple

▷ IsLinkedTriple(*S*, *N*, *colBlocks*, *rowBlocks*) (operation)

**Returns:** true or false.

This operation returns true if and only if the arguments (*N*, *colBlocks*, *rowBlocks*) describe a linked triple of the Rees matrix or 0-matrix semigroup *S*, as described above.

Example

```
gap> G := Group([(1, 4, 5), (1, 5, 3, 4)]);;
gap> mat := [[0, 0, (1, 4, 5), 0, 0, (1, 4, 3, 5)],
>            [0, (), 0, 0, (3, 5), 0],
>            [(), 0, 0, (3, 5), 0, 0]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> N := Group([(1, 4)(3, 5), (1, 5)(3, 4)]);;
gap> colBlocks := [[1], [2, 5], [3, 6], [4]];;
gap> rowBlocks := [[1], [2], [3]];;
gap> IsLinkedTriple(S, N, colBlocks, rowBlocks);
true
```

### 13.6.6 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 linked triple (3,2,2)>
```

## 13.7 Congruences on inverse semigroups

This section describes the implementation of congruences of inverse semigroups in the Semigroups package, and the functions associated with them. This code and this part of the manual were written by Michael Young. Most of the theorems used in this chapter are from Section 5.3 of [How95].

The congruences of an inverse semigroup are in 1-1 correspondence with its *congruence pairs*. A congruence pair is a pair `(N, t)` such that:

- `N` is a normal subsemigroup of `S` -- that is, a self-conjugate subsemigroup which contains all the idempotents of `S`,
- `t` is a normal congruence on `E`, the subsemigroup of all idempotents in `S` -- that is, a congruence on `E` such that if $(e, f)$ is a pair in `t`, then the pair $(a^{-1}ea, a^{-1}fa)$ is also in `t`,

satisfying the following conditions:

- If $ae \in N$ and $(e, a^{-1}a) \in t$, then $a \in N$,
- If $a \in N$, then $(aa^{-1}, a^{-1}a) \in t$.

By Theorem 5.3.3 in [How95], for any inverse semigroup, there is a bijection between its congruences and its congruence pairs. In this way, we can internally represent any congruence of such a semigroup by storing its associated congruence pair instead of a set of generating pairs, and thus perform many calculations on it more efficiently.

If we have a congruence `C` with congruence pair `(N, t)`, it turns out that `N` is its *kernel* (that is, the set of all elements congruent to an idempotent) and that `t` is its *trace* (that is, the restriction of `C` to the idempotents). Hence, we refer to a congruence stored in this format as a "congruence by kernel and trace".

See `cong_by_ker_trace_threshold` in Section 6.3 for details on when this method is used.

### 13.7.1 IsInverseSemigroupCongruenceByKernelTrace

▷ `IsInverseSemigroupCongruenceByKernelTrace(`*cong*`)` (category)

**Returns:** `true` or `false`.

This category contains any inverse semigroup congruence *cong* which is represented internally by its kernel and trace. The SemigroupCongruence (13.2.1) function may create an object of this category if its first argument *S* is an inverse semigroup and has sufficiently large size. It can be treated like any other semigroup congruence object.

See [How95] Section 5.3 for more details. See also InverseSemigroupCongruenceByKernelTrace (13.7.2).

Example

```
gap> S := InverseSemigroup([
>  PartialPerm([4, 3, 1, 2]),
>  PartialPerm([1, 4, 2, 0, 3])],
>  rec(cong_by_ker_trace_threshold := 0));;
gap> cong := SemigroupCongruence(S, []);
<semigroup congruence over <inverse partial perm semigroup
 of size 351, rank 5 with 2 generators> with congruence pair (24,24)>
gap> IsInverseSemigroupCongruenceByKernelTrace(cong);
true
```

### 13.7.2 InverseSemigroupCongruenceByKernelTrace

▷ InverseSemigroupCongruenceByKernelTrace(*S*, *kernel*, *trace*) (function)

**Returns:** An inverse semigroup congruence by kernel and trace.

If *S* is an inverse semigroup, *kernel* is a subsemigroup of *S*, *trace* is a list of lists describing a congruence on the idempotents of *S*, and (*kernel*, *trace*) describes a valid congruence pair for *S* (see [How95] Section 5.3), then this function returns the semigroup congruence defined by that congruence pair.

See also KernelOfSemigroupCongruence (13.7.4) and TraceOfSemigroupCongruence (13.7.5).

Example

```
gap> S := InverseSemigroup([
>   PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> kernel := InverseSemigroup([
>   PartialPerm([1, 0, 3]), PartialPerm([0, 2, 3]),
>   PartialPerm([1, 2]), PartialPerm([3]),
>   PartialPerm([2])]);;
gap> trace := [
>  [PartialPerm([0, 2, 3])],
>  [PartialPerm([1, 2])],
>  [PartialPerm([1, 0, 3])],
>  [PartialPerm([0, 0, 3]), PartialPerm([0, 2]),
>   PartialPerm([1]), PartialPerm([], [])]];;
gap> cong := InverseSemigroupCongruenceByKernelTrace(S, kernel, trace);
<semigroup congruence over <inverse partial perm semigroup of rank 3
 with 2 generators> with congruence pair (13,4)>
```

### 13.7.3 AsInverseSemigroupCongruenceByKernelTrace

▷ AsInverseSemigroupCongruenceByKernelTrace(*cong*) (attribute)

**Returns:** An inverse semigroup congruence by kernel and trace.

If *cong* is a semigroup congruence over an inverse semigroup, then this attribute returns an object which describes the same congruence, but with an internal representation defined by that congruence's kernel and trace.

See [How95] section 5.3 for more details.

Example

```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> cong := SemigroupCongruenceByGeneratingPairs(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);
<2-sided semigroup congruence over <inverse partial perm semigroup of
 rank 3 with 2 generators> with 2 generating pairs>
gap> cong2 := AsInverseSemigroupCongruenceByKernelTrace(cong);
<semigroup congruence over <inverse partial perm semigroup
 of size 19, rank 3 with 2 generators> with congruence pair (19,1)>
```

### 13.7.4 KernelOfSemigroupCongruence

▷ KernelOfSemigroupCongruence(*cong*) (attribute)

**Returns:** An inverse semigroup.

If *cong* is a congruence over a semigroup with an inverse operation, then this attribute returns the *kernel* of that congruence; that is, the inverse subsemigroup consisting of all elements which are related to an idempotent by *cong*.

Example

```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> cong := SemigroupCongruence(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);
<2-sided semigroup congruence over <inverse partial perm semigroup
 of size 19, rank 3 with 2 generators> with 2 generating pairs>
gap> KernelOfSemigroupCongruence(cong);
<inverse partial perm semigroup of size 19, rank 3 with 5 generators>
```

### 13.7.5 TraceOfSemigroupCongruence

▷ TraceOfSemigroupCongruence(*cong*) (attribute)

**Returns:** A list of lists.

If *cong* is an inverse semigroup congruence by kernel and trace, then this attribute returns the restriction of *cong* to the idempotents of the semigroup. This is in block form: each idempotent will appear in precisely one list, and two idempotents will be in the same list if and only if they are related by *cong*.

Example

```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])]);;
gap> cong := SemigroupCongruence(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);
<2-sided semigroup congruence over <inverse partial perm semigroup
 of size 19, rank 3 with 2 generators> with 2 generating pairs>
```

```
gap> TraceOfSemigroupCongruence(cong);
[ [ <empty partial perm>, <identity partial perm on [ 1 ]>,
      <identity partial perm on [ 2 ]>,
      <identity partial perm on [ 1, 2 ]>,
      <identity partial perm on [ 3 ]>,
      <identity partial perm on [ 2, 3 ]>,
      <identity partial perm on [ 1, 3 ]> ] ]
```

### 13.7.6 IsInverseSemigroupCongruenceClassByKernelTrace

▷ IsInverseSemigroupCongruenceClassByKernelTrace(*obj*) (category)

**Returns:** true or false.

This category contains any congruence class which belongs to a congruence which is represented internally by its kernel and trace. See InverseSemigroupCongruenceByKernelTrace (13.7.2).

See [How95] Section 5.3 for more details.

Example

```
gap> I := InverseSemigroup([
>  PartialPerm([2, 3]), PartialPerm([2, 0, 3])],
> rec(cong_by_ker_trace_threshold := 0));;
gap> cong := SemigroupCongruence(I,
> [[PartialPerm([0, 1, 3]), PartialPerm([0, 1])],
>  [PartialPerm([]), PartialPerm([1, 2])]]);;
gap> class := EquivalenceClassOfElement(cong,
>                                       PartialPerm([1, 2], [2, 3]));;
gap> IsInverseSemigroupCongruenceClassByKernelTrace(class);
true
```

### 13.7.7 MinimumGroupCongruence

▷ MinimumGroupCongruence(*S*) (attribute)

**Returns:** An inverse semigroup congruence by kernel and trace.

If *S* is an inverse semigroup, then this function returns the least congruence on *S* whose quotient is a group.

Example

```
gap> S := InverseSemigroup([
>   PartialPerm([5, 2, 0, 0, 1, 4]),
>   PartialPerm([1, 4, 6, 3, 5, 0, 2])]);;
gap> cong := MinimumGroupCongruence(S);
<semigroup congruence over <inverse partial perm semigroup
 of size 101, rank 7 with 2 generators> with congruence pair (59,1)>
gap> IsGroupAsSemigroup(S / cong);
true
```

## 13.8 Congruences on graph inverse semigroups

### 13.8.1 IsCongruenceByWangPair

▷ IsCongruenceByWangPair(*cong*) (property)

A congruence by Wang pair cong is a congruence of a graph inverse semigroup S which is expressed in terms of two sets $H$ and $W$ of vertices of the corresponding graph of $S$. The set $H$ must be a hereditary subset (closed under reachability) and all vertices in $W$ must have all but one of their out-neighbours in $H$. For more information on Wang pairs see [Wan19] and [AMM23].

Example

```
gap> D := Digraph([[3, 4], [3, 4], [4], []]);
<immutable digraph with 4 vertices, 5 edges>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 4 vertices, 5 edges>
gap> cong := CongruenceByWangPair(S, [3, 4], []);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [  ]>
gap> IsCongruenceByWangPair(cong);
true
gap> cong := CongruenceByWangPair(S, [4], [2]);
<graph inverse semigroup congruence with H = [ 4 ] and W = [ 2 ]>
gap> IsCongruenceByWangPair(cong);
true
gap> e_1 := S.1;
e_1
gap> e_3 := S.3;
e_3
gap> cong := SemigroupCongruence(S, [[e_1, e_3]]);
<2-sided semigroup congruence over <finite graph inverse semigroup wit\
h 4 vertices, 5 edges> with 1 generating pairs>
gap> IsCongruenceByWangPair(cong);
false
```

### 13.8.2 CongruenceByWangPair

▷ CongruenceByWangPair(*S*, *H*, *W*) (function)

**Returns:** A semigroup congruence.

This function returns a semigroup congruence over the graph inverse semigroup $S$ in the form of a Wang pair.

If $S$ is a finite graph inverse semigroup, and $H$ and $W$ are two lists of vertices in the graph of $S$ representing a valid hereditary subset and a W-set respectively, then this function will return the semigroup congruence defined by this Wang pair. For the definition of a Wang pair, see IsCongruenceByWangPair (13.8.1).

Example

```
gap> D := Digraph([[3, 4], [3, 4], [4], []]);
<immutable digraph with 4 vertices, 5 edges>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 4 vertices, 5 edges>
gap> cong := CongruenceByWangPair(S, [3, 4], []);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [  ]>
gap> cong := CongruenceByWangPair(S, [4], [2]);
<graph inverse semigroup congruence with H = [ 4 ] and W = [ 2 ]>
gap> cong := CongruenceByWangPair(S, [3, 4], []);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [  ]>
```

### 13.8.3 AsCongruenceByWangPair

▷ AsCongruenceByWangPair(*cong*) (operation)

**Returns:** A congruence by Wang pair.

This operation takes *cong*, a finite graph inverse semigroup congruence, and returns an object representing the same congruence, but described as a congruence by Wang pair: a pair of sets $H$ and $W$ of the corresponding graph of $S$ that are a hereditary subset and a W-set of the graph of $S$ respectively. For more information about Wang pairs see [Wan19] and [AMM23].

Example

```
gap> D := Digraph([[2, 3], [3], [4], []]);
<immutable digraph with 4 vertices, 4 edges>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 4 vertices, 4 edges>
gap> CongruenceByWangPair(S, [4], [2]);
<graph inverse semigroup congruence with H = [ 4 ] and W = [ 2 ]>
gap> cong := AsSemigroupCongruenceByGeneratingPairs(last);
<2-sided semigroup congruence over <finite graph inverse semigroup wit\
h 4 vertices, 4 edges> with 2 generating pairs>
gap> AsCongruenceByWangPair(cong);
<graph inverse semigroup congruence with H = [ 4 ] and W = [ 2 ]>
gap> CongruenceByWangPair(S, [3, 4], [1]);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [ 1 ]>
gap> cong := AsSemigroupCongruenceByGeneratingPairs(last);
<2-sided semigroup congruence over <finite graph inverse semigroup wit\
h 4 vertices, 4 edges> with 3 generating pairs>
gap> AsCongruenceByWangPair(cong);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [ 1 ]>
```

### 13.8.4 GeneratingCongruencesOfLattice

▷ GeneratingCongruencesOfLattice(*S*) (attribute)

**Returns:** A semigroup.

This attribute takes a finite graph inverse semigroup $S$ and returns a minimal generating set for the lattice of congruences of $S$, as described in [AMM23]. This operation works only if the corresponding digraph of the graph inverse semigroup is simple. If there are multiple edges, an error is returned.

Example

```
gap> D := Digraph([[2, 3], [3], [4], []]);
<immutable digraph with 4 vertices, 4 edges>
gap> S := GraphInverseSemigroup(D);
<finite graph inverse semigroup with 4 vertices, 4 edges>
gap> CongruenceByWangPair(S, [4], [2]);
<graph inverse semigroup congruence with H = [ 4 ] and W = [ 2 ]>
gap> cong := AsSemigroupCongruenceByGeneratingPairs(last);
<2-sided semigroup congruence over <finite graph inverse semigroup wit\
h 4 vertices, 4 edges> with 2 generating pairs>
gap> AsCongruenceByWangPair(cong);
<graph inverse semigroup congruence with H = [ 4 ] and W = [ 2 ]>
gap> CongruenceByWangPair(S, [3, 4], [1]);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [ 1 ]>
gap> cong := AsSemigroupCongruenceByGeneratingPairs(last);
<2-sided semigroup congruence over <finite graph inverse semigroup wit\
```

```
h 4 vertices, 4 edges> with 3 generating pairs>
gap> AsCongruenceByWangPair(cong);
<graph inverse semigroup congruence with H = [ 3, 4 ] and W = [ 1 ]>
```

## 13.9 Rees congruences

A Rees congruence is defined by a semigroup ideal. It is a congruence on a semigroup `S` which has one congruence class equal to a semigroup ideal `I` of `S`, and every other congruence class being a singleton.

### 13.9.1 SemigroupIdealOfReesCongruence

▷ `SemigroupIdealOfReesCongruence(`*cong*`)` (attribute)

**Returns:** A semigroup ideal.

If *cong* is a Rees congruence (see `IsReesCongruence` (**Reference: IsReesCongruence**)), then this attribute returns the two-sided ideal that was used to define it, i.e.˜the ideal of elements in the only non-trivial congruence class of *cong*.

Example

```
gap> S := Semigroup([
> Transformation([2, 3, 4, 3, 1, 1]),
> Transformation([6, 4, 4, 4, 6, 1])]);;
gap> I := SemigroupIdeal(S,
> Transformation([4, 4, 4, 4, 4, 2]),
> Transformation([3, 3, 3, 3, 3, 2]));;
gap> cong := ReesCongruenceOfSemigroupIdeal(I);;
gap> SemigroupIdealOfReesCongruence(cong);
<non-regular transformation semigroup ideal of degree 6 with
  2 generators>
```

### 13.9.2 IsReesCongruenceClass

▷ `IsReesCongruenceClass(`*obj*`)` (category)

**Returns:** `true` or `false`.

This category describes a congruence class of a Rees congruence. A congruence class of a Rees congruence either contains all the elements of an ideal, or is a singleton (see `IsReesCongruence` (**Reference: IsReesCongruence**)).

An object of this type may be used in the same way as any other congruence class object.

Example

```
gap> S := Semigroup(
> Transformation([2, 3, 4, 3, 1, 1]),
> Transformation([6, 4, 4, 4, 6, 1]));;
gap> I := SemigroupIdeal(S,
> Transformation([4, 4, 4, 4, 4, 2]),
> Transformation([3, 3, 3, 3, 3, 2]));;
gap> cong := ReesCongruenceOfSemigroupIdeal(I);;
gap> classes := EquivalenceClasses(cong);;
gap> IsReesCongruenceClass(classes[1]);
true
```

## 13.10 Universal and trivial 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.

### 13.10.1 IsUniversalSemigroupCongruence

▷ `IsUniversalSemigroupCongruence(`*obj*`)` (property)

**Returns:** `true` or `false`.

This property describes a type of semigroup congruence, which must refer to the *universal semigroup congruence* $S \times S$. Externally, an object of this type may be used in the same way as any other object in the category `IsSemigroupCongruence` (**Reference: IsSemigroupCongruence**).

An object of this type may be constructed with `UniversalSemigroupCongruence` or this representation may be selected automatically as an alternative to an `IsRZMSCongruenceByLinkedTriple` object (since the universal congruence cannot be represented by a linked triple).

Example

```
gap> S := Semigroup([Transformation([3, 2, 3])]);;
gap> U := UniversalSemigroupCongruence(S);;
gap> IsUniversalSemigroupCongruence(U);
true
```

### 13.10.2 IsUniversalSemigroupCongruenceClass

▷ `IsUniversalSemigroupCongruenceClass(`*obj*`)` (category)

**Returns:** `true` or `false`.

This category describes a class of the universal semigroup congruence (see `IsUniversalSemigroupCongruence` (13.10.1)). A universal semigroup congruence by definition has precisely one congruence class, which contains all of the elements of the semigroup in question.

Example

```
gap> S := Semigroup([Transformation([3, 2, 3])]);;
gap> U := UniversalSemigroupCongruence(S);;
gap> classes := EquivalenceClasses(U);;
gap> IsUniversalSemigroupCongruenceClass(classes[1]);
true
```

### 13.10.3 UniversalSemigroupCongruence

▷ `UniversalSemigroupCongruence(`*S*`)` (operation)

**Returns:** A universal semigroup congruence.

This operation returns the universal semigroup congruence for the semigroup *S*. It can be used in the same way as any other semigroup congruence object.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(1, 2), 0]]);;
gap> UniversalSemigroupCongruence(S);
<universal semigroup congruence over
<Rees 0-matrix semigroup 2x2 over Sym( [ 1 .. 3 ] )>>
```

### 13.10.4 TrivialCongruence

▷ `TrivialCongruence(`*S*`)` (attribute)

**Returns:** A trivial semigroup congruence.

This operation returns the trivial 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> TrivialCongruence(S);
<semigroup congruence over <Rees 0-matrix semigroup 2x2 over
  Sym( [ 1 .. 3 ] )> with linked triple (1,2,2)>
gap> S := PartitionMonoid(2);
<regular bipartition *-monoid of size 15, degree 2 with 3 generators>
gap> TrivialCongruence(S);
<2-sided semigroup congruence over <regular bipartition *-monoid
 of size 15, degree 2 with 3 generators> with 0 generating pairs>
```

# Chapter 14

# Semigroup homomorphisms

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

## 14.1 Homomorphisms of arbitrary semigroups

### 14.1.1 SemigroupHomomorphismByImages (for two semigroups and two lists)

▷ SemigroupHomomorphismByImages(*S*, *T*, *gens*, *imgs*) (operation)
▷ SemigroupHomomorphismByImages(*S*, *T*, *imgs*) (operation)
▷ SemigroupHomomorphismByImages(*S*, *T*) (operation)
▷ SemigroupHomomorphismByImages(*S*, *gens*, *imgs*) (operation)

**Returns:** A semigroup homomorphism, or `fail`.

`SemigroupHomomorphismByImages` attempts to construct a homomorphism from the semigroup *S* to the semigroup *T* by mapping the `i`-th element of *gens* to the `i`-th element of *imgs*. If this mapping corresponds to a homomorphism, the homomorphism is returned, and if not, then `fail` is returned. Similarly, if *gens* does not generate *S*, `fail` is returned.

If omitted, the arguments *gens* and *imgs* default to the generators of *S* and *T* respectively. See `GeneratorsOfSemigroup` (**Reference: GeneratorsOfSemigroup**).

If *T* is not given, then it defaults to the semigroup generated by *imgs*, resulting in the mapping being surjective.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> gens := GeneratorsOfSemigroup(S);;
gap> J := FullTransformationMonoid(4);;
gap> imgs := ListWithIdenticalEntries(4,
> ConstantTransformation(3, 1));;
gap> hom := SemigroupHomomorphismByImages(S, J, gens, imgs);
<full transformation monoid of degree 3> ->
<full transformation monoid of degree 4>
```

### 14.1.2 SemigroupHomomorphismByFunctionNC

▷ SemigroupHomomorphismByFunctionNC(*S*, *T*, *fun*) (operation)
▷ SemigroupHomomorphismByFunction(*S*, *T*, *fun*) (operation)

**Returns:** A semigroup homomorphism or `fail`.

SemigroupHomomorphismByFunctionNC returns a semigroup homomorphism with source *S* and range *T*, such that each element s in *S* is mapped to the element *fun*(s), where *fun* is a GAP function.

The function SemigroupHomomorphismByFunctionNC performs no checks on whether the function actually gives a homomorphism, and so it is possible for this operation to return a mapping from *S* to *T* that is not a homomorphism.

The function SemigroupHomomorphismByFunction checks that the mapping from *S* to *T* defined by *fun* satisfies RespectsMultiplication (**Reference: RespectsMultiplication**), which can be expensive. If RespectsMultiplication (**Reference: RespectsMultiplication**) does not hold, then fail is returned.

Example

```
gap> g := Semigroup([(1, 2, 3, 4), (1, 2)]);;
gap> h := Semigroup([(1, 2, 3), (1, 2)]);;
gap> hom := SemigroupHomomorphismByFunction(g, h,
> function(x)
> if SignPerm(x) = -1 then return (1, 2);
> else return ();
> fi; end);
<semigroup of size 24, with 2 generators> ->
<semigroup of size 6, with 2 generators>
```

The following methods relate to semigroup homomorphisms by images or by function:

- Range (**Reference: range**),
- Image (**Reference: Image**),
- Images (**Reference: Images**),
- ImageElm (**Reference: ImageElm**),
- PreImage (**Reference: PreImage**),
- PreImages (**Reference: PreImages**),
- PreImagesRepresentative (**Reference: PreImagesRepresentative**),
- PreImagesRange (**Reference: PreImagesRange**),
- PreImagesElm (**Reference: PreImagesElm**),
- PreImagesSet (**Reference: PreImagesSet**),
- IsSurjective (**Reference: IsSurjective**),
- IsInjective (**Reference: IsInjective**),
- IsBijective (**Reference: IsBijective**),
- Source (**Reference: Source**),
- Range (**Reference: range**),
- ImagesSource (**Reference: ImagesSource**),
- KernelOfSemigroupHomomorphism (14.1.7).

### 14.1.3 IsSemigroupHomomorphismByImages

▷ IsSemigroupHomomorphismByImages(*hom*) (filter)

**Returns:** true or false.

IsSemigroupHomomorphismByImages returns true if *hom* is a semigroup homomorphism by images and false if it is not. A semigroup homomorphism is a mapping from a semigroup S to a semigroup T that respects multiplication. This representation describes semigroup homomorphisms internally by the generators of S and their images in T. See SemigroupHomomorphismByImages (14.1.1).

Example

```
gap> S := FullTransformationMonoid(3);;
gap> gens := GeneratorsOfSemigroup(S);;
gap> T := FullTransformationMonoid(4);;
gap> imgs := ListWithIdenticalEntries(4, ConstantTransformation(3, 1));;
gap> hom := SemigroupHomomorphismByImages(S, T, gens, imgs);
<full transformation monoid of degree 3> ->
<full transformation monoid of degree 4>
gap> IsSemigroupHomomorphismByImages(hom);
true
```

### 14.1.4 IsSemigroupHomomorphismByFunction

▷ IsSemigroupHomomorphismByFunction(*hom*) (filter)

**Returns:** true or false.

IsSemigroupHomomorphismByFunction returns true if *hom* was created using SemigroupHomomorphismByFunction (14.1.2) and false if it was not. Note that this filter may return true even if the underlying GAP function does not define a homomorphism. A semigroup homomorphism is a mapping from a semigroup S to a semigroup T that respects multiplication. This representation describes semigroup homomorphisms internally using a GAP function mapping elements of S to their images in T.

Example

```
gap> S := Semigroup([(1, 2, 3, 4), (1, 2)]);;
gap> T := Semigroup([(1, 2, 3), (1, 2)]);;
gap> hom := SemigroupHomomorphismByFunction(S, T,
> function(x) if SignPerm(x) = -1 then return (1, 2);
> else return ();fi; end);
<semigroup of size 24, with 2 generators> ->
<semigroup of size 6, with 2 generators>
gap> IsSemigroupHomomorphismByFunction(hom);
true
```

### 14.1.5 AsSemigroupHomomorphismByImages (for a semigroup homomorphism by function)

▷ AsSemigroupHomomorphismByImages(*hom*) (operation)

**Returns:** A semigroup homomorphism, or fail.

AsSemigroupHomomorphismByImages takes *hom*, a semigroup homomorphism, and returns the same mapping but represented internally using the generators of Source(*hom*) and their images in Range(*hom*). If *hom* is not a semigroup homomorphism, then fail is returned. For example, this

could happen if *hom* was created using SemigroupIsomorphismByFunction (14.2.9) and a function which does not give a homomorphism.

Example

```
gap> S := Semigroup([(1, 2, 3, 4), (1, 2)]);;
gap> T := Semigroup([(1, 2, 3), (1, 2)]);;
gap> hom := SemigroupHomomorphismByFunction(S, T,
> function(x) if SignPerm(x) = -1 then return (1, 2);
> else return (); fi; end);
<semigroup of size 24, with 2 generators> ->
<semigroup of size 6, with 2 generators>
```

### 14.1.6 AsSemigroupHomomorphismByFunction (for a semigroup homomorphism by images)

▷ AsSemigroupHomomorphismByFunction(*hom*) (operation)

**Returns:** A semigroup homomorphism.

AsSemigroupHomomorphismByFunction takes *hom*, a semigroup homomorphism, and returns the same mapping but described by a GAP function mapping elements of Source(*hom*) to their images in Range(*hom*).

Example

```
gap> T := TrivialSemigroup();;
gap> S := GLM(2, 2);;
gap> gens := GeneratorsOfSemigroup(S);;
gap> imgs := ListX(gens, x -> IdentityTransformation);;
gap> hom := SemigroupHomomorphismByImages(S, T, gens, imgs);;
gap> hom := AsSemigroupHomomorphismByFunction(hom);
<general linear monoid 2x2 over GF(2)> ->
<trivial transformation group of degree 0 with 1 generator>
```

### 14.1.7 KernelOfSemigroupHomomorphism

▷ KernelOfSemigroupHomomorphism(*hom*) (attribute)

**Returns:** A semigroup congruence.

KernelOfSemigroupHomomorphism returns the kernel of the semigroup homomorphism *hom*. The kernel of a semigroup homomorphism *hom* is a semigroup congruence relating pairs of elements in Source(*hom*) mapping to the same element under *hom*.

Example

```
gap> S := Semigroup([Transformation([2, 1, 5, 1, 5]),
>       Transformation([1, 1, 1, 5, 3]),
>       Transformation([2, 5, 3, 5, 3])]);;
gap> congs := CongruencesOfSemigroup(S);;
gap> cong := congs[3];;
gap> T := S / cong;;
gap> gens := GeneratorsOfSemigroup(S);;
gap> images := List(gens, gen -> EquivalenceClassOfElement(cong, gen));;
gap> hom1 := SemigroupHomomorphismByImages(S, T, gens, images);;
gap> cong = KernelOfSemigroupHomomorphism(hom1);
true
```

## 14.2 Isomorphisms of arbitrary semigroups

### 14.2.1 IsIsomorphicSemigroup

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

**Returns:** `true` or `false`.

If $S$ and $T$ are semigroups, then this operation attempts to determine whether $S$ and $T$ are isomorphic semigroups by using the operation `IsomorphismSemigroups` (14.2.6). If `IsomorphismSemigroups(S, T)` returns an isomorphism, then `IsIsomorphicSemigroup(S, T)` returns `true`, while if `IsomorphismSemigroups(S, T)` returns `fail`, then `IsIsomorphicSemigroup(S, T)` returns `false`.

Note that in some cases, at present, there is no method for determining whether $S$ is isomorphic to $T$, even if it is obvious to the user whether or not $S$ and $T$ are isomorphic. There are plans to improve this in the future.

Example

```
gap> S := Semigroup(PartialPerm([1, 2, 4], [1, 3, 5]),
>                   PartialPerm([1, 3, 5], [1, 2, 4]));;
gap> T := AsSemigroup(IsTransformationSemigroup, S);;
gap> IsIsomorphicSemigroup(S, T);
true
gap> IsIsomorphicSemigroup(FullTransformationMonoid(4),
> PartitionMonoid(4));
false
```

### 14.2.2 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` returns the lex-least multiplication of any semigroup isomorphic to $S$. 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.

From Version 3.3.0 of Semigroups this attribute is computed using `MinimalImage` (**images: MinimalImage**) from the images package. See also: `CanonicalMultiplicationTable` (14.2.3).

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 ] ]
```

### 14.2.3 CanonicalMultiplicationTable

▷ `CanonicalMultiplicationTable(`*S*`)` (attribute)

**Returns:** A canonical multiplication table (up to isomorphism) of a semigroup.

This function returns a multiplication table of a semigroup isomorphic to the semigroup *S*. `CanonicalMultiplicationTable` returns a multiplication that is canonical, in the sense that if two semigroups `S` and `T` are isomorphic, then the return values of `CanonicalMultiplicationTable` are equal.

`CanonicalMultiplicationTable` uses the machinery for canonical labelling of vertex coloured digraphs in bliss via `BlissCanonicalLabelling` (**Digraphs: BlissCanonicalLabelling for a digraph and a list**).

The multiplication table returned by this function is the result of `OnMultiplicationTable(MultiplicationTable(`*S*`), CanonicalMultiplicationTablePerm(`*S*`));`

Note that the performance of `CanonicalMultiplicationTable` is vastly superior to that of `SmallestMultiplicationTable`.

See also: `CanonicalMultiplicationTablePerm` (14.2.4) and `OnMultiplicationTable` (14.2.5).

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> CanonicalMultiplicationTable(S);
[ [ 1, 2, 3, 2, 1, 6, 3, 2 ], [ 1, 2, 3, 2, 1, 6, 3, 2 ],
  [ 1, 2, 3, 2, 1, 6, 3, 1 ], [ 1, 2, 3, 4, 5, 6, 7, 4 ],
  [ 1, 2, 3, 4, 5, 6, 7, 4 ], [ 1, 2, 3, 2, 1, 6, 3, 1 ],
  [ 1, 2, 3, 4, 5, 6, 7, 5 ], [ 1, 2, 3, 4, 5, 6, 7, 4 ] ]
```

### 14.2.4 CanonicalMultiplicationTablePerm

▷ `CanonicalMultiplicationTablePerm(`*S*`)` (attribute)

**Returns:** A permutation.

This function returns a permutation `p` such that `OnMultiplicationTable(MultiplicationTable(`*S*`), p);` equals `CanonicalMultiplicationTable(`*S*`)`.

See `CanonicalMultiplicationTable` (14.2.3) for more details.

`CanonicalMultiplicationTablePerm` uses the machinery for canonical labelling of vertex coloured digraphs in bliss via `BlissCanonicalLabelling` (**Digraphs: BlissCanonicalLabelling for a digraph and a list**).

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> CanonicalMultiplicationTablePerm(S);
(1,5,3,4)(2,7)
```

### 14.2.5 OnMultiplicationTable

▷ OnMultiplicationTable(*table*, *p*) (operation)
▷ PermuteMultiplicationTable(*output*, *table*, *p*) (operation)
▷ PermuteMultiplicationTableNC(*output*, *table*, *p*) (operation)

If *table* is a multiplication table of a semigroup and *p* is a permutation of [1 .. Length(*table*)], then OnMultiplicationTable returns the multiplication table obtained from *table* where the elements [1 .. Length(*table*)] are relabelled according to *p*.

PermuteMultiplicationTable works similarly but takes an additional argument *output*, which should be a list of mutable lists of the same dimensions as *table*. The argument *output* is modified in-place to store the multiplication table that would be given by OnMultiplicationTable(*table*, *p*), while the argument *table* remains unchanged. This approach is preferred over OnMultiplicationTable when the user does not need to create a separate result variable.

PermuteMultiplicationTableNC is the operation called by PermuteMultiplicationTable. It assumes that the arguments *output*, *table* and *p* are well-formed and can therefore lead to unexpected results.

Example

```
gap> table := [[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]];;
gap> p := (1, 2, 3, 4)(10, 11, 12);;
gap> OnMultiplicationTable(table, p);
[ [ 1, 2, 4, 4, 5, 6, 7, 8 ], [ 1, 2, 2, 4, 5, 6, 7, 8 ],
  [ 1, 2, 2, 4, 5, 6, 7, 8 ], [ 1, 2, 2, 4, 5, 6, 7, 8 ],
  [ 7, 5, 5, 6, 5, 6, 7, 8 ], [ 7, 5, 5, 6, 5, 6, 7, 8 ],
  [ 7, 5, 6, 6, 5, 6, 7, 8 ], [ 7, 5, 6, 6, 5, 6, 7, 8 ] ]
gap> temp := List([1 .. Size(table)], x -> [1 .. Size(table)]);;
gap> PermuteMultiplicationTable(temp, table, p);;
gap> temp = OnMultiplicationTable(table, p);
true
```

### 14.2.6 IsomorphismSemigroups

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

**Returns:** An isomorphism, or fail.

This operation attempts to find an isomorphism from the semigroup *S* to the semigroup *T*. If it finds one, then it is returned, and if not, then fail is returned.

IsomorphismSemigroups uses the machinery for finding isomorphisms between vertex coloured digraphs in bliss via IsomorphismDigraphs (**Digraphs: IsomorphismDigraphs for digraphs and homogeneous lists**) using digraphs constructed from the multiplication tables of $S$ and $T$.

Note that finding an isomorphism between two semigroups is difficult, and may not be possible for semigroups whose size exceeds a few hundred elements. On the other hand, IsomorphismSemigroups may be able to deduce that $S$ and $T$ are not isomorphic by finding that some of their semigroup-theoretic properties differ.

Example

```
gap> S := RectangularBand(IsTransformationSemigroup, 4, 5);
<regular transformation semigroup of size 20, degree 9 with 5
 generators>
gap> T := RectangularBand(IsBipartitionSemigroup, 4, 5);
<regular bipartition semigroup of size 20, degree 3 with 5 generators>
gap> IsomorphismSemigroups(S, T) <> fail;
true
gap> D := DClass(FullTransformationMonoid(5),
>                Transformation([1, 2, 3, 4, 1]));;
gap> S := PrincipalFactor(D);;
gap> StructureDescription(UnderlyingSemigroup(S));
"S4"
gap> S;
<Rees 0-matrix semigroup 10x5 over S4>
gap> D := DClass(PartitionMonoid(5),
> Bipartition([[1], [2, -2], [3, -3], [4, -4], [5, -5], [-1]]));;
gap> T := PrincipalFactor(D);;
gap> StructureDescription(UnderlyingSemigroup(T));
"S4"
gap> T;
<Rees 0-matrix semigroup 15x15 over S4>
gap> IsomorphismSemigroups(S, T);
fail
gap> I := SemigroupIdeal(FullTransformationMonoid(5),
>                        Transformation([1, 1, 2, 3, 4]));;
gap> T := PrincipalFactor(DClass(I, I.1));;
gap> StructureDescription(UnderlyingSemigroup(T));
"S4"
gap> T;
<Rees 0-matrix semigroup 10x5 over S4>
gap> IsomorphismSemigroups(S, T) <> fail;
true
```

### 14.2.7 AutomorphismGroup (for a semigroup)

▷ AutomorphismGroup($S$) (operation)

**Returns:** A group.

This operation returns the group of automorphisms of the semigroup $S$. AutomorphismGroup uses bliss via AutomorphismGroup (**Digraphs: AutomorphismGroup for a digraph and a homogeneous list**) using a vertex coloured digraph constructed from the multiplication table of $S$. Consequently, this method is only really feasible for semigroups whose size does not exceed a few hundred elements.

Example

```
gap> S := RectangularBand(IsTransformationSemigroup, 4, 5);
<regular transformation semigroup of size 20, degree 9 with 5
 generators>
gap> StructureDescription(AutomorphismGroup(S));
"S4 x S5"
```

### 14.2.8 SemigroupIsomorphismByImages (for two semigroups and two lists)

▷ SemigroupIsomorphismByImages(*S*, *T*, *gens*, *imgs*) (operation)
▷ SemigroupIsomorphismByImages(*S*, *T*, *imgs*) (operation)
▷ SemigroupIsomorphismByImages(*S*, *T*) (operation)
▷ SemigroupIsomorphismByImages(*S*, *gens*, *imgs*) (operation)

**Returns:** A semigroup isomorphism, or fail.

SemigroupIsomorphismByImages attempts to construct a isomorphism from the semigroup *S* to the semigroup *T*, by mapping the i-th element of *gens* to the i-th element of *imgs*. If this mapping corresponds to an isomorphism, the isomorphism is returned, and if not, then fail is returned. An isomorphism is a bijective homomorphism. See also SemigroupHomomorphismByImages (14.1.1).

Example

```
gap> S := Semigroup([
>  Matrix(IsNTPMatrix, [[0, 1, 2], [4, 3, 0], [0, 2, 0]], 9, 4),
>  Matrix(IsNTPMatrix, [[1, 1, 0], [4, 1, 1], [0, 0, 0]], 9, 4)]);;
gap> T := AsSemigroup(IsTransformationSemigroup, S);;
gap> iso := SemigroupIsomorphismByImages(S, T);
<semigroup of size 46, 3x3 ntp matrices with 2 generators> ->
<transformation semigroup of size 46, degree 47 with 2 generators>
```

### 14.2.9 SemigroupIsomorphismByFunctionNC

▷ SemigroupIsomorphismByFunctionNC(*S*, *T*, *fun*, *invFun*) (operation)
▷ SemigroupIsomorphismByFunction(*S*, *T*, *fun*, *invFun*) (operation)

**Returns:** A semigroup isomorphism or fail.

SemigroupIsomorphismByFunctionNC returns a semigroup isomorphism with source *S* and range *T*, such that each element s in *S* is mapped to the element *fun*(s), where *fun* is a GAP function, and *invFun* its inverse, mapping *fun*(s) back to s.

The function SemigroupIsomorphismByFunctionNC performs no checks on whether the function actually gives an isomorphism, and so it is possible for this operation to return a mapping from *S* to *T* that is not a homomorphism, or not a bijection, or where the return value of InverseGeneralMapping (**Reference: InverseGeneralMapping**) is not the inverse of the returned function.

The function SemigroupIsomorphismByFunction checks that: the mapping from *S* to *T* defined by *fun* satisfies RespectsMultiplication (**Reference: RespectsMultiplication**); that the function from *T* to *S* defined by *invFun* satisfies RespectsMultiplication (**Reference: RespectsMultiplication**); and that these functions are mutual inverses. This can be expensive. If any of these checks fails, then fail is returned.

Example

```
gap> S := MonogenicSemigroup(IsTransformationSemigroup, 3, 2);;
gap> T := MonogenicSemigroup(IsBipartitionSemigroup, 3, 2);;
```

```
gap> map := x -> T.1 ^ Length(Factorization(S, x));;
gap> inv := x -> S.1 ^ Length(Factorization(T, x));;
gap> iso := SemigroupIsomorphismByFunction(S, T, map, inv);
<commutative non-regular transformation semigroup of size 4, degree 5
  with 1 generator> -> <commutative non-regular block bijection
  semigroup of size 4, degree 6 with 1 generator>
```

### 14.2.10 IsSemigroupIsomorphismByFunction

▷ `IsSemigroupIsomorphismByFunction(`*iso*`)` (filter)

**Returns:** `true` or `false`.

`IsSemigroupIsomorphismByFunction` returns `true` if *hom* satisfies `IsSemigroupHomomorphismByFunction` (14.1.4) and `IsBijective` (**Reference: IsBijective**), and `false` if it does not. Note that this filter may return `true` even if the underlying GAP function does not define a homomorphism. A semigroup isomorphism is a mapping from a semigroup `S` to a semigroup `T` that respects multiplication. This representation describes semigroup isomorphisms internally by using a GAP function mapping elements of `S` to their images in `T`. See `SemigroupIsomorphismByFunction` (14.2.9).

Example

```
gap> S := MonogenicSemigroup(IsTransformationSemigroup, 3, 2);;
gap> T := MonogenicSemigroup(IsBipartitionSemigroup, 3, 2);;
gap> map := x -> T.1 ^ Length(Factorization(S, x));;
gap> inv := x -> S.1 ^ Length(Factorization(T, x));;
gap> iso := SemigroupIsomorphismByFunction(S, T, map, inv);
<commutative non-regular transformation semigroup of size 4, degree 5
  with 1 generator> -> <commutative non-regular block bijection
  semigroup of size 4, degree 6 with 1 generator>
gap> IsSemigroupIsomorphismByFunction(iso);
true
```

### 14.2.11 AsSemigroupIsomorphismByFunction (for a semigroup homomorphism by images)

▷ `AsSemigroupIsomorphismByFunction(`*hom*`)` (operation)

**Returns:** A semigroup isomorphism, or `fail`.

`AsSemigroupIsomorphismByFunction` takes a semigroup homomorphism *hom* and returns a semigroup isomorphism represented using GAP functions for the isomorphism and its inverse. If *hom* is not bijective, then `fail` is returned.

Example

```
gap> S := FullTransformationMonoid(3);;
gap> gens := GeneratorsOfSemigroup(S);;
gap> imgs := ListWithIdenticalEntries(4, ConstantTransformation(3, 1));;
gap> hom := SemigroupHomomorphismByImages(S, S, gens, gens);;
gap> AsSemigroupIsomorphismByFunction(hom);
<full transformation monoid of degree 3> ->
<full transformation monoid of degree 3>
```

### 14.2.12 SmallerDegreeTransformationRepresentation

▷ SmallerDegreeTransformationRepresentation(*S*) (attribute)

**Returns:** An isomorphism to a transformation semigroup.

This function attempts to find a small degree transformation representation of the semigroup *S*. The implementation attempts to find a right congruence of *S* that *S* acts on (the equivalence classes of) faithfully.

If *S* is not a finitely presented semigroup, then the returned isomorphism is the composition of an isomorphism to a finitely presented semigroup and an isomorphism from that finitely presented semigroup to a transformation semigroup.

The runtime of this function depends on the presentation for *S* that is either given explicitly or computed by the Semigroups package, but it is difficult to predict what properties of the presentation lead to a shorter runtime. This is unlikely to terminate in a reasonable amount of time for semigroups with more than approx. 10000 elements, but might also not terminate quickly for smaller semigroups depending on the presentation used.

Example

```
gap> S := BrauerMonoid(3);
<regular bipartition *-monoid of degree 3 with 3 generators>
gap> IsomorphismTransformationSemigroup(S);
<regular bipartition *-monoid of size 15, degree 3 with 3 generators>
-> <transformation monoid of size 15, degree 15 with 3 generators>
gap> SmallerDegreeTransformationRepresentation(S);
CompositionMapping(
<fp semigroup with 4 generators and 20 relations of length 81> ->
<transformation monoid of degree 7 with 3 generators>,
<regular bipartition *-monoid of size 15, degree 3 with 3 generators>
-> <fp semigroup with 4 generators and 20 relations of length 81> )
gap> S := JonesMonoid(5);
<regular bipartition *-monoid of degree 5 with 4 generators>
gap> Size(S);
42
gap> SmallerDegreeTransformationRepresentation(S);
CompositionMapping(
<fp semigroup with 5 generators and 28 relations of length 120> ->
<transformation monoid of degree 10 with 4 generators>,
<regular bipartition *-monoid of size 42, degree 5 with 4 generators>
-> <fp semigroup with 5 generators and 28 relations of length 120> )
```

### 14.2.13 MinimalFaithfulTransformationDegree

▷ MinimalFaithfulTransformationDegree(*S*) (attribute)

**Returns:** A positive integer.

This function returns the minimal degree of a faithful transformation representation of the semigroup *S*. This is currently only implemented for a very small number of types of semigroups.

Example

```
gap> S := RightZeroSemigroup(10);
<transformation semigroup of degree 7 with 10 generators>
gap> MinimalFaithfulTransformationDegree(S);
7
```

## 14.3 Isomorphisms of Rees (0-)matrix semigroups

An isomorphism between two regular finite Rees (0-)matrix semigroups whose underlying semigroups are groups can be described by a triple defined in terms of the matrices and underlying groups of the semigroups. For a full description of the theory involved, see Section 3.4 of [How95].

An isomorphism described in this way can be constructed using `RMSIsoByTriple` (14.3.2) or `RZMSIsoByTriple` (14.3.2), and will satisfy the filter `IsRMSIsoByTriple` (14.3.1) or `IsRZMSIsoByTriple` (14.3.1).

### 14.3.1 IsRMSIsoByTriple

▷ `IsRMSIsoByTriple` (category)
▷ `IsRZMSIsoByTriple` (category)

The isomorphisms between finite Rees matrix or 0-matrix semigroups `S` and `T` over groups `G` and `H`, respectively, specified by a triple consisting of:

1. an isomorphism of the underlying graph of `S` to the underlying graph of `T`
2. an isomorphism from `G` to `H`
3. a function from `Rows(S)` union `Columns(S)` to `H`

belong to the categories `IsRMSIsoByTriple` and `IsRZMSIsoByTriple`. Basic operators for such isomorphisms are given in 14.3.7, and basic operations are: `Range` (**Reference: range**), `Source` (**Reference: Source**), `ELM_LIST` (14.3.3), `CompositionMapping` (**Reference: CompositionMapping**), `ImagesElm` (14.3.5), `ImagesRepresentative` (14.3.5), `InverseGeneralMapping` (**Reference: InverseGeneralMapping**), `PreImagesRepresentative` (**Reference: PreImagesRepresentative**), `IsOne` (**Reference: IsOne**).

### 14.3.2 RMSIsoByTriple

▷ `RMSIsoByTriple(`*`R1, R2, triple`*`)` (operation)
▷ `RZMSIsoByTriple(`*`R1, R2, triple`*`)` (operation)

**Returns:** An isomorphism.

If *`R1`* and *`R2`* are isomorphic regular Rees 0-matrix semigroups whose underlying semigroups are groups then `RZMSIsoByTriple` returns the isomorphism between *`R1`* and *`R2`* defined by *`triple`*, which should be a list consisting of the following:

- *`triple`*`[1]` should be a permutation describing an isomorphism from the graph of *`R1`* to the graph of *`R2`*, i.e. it should satisfy `OnDigraphs(RZMSDigraph(`*`R1`*`), `*`triple`*`[1]) = RZMSDigraph(`*`R2`*`)`.
- *`triple`*`[2]` should be an isomorphism from the underlying group of *`R1`* to the underlying group of *`R2`* (see `UnderlyingSemigroup` (**Reference: UnderlyingSemigroup for a Rees 0-matrix semigroup**)).
- *`triple`*`[3]` should be a list of elements from the underlying group of *`R2`*. If the `Matrix` (**Reference: Matrix**) of *`R1`* has $m$ columns and $n$ rows, then the list should have length $m+n$, where the first $m$ entries should correspond to the columns of *`R1`*'s matrix, and the last $n$ entries

should correspond to the rows. These column and row entries should correspond to the $u_i$ and $v_\lambda$ elements in Theorem 3.4.1 of [How95].

If *triple* describes a valid isomorphism from *R1* to *R2* then this will return an object in the category IsRZMSIsoByTriple (14.3.1); otherwise an error will be returned.

If *R1* and *R2* are instead Rees matrix semigroups (without zero) then RMSIsoByTriple should be used instead. This operation is used in the same way, but it should be noted that since an RMS's graph is a complete bipartite graph, *triple*[1] can be any permutation on [1 .. m + n], so long as no point in [1 .. m] is mapped to a point in [m + 1 .. m + n].

Example

```
gap> g := SymmetricGroup(3);;
gap> mat := [[0, 0, (1, 3)], [(1, 2, 3), (), (2, 3)], [0, 0, ()]];;
gap> R := ReesZeroMatrixSemigroup(g, mat);;
gap> id := IdentityMapping(g);;
gap> g_elms_list := [(), (), (), (), (), ()];;
gap> RZMSIsoByTriple(R, R, [(), id, g_elms_list]);
((), IdentityMapping( SymmetricGroup( [ 1 .. 3 ] ) ),
[ (), (), (), (), (), () ])
```

### 14.3.3 ELM_LIST (for IsRMSIsoByTriple)

▷ ELM_LIST(*map*, *pos*) (operation)

**Returns:** A component of an isomorphism of Rees (0-)matrix semigroups by triple.

ELM_LIST(*map*, *i*) returns the ith component of the Rees (0-)matrix semigroup isomorphism by triple *map* when i = 1, 2, 3.

The components of an isomorphism of Rees (0-)matrix semigroups by triple are:

1. An isomorphism of the underlying graphs of the source and range of *map*, respectively.
2. An isomorphism of the underlying groups of the source and range of *map*, respectively.
3. A function from the union of the rows and columns of the source of *map* to the underlying group of the range of *map*.

### 14.3.4 CompositionMapping2 (for IsRMSIsoByTriple)

▷ CompositionMapping2(*map1*, *map2*) (operation)

▷ CompositionMapping2(*map1*, *map2*) (operation)

**Returns:** A Rees (0-)matrix semigroup by triple.

If *map1* and *map2* are isomorphisms of Rees matrix or 0-matrix semigroups specified by triples and the range of *map2* is contained in the source of *map1*, then CompositionMapping2(*map1*, *map2*) returns the isomorphism from Source(*map2*) to Range(*map1*) specified by the triple with components:

1. *map1*[1] * *map2*[1]
2. *map1*[2] * *map2*[2]
3. the componentwise product of *map1*[1] * *map2*[3] and *map1*[3] * *map2*[2].

Example

```
gap> R := ReesZeroMatrixSemigroup(Group([(1, 2, 3, 4)]),
> [[(1, 3)(2, 4), (1, 4, 3, 2), (), (1, 2, 3, 4), (1, 3)(2, 4), 0],
>  [(1, 4, 3, 2), 0, (), (1, 4, 3, 2), (1, 2, 3, 4), (1, 2, 3, 4)],
>  [(), (), (1, 4, 3, 2), (1, 2, 3, 4), 0, (1, 2, 3, 4)],
>  [(1, 2, 3, 4), (1, 4, 3, 2), (1, 2, 3, 4), 0, (), (1, 2, 3, 4)],
>  [(1, 3)(2, 4), (1, 2, 3, 4), 0, (), (1, 4, 3, 2), (1, 2, 3, 4)],
>  [0, (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), ()]]);
<Rees 0-matrix semigroup 6x6 over Group([ (1,2,3,4) ])>
gap> G := AutomorphismGroup(R);;
gap> G.2;
((), IdentityMapping( Group( [ (1,2,3,4) ] ) ),
[ (), (), (), (), (), (), (), (), (), (), (), () ])
gap> G.3;
(( 2, 4)( 3, 5)( 8,10)( 9,11), GroupHomomorphismByImages( Group(
[ (1,2,3,4) ] ), Group( [ (1,2,3,4) ] ), [ (1,2,3,4) ],
[ (1,2,3,4) ] ), [ (), (1,3)(2,4), (1,3)(2,4), (1,3)(2,4),
  (1,3)(2,4), (1,3)(2,4), (), (1,3)(2,4), (1,3)(2,4), (1,3)(2,4),
  (1,3)(2,4), (1,3)(2,4) ])
gap> CompositionMapping2(G.2, G.3);
(( 2, 4)( 3, 5)( 8,10)( 9,11), GroupHomomorphismByImages( Group(
[ (1,2,3,4) ] ), Group( [ (1,2,3,4) ] ), [ (1,2,3,4) ],
[ (1,2,3,4) ] ), [ (), (1,3)(2,4), (1,3)(2,4), (1,3)(2,4),
  (1,3)(2,4), (1,3)(2,4), (), (1,3)(2,4), (1,3)(2,4), (1,3)(2,4),
  (1,3)(2,4), (1,3)(2,4) ])
```

### 14.3.5 ImagesElm (for IsRMSIsoByTriple)

▷ `ImagesElm(`*map*`, `*pt*`)` (operation)

▷ `ImagesRepresentative(`*map*`, `*pt*`)` (operation)

**Returns:** An element of a Rees (0-)matrix semigroup or a list containing such an element.

If *map* is an isomorphism of Rees matrix or 0-matrix semigroups specified by a triple and *pt* is an element of the source of *map*, then `ImagesRepresentative(`*map*`, `*pt*`) = `*pt*` ^ `*map* returns the image of *pt* under *map*.

The image of *pt* under *map* in `Range(`*map*`)` is the element with components:

1. *pt*`[1] ^ `*map*`[1]`
2. `(`*pt*`[1] ^ `*map*`[3]) * (`*pt*`[2] ^ `*map*`[2]) * (`*pt*`[3] ^ `*map*`[3]) ^ -1`
3. *pt*`[3] ^ `*map*`[1]`.

`ImagesElm(`*map*`, `*pt*`)` simply returns `[ImagesRepresentative(`*map*`, `*pt*`)]`.

Example

```
gap> R := ReesZeroMatrixSemigroup(Group([(2, 8), (2, 8, 6)]),
> [[0, (2, 8), 0, 0, 0, (2, 8, 6)],
>  [(), 0, (2, 8, 6), (2, 6), (2, 6, 8), 0],
>  [(2, 8, 6), 0, (2, 6, 8), (2, 8), (), 0],
>  [(2, 8, 6), 0, (2, 6, 8), (2, 8), (), 0],
>  [0, (2, 8, 6), 0, 0, 0, (2, 8)],
>  [(2, 8, 6), 0, (2, 6, 8), (2, 8), (), 0]]);
<Rees 0-matrix semigroup 6x6 over Group([ (2,8), (2,8,6) ])>
```

```
gap> map := RZMSIsoByTriple(R, R,
> [(), IdentityMapping(Group([(2, 8), (2, 8, 6)])),
> [(), (2, 6, 8), (), (), (), (2, 8, 6),
>  (2, 8, 6), (), (), (), (2, 6, 8), ()]]);;
gap> ImagesElm(map, RMSElement(R, 1, (2, 8), 2));
[ (1,(2,8),2) ]
```

### 14.3.6 CanonicalReesZeroMatrixSemigroup

▷ `CanonicalReesZeroMatrixSemigroup(`*S*`)` (attribute)
▷ `CanonicalReesMatrixSemigroup(`*S*`)` (attribute)

**Returns:** A Rees zero matrix semigroup.

If *S* is a Rees 0-matrix semigroup then `CanonicalReesZeroMatrixSemigroup` returns an isomorphic Rees 0-matrix semigroup `T` with the same `UnderlyingSemigroup` (**Reference: UnderlyingSemigroup for a Rees 0-matrix semigroup**) as *S* but the `Matrix` (**Reference: Matrix**) of `T` has been canonicalized. The output `T` is canonical in the sense that for any two inputs which are isomorphic Rees zero matrix semigroups the output of this function is the same.

`CanonicalReesMatrixSemigroup` works the same but for Rees matrix semigroups.

Example

```
gap> S := ReesZeroMatrixSemigroup(SymmetricGroup(3),
> [[(), (1, 3, 2)], [(), ()]]);;
gap> T := CanonicalReesZeroMatrixSemigroup(S);
<Rees 0-matrix semigroup 2x2 over Sym( [ 1 .. 3 ] )>
gap> Matrix(S);
[ [ (), (1,3,2) ], [ (), () ] ]
gap> Matrix(T);
[ [ (), () ], [ (), (1,2,3) ] ]
```

### 14.3.7 Operators for isomorphisms of Rees (0-)matrix semigroups

*map*`[`*i*`]`

*map*`[i]` returns the *i*th component of the Rees (0-)matrix semigroup isomorphism by triple *map* when *i* `= 1, 2, 3`; see `ELM_LIST` (14.3.3).

*map1* `*` *map2*

returns the composition of *map2* and *map1*; see `CompositionMapping2` (14.3.4).

*map1* `<` *map2*

returns `true` if *map1* is lexicographically less than *map2*.

*map1* `=` *map2*

returns `true` if the Rees (0-)matrix semigroup isomorphisms by triple *map1* and *map2* have equal source and range, and are equal as functions, and `false` otherwise.

It is possible for *map1* and *map2* to be equal but to have distinct components.

*pt* `^` *map*

returns the image of the element *pt* of the source of *map* under the isomorphism *map*; see `ImagesElm` (14.3.5).

# Chapter 15

# Finitely presented semigroups and Tietze transformations

In this chapter we describe the functions implemented in Semigroups that extend the features available in GAP for dealing with finitely presented semigroups and monoids.

Section 15.1 (written by Maria Tsalakou and Murray Whyte) and Section 15.2 (written by Luke Elliott) demonstrate a number of helper functions that allow the user to convert between different representations of words and relations.

In the later sections, written and implemented by Ben Spiers and Tom Conti-Leslie, we describe how to change the relations of a finitely presented semigroup either manually or automatically using Tietze transformations (which is abbreviated to STZ.

## 15.1 Changing representation for words and strings

This section contains various methods for dealing with words, which for these purposes are lists of positive integers.

### 15.1.1 WordToString (for a string and a list)

▷ `WordToString(A, w)` (operation)

**Returns:** A string.

Returns the word `w`, in the form of a string of letters of the alphabet `A`. The alphabet is given as a string containing its members.

Example

```
gap> WordToString("abcd", [4, 2, 3, 1, 1, 4, 2, 3]);
"dbcaadbc"
```

### 15.1.2 RandomWord (for two integers)

▷ `RandomWord(l, n)` (operation)

**Returns:** A word.

Returns a random word of length `l` over `n` letters.

Example

```
gap> RandomWord(8, 5);
[ 2, 4, 3, 4, 5, 3, 3, 2 ]
```

```
gap> RandomWord(8, 5);
[ 3, 3, 5, 5, 5, 4, 4, 5 ]
gap> RandomWord(8, 4);
[ 1, 4, 1, 1, 3, 3, 4, 4 ]
```

### 15.1.3 StandardiseWord

▷ StandardiseWord(w) (operation)
▷ StandardizeWord(w) (operation)

**Returns:** A list of positive integers.

This function takes a word w, consisting of n distinct positive integers and returns a word s where the characters of s correspond to those of w in order of first appearance.

The word w is changed in-place into the word s.

Example

```
gap> w := [3, 1, 2];
[ 3, 1, 2 ]
gap> StandardiseWord(w);
[ 1, 2, 3 ]
gap> w;
[ 1, 2, 3 ]
gap> w := [4, 2, 10, 2];
[ 4, 2, 10, 2 ]
gap> StandardiseWord(w);
[ 1, 2, 3, 2 ]
```

### 15.1.4 StringToWord (for a string)

▷ StringToWord(s) (operation)

**Returns:** A list of positive integers.

This function takes a string s, consisting of n distinct characters, and returns a word w (i.e. a list of positive integers) over the alphabet [1 .. n]. The positive integers of w correspond to the characters of s, in order of first appearance.

Example

```
gap> w := "abac";
"abac"
gap> StringToWord(w);
[ 1, 2, 1, 3 ]
gap> w := "ccala";
"ccala"
gap> StringToWord(w);
[ 1, 1, 2, 3, 2 ]
gap> w := "a1b5";
"a1b5"
gap> StringToWord(w);
[ 1, 2, 3, 4 ]
```

## 15.2 Helper functions

This section describes operations implemented in Semigroups that are designed to interact with standard GAP methods for creating finitely presented semigroups and monoids (see (**Reference: Finitely Presented Semigroups and Monoids**)).

### 15.2.1 ParseRelations

▷ `ParseRelations(`*`gens, rels`*`)` (operation)

**Returns:** A list of pairs of semigroup or monoid elements.

`ParseRelations` converts a string describing relations for a semigroup or monoid to the list of pairs of semigroup or monoid elements it represents. Any white space given is ignored. The output list is then compatible with other GAP functions. In the below examples we see free semigroups and monoids being directly quotiented by the output of the `ParseRelations` function.

The argument *`gens`* must be a list of generators for a free semigroup, each being a single alphabet letter (in upper or lower case). The argument *`rels`* must be a string that lists the equalities desired.

To take a quotient of a free monoid, it is necessary to use `GeneratorsOfMonoid` (**Reference: GeneratorsOfMonoid**) as the 1st argument to `ParseRelations` and the identity may appear as `1` in the specified relations.

Example

```
gap> f := FreeSemigroup("x", "y", "z");;
gap> AssignGeneratorVariables(f);
gap> ParseRelations([x, y, z], "  x=(y^2z) ^2x, y=xxx , z=y^3");
[ [ x, (y^2*z)^2*x ], [ y, x^3 ], [ z, y^3 ] ]
gap> r := ParseRelations([x, y, z], "  x=(y^2z)^2x, y=xxx=z , z=y^3");
[ [ x, (y^2*z)^2*x ], [ y, x^3 ], [ x^3, z ], [ z, y^3 ] ]
gap> f / r;
<fp semigroup with 3 generators and 4 relations of length 23>
gap> f2 := FreeSemigroup("a");
<free semigroup on the generators [ a ]>
gap> f2 / ParseRelations(GeneratorsOfSemigroup(f2), "a = a^2");
<fp semigroup with 1 generator and 1 relation of length 4>
gap> FreeMonoidAndAssignGeneratorVars("a", "b")
> / ParseRelations([a, b], "a^2=1,b^3=1,(ab)^3=1");
<fp monoid with 2 generators and 3 relations of length 13>
```

### 15.2.2 ElementOfFpSemigroup

▷ `ElementOfFpSemigroup(`*`S, word`*`)` (operation)

**Returns:** An element of the fp semigroup *`S`*.

When *`S`* is a finitely presented semigroup and *`word`* is an associative word in the associated free semigroup (see `IsAssocWord` (**Reference: IsAssocWord**)), this returns the fp semigroup element with representative *`word`*.

This function is just a short form of the GAP library implementation of `ElementOfFpSemigroup` (**Reference: ElementOfFpSemigroup**) which does not require retrieving an element family.

Example

```
gap> f := FreeSemigroup("x", "y");;
gap> AssignGeneratorVariables(f);
gap> s := f / [[x * x, x], [y * y, y]];
```

```
<fp semigroup with 2 generators and 2 relations of length 8>
gap> a := ElementOfFpSemigroup(s, x * y);
x*y
gap> b := ElementOfFpSemigroup(s, x * y * y);
x*y^2
gap> a in s;
true
gap> a = b;
true
```

### 15.2.3 ElementOfFpMonoid

▷ ElementOfFpMonoid(*M*, *word*) (operation)

**Returns:** An element of the fp monoid *M*.

When *M* is a finitely presented monoid and *word* is an associative word in the associated free monoid (see IsAssocWord (**Reference: IsAssocWord**)), this returns the fp monoid element with representative *word*.

This is analogous to ElementOfFpSemigroup (15.2.2).

### 15.2.4 FreeMonoidAndAssignGeneratorVars

▷ FreeMonoidAndAssignGeneratorVars(*arg...*) (function)

▷ FreeSemigroupAndAssignGeneratorVars(*arg...*) (function)

**Returns:** A free semigroup or monoid.

FreeMonoidAndAssignGeneratorVars is synonymous with:

Example

```
FreeMonoid(arg...);
AssignGeneratorVariables(last);
```

These functions exist so that the String method for a finitely presented semigroup or monoid to be valid GAP input which can be used to reconstruct the semigroup or monoid.

Example

```
gap> F := FreeSemigroupAndAssignGeneratorVars("x", "y");;
gap> IsBound(x);
true
gap> IsBound(y);
true
```

### 15.2.5 IsSubsemigroupOfFpMonoid

▷ IsSubsemigroupOfFpMonoid(*S*) (property)

**Returns:** true or false.

This property is true if the object *S* is a subsemigroup of an fp monoid, and false otherwise. This property is just a synonym for IsSemigroup (**Reference: IsSemigroup**) and IsElementOfFpMonoidCollection.

Example

```
gap> F := FreeSemigroup("a", "b");
<free semigroup on the generators [ a, b ]>
gap> AssignGeneratorVariables(F);
```

```
gap> R := [[a ^ 3, a], [b ^ 2, b], [(a * b) ^ 2, a]];
[ [ a^3, a ], [ b^2, b ], [ (a*b)^2, a ] ]
gap> S := F / R;
<fp semigroup with 2 generators and 3 relations of length 14>
gap> IsSubsemigroupOfFpMonoid(S);
false
gap> map := EmbeddingFpMonoid(S);
<fp semigroup with 2 generators and 3 relations of length 14> ->
<fp monoid with 2 generators and 3 relations of length 14>
gap> IsSubsemigroupOfFpMonoid(Image(map));
true
gap> IsSubsemigroupOfFpMonoid(Range(map));
true
```

## 15.3 Creating Tietze transformation objects

It is possible to use GAP to create finitely presented semigroups without the Semigroups package, by creating a free semigroup, then quotienting by a list of relations. This is described in the reference manual ( (**Reference: Finitely Presented Semigroups and Monoids**)).

However, finitely presented semigroups do not allow for their relations to be simplified, so in the following sections, we describe how to create and modify the semigroup Tietze (`IsStzPresentation` (15.3.2)) object associated with an fp semigroup. This object can be automatically simplified, or the user can manually apply Tietze transformations to add or remove relations or generators in the presentation.

This object is analogous to `PresentationFpGroup` (**Reference: PresentationFpGroup**) implemented for fp groups in the main GAP distribution ( (**Reference: Presentations and Tietze Transformations**)), but its features are semigroup-specific. Most of the functions used to create, view and manipulate semigroup Tietze objects are prefixed with Stz.

### 15.3.1 StzPresentation

▷ `StzPresentation(`*S*`)` (operation)

**Returns:** A semigroup Tietze (Stz) object.

If *S* is an fp semigroup (`IsFpSemigroup` (**Reference: IsFpSemigroup**)), then this function returns a modifiable object representing the generators and relations of *S*.

Example

```
gap> F := FreeSemigroup("a", "b", "c");;
gap> AssignGeneratorVariables(F);;
gap> S := F / [[a * b, c], [b * c, a], [c * a, b]];
<fp semigroup with 3 generators and 3 relations of length 12>
gap> stz := StzPresentation(S);
<fp semigroup presentation with 3 generators and 3 relations
 with length 12>
```

### 15.3.2 IsStzPresentation

▷ `IsStzPresentation(`*stz*`)` (filter)

**Returns:** `true` or `false`.

Every semigroup Tietze object is an element of the category IsStzPresentation. Internally, each Stz object contains a list of generators (each represented as a string) and a list of relations (each represented as a pair of LetterRep words, see LetterRepAssocWord (**Reference: LetterRepAssocWord**)). These generator and relation lists can be modified using Tietze transformations (15.5).

When a IsStzPresentation object *stz* is created from an fp semigroup s using stz := StzPresentation(s), the generators and relations of *stz* are initially equal to the generators and relations of s. However, as the Stz object *stz* is modified, these lists may change, and their current state can be viewed using GeneratorsOfStzPresentation (15.3.3) and RelationsOfStzPresentation (15.3.4).

Example

```
gap> F := FreeSemigroup("a", "b", "c");;
gap> AssignGeneratorVariables(F);;
gap> S := F / [[a * b, c], [b * c, a], [c * a, b]];
<fp semigroup with 3 generators and 3 relations of length 12>
gap> stz := StzPresentation(S);
<fp semigroup presentation with 3 generators and 3 relations
 with length 12>
gap> IsStzPresentation(stz);
true
```

### 15.3.3 GeneratorsOfStzPresentation

▷ GeneratorsOfStzPresentation(*stz*) (attribute)

**Returns:** A list of strings.

If *stz* is an StzPresentation (15.3.1) object, then GeneratorsOfStzPresentation will return (as strings) the generators of the fp semigroup that the presentation was created from. In the StzPresentation (15.3.1) object, it is only necessary to know how many generators there are, but for the purposes of representing generators and relations of the presentation object and building a new fp semigroup from the object, the strings representing the generators are stored.

As Tietze transformations are performed on *stz*, the generators will change, but the labels will remain as close to the original labels as possible, so that if a generator in the fp semigroup obtained from the presentation is the same as a generator in the original fp semigroup, then they should have the same label.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup with 3 generators and 2 relations of length 19>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 2 relations
 with length 19>
gap> GeneratorsOfStzPresentation(stz);
[ "a", "b", "c" ]
```

### 15.3.4 RelationsOfStzPresentation

▷ RelationsOfStzPresentation(*stz*) (attribute)

**Returns:** A list of pairs of words in LetterRep (LetterRepAssocWord (**Reference: Letter-**

**RepAssocWord**)) form.

If *stz* is an StzPresentation (15.3.1) object, then RelationsOfStzPresentation will return in letter rep form the current relations of the presentation object. When the presentation object is first created, these will be the LetterRep forms of the relations of the fp semigroup object that is used to create *stz*. As Tietze transformations are performed on the presentation object, the relations returned by this function will change to reflect the transformations.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup with 3 generators and 2 relations of length 19>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 2 relations
 with length 19>
gap> RelationsOfStzPresentation(stz);
[ [ [ 1 ], [ 2, 2, 2, 2, 2, 3 ] ],
  [ [ 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2 ] ] ]
```

### 15.3.5 UnreducedFpSemigroup (for a presentation)

▷ UnreducedFpSemigroup(*stz*) (attribute)

**Returns:** An fp semigroup.

If *stz* is an StzPresentation (15.3.1) object, then UnreducedFpSemigroup will return the fp semigroup that was used to create *stz* using StzPresentation (15.3.1).

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup with 3 generators and 2 relations of length 19>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 2 relations
 with length 19>
gap> UnreducedFpSemigroup(stz) = T;
true
```

### 15.3.6 Length

▷ Length(*stz*) (operation)

**Returns:** A non-negative integer.

If *stz* is an StzPresentation (15.3.1) object, then the Length of the object is defined as the number of generators plus the lengths of each word in each relation of *stz*.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
```

```
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> Length(stz);
22
```

## 15.4 Printing Tietze transformation objects

Since the relations are stored as flat lists of numbers, there are several methods installed to print the presentations in more user-friendly forms.

All printing methods in this section are displayed as information (`Info` (**Reference: Info**)) in the class `InfoFpSemigroup` at level 1. Setting `SetInfoLevevl(InfoFpSemigroup, 0)` will suppress the messages, while any higher number will display them.

### 15.4.1 StzPrintRelations

▷ `StzPrintRelations(`*`stz[, list]`*`)` (operation)

If *stz* is an `StzPresentation` (15.3.1) object and *list* is a list of positive integers, then `StzPrintRelations` prints for each *i* in *list* the *i*th relation to the console in terms of the stored labels for the generators (that is, as words over the alphabet consisting of the generators of *stz*).

If *list* is not specified then `StzPrintRelations` prints all relations of *stz* in order.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzPrintRelations(stz, [2, 3]);
#I  2. b^6 = b^3
#I  3. b^2 = b
gap> StzPrintRelations(stz);
#I  1. a = b^5*c
#I  2. b^6 = b^3
#I  3. b^2 = b
```

### 15.4.2 StzPrintRelation

▷ `StzPrintRelation(`*`stz, int`*`)` (operation)

If *stz* is an `StzPresentation` (15.3.1) object, then `StzPrintRelation` calls `StzPrintRelations` with parameters *stz* and `[`*int*`]`.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
```

```
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzPrintRelation(stz, 2);
#I  2. b^6 = b^3
```

### 15.4.3 StzPrintGenerators

▷ StzPrintGenerators(*stz[, list]*) (operation)

If *stz* is an StzPresentation (15.3.1) object and *list* is a list of positive integers, then StzPrintGenerators for each *i* in *list* the *i*th generator and the number of occurrences of that generator in the relations is printed to the screen.

If *list* is not specified then StzPrintGenerators prints all generators of *stz* in order.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzPrintGenerators(stz, [1, 2]);
#I  1.  a  1 occurrences
#I  2.  b  17 occurrences
gap> StzPrintGenerators(stz);
#I  1.  a  1 occurrences
#I  2.  b  17 occurrences
#I  3.  c  1 occurrences
```

### 15.4.4 StzPrintPresentation

▷ StzPrintPresentation(*stz*) (operation)

If *stz* is an StzPresentation (15.3.1) object, then StzPrintPresentation prints a comprehensive overview of *stz*, including the generators and number of occurrences of each generator in the relations, the relations as words over the generators, and the forward and backward maps that indicate how the unreduced semigroup maps to the semigroup currently described by *stz*.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzPrintPresentation(stz);
```

```
#I  Current generators:
#I  1.  a  1 occurrences
#I  2.  b  17 occurrences
#I  3.  c  1 occurrences
#I
#I  Current relations:
#I  1. a = b^5*c
#I  2. b^6 = b^3
#I  3. b^2 = b
#I
#I  There are 3 generators and 3 relations of total length 22.
#I
#I  Generators of original fp semigroup expressed as
#I  combinations of generators in current presentation:
#I  1. a = a
#I  2. b = b
#I  3. c = c
#I
#I  Generators of current presentation expressed as
#I  combinations of generators of original fp semigroup:
#I  1. a = a
#I  2. b = b
#I  3. c = c
```

## 15.5 Changing Tietze transformation objects

Fundamentally, there are four different changes that can be made to a presentation without changing the algebraic structure of the fp semigroup that can be derived from it. These four changes are called Tietze transformations, and they are primarily implemented in this section as operations on an `StzPresentation` object that will throw errors if the conditions have not been met to perform the Tietze transformation.

However, the checks required in order to ensure that a Tietze transformation is valid sometimes require verifying equality of two words in an fp semigroup (for example, to ensure that a relation we are adding to the list of relations can be derived from the relations already present). Since these checks sometimes do not terminate, a second implementation of Tietze transformations assumes good faith and does not perform any checks to see whether the requested Tietze transformation actually maintains the structure of the semigroup. This latter type should be used at the user's discretion. If only the first type are used, the presentation will always give a semigroup isomorphic to the one used to create the object, but if instead one is not changing the presentation with the intention of maintaining algebraic structure, these `no-check` functions are available for use.

The four Tietze transformations on a presentation are adding a relation, removing a relation, adding a generator, and removing a generator, with particular conditions on what can be added/removed in order to maintain structure. More details on each transformation and its arguments and conditions is given in each entry below. In addition to the four elementary transformations, there is an additional function `StzSubstituteRelation` which applies multiple Tietze transformations in sequence.

### 15.5.1 StzAddRelation

▷ StzAddRelation(*stz*, *pair*) (operation)

If *stz* is an StzPresentation (15.3.1) object and *pair* is a list containing two LetterRep words over the generators of *stz*, then StzAddRelation will perform a Tietze transformation of the first type and add a new relation to *stz*. This only happens if the new relation that would be formed from *pair* can be constructed from the other existing relations; that is, if we can perform elementary operations using the existing relations of *stz* to convert *pair*[1] into *pair*[2].

If, instead, *pair* is a list containing two elements of the fp semigroup S that was used to create stz, and the two words are equal in that semigroup, then this function will add the LetterRep of these words as a new relation to *stz*.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup with 3 generators and 2 relations of length 19>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 2 relations
 with length 19>
gap> pair := [[2, 2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2]];;
gap> StzAddRelation(stz, pair);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1 ], [ 2, 2, 2, 2, 2, 3 ] ],
  [ [ 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2 ] ],
  [ [ 2, 2, 2, 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2 ] ] ]
gap> pair2 := [[1, 1], [3]];
[ [ 1, 1 ], [ 3 ] ]
gap> StzAddRelation(stz, pair2);
Error, StzAddRelation: second argument <pair> must list two
words that are equal in the presentation <stz>
```

### 15.5.2 StzRemoveRelation

▷ StzRemoveRelation(*stz*, *index*) (operation)

If *stz* is an StzPresentation (15.3.1) object and *index* is a positive integer less than or equal to the number of relations of *stz*, then StzRemoveRelation will perform a Tietze transformation of the second type and remove the *index*th relation of *stz* if that relation is such that one side of it can be obtained from the other by a sequence of elementary operations using only the other relations of *stz*.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
```

```
gap> StzRemoveRelation(stz, 2);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1 ], [ 2, 2, 2, 2, 2, 3 ] ], [ [ 2, 2 ], [ 2 ] ] ]
gap> StzRemoveRelation(stz, 2);
Error, StzRemoveRelation: second argument <index> must point to
a relation that is redundant in the presentation <stz>
```

### 15.5.3 StzAddGenerator

▷ StzAddGenerator(*stz, word[, name]*) (operation)

If *stz* is an StzPresentation (15.3.1) object and *word* is a LetterRep word over the generators of *stz*, then StzAddGenerator will perform a Tietze transformation which adds a new generator to *stz* and a new relation of the form newgenerator = *word*.

If, instead, *word* is a word over the fp semigroup *S* that was used to create *stz*, then this function will add the new generator and a new relation with the new generator equal to the LetterRep of this word as a new relation to *stz*.

A new name for the generator is chosen and added automatically based on the names of the existing generators to GeneratorsOfStzPresentation (15.3.3) if the argument *name* is not included. If it is, and if *name* is a string that is not equal to an existing generator, then the string added to the list of generators will be *name* instead.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzAddGenerator(stz, [2, 2, 2]);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1 ], [ 2, 2, 2, 2, 2, 3 ] ],
  [ [ 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2 ] ], [ [ 2, 2 ], [ 2 ] ],
  [ [ 2, 2, 2 ], [ 4 ] ] ]
```

### 15.5.4 StzRemoveGenerator

▷ StzRemoveGenerator(*stz, gen/genname[, index]*) (operation)

If *stz* is an StzPresentation (15.3.1) object and *gen* is a positive integer less than or equal to the number of generators of *stz*, then StzRemoveGenerator will perform a Tietze transformation which removes the *gen*th generator of *stz*.

The argument *stz* must contain a relation of the form *gen* = word or word = *gen*, where word contains no occurrences of the generator *gen* being removed. The generator is then removed from the presentation by replacing every instance of *gen* with a copy of word.

If the second argument is a string *genname* rather than a positive integer *gen*, then the function searches the generators of *stz* for a generator with the same name and attempts to remove the generator if the same conditions as above are met.

If the argument *index* is included and is a positive integer less than or equal to the number of relations, then rather than searching the relations for the first to satisfy the necessary conditions, the function checks the *index*th relation to see if it satisfies those conditions, and applies the Tietze transformation by removing this relation.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzRemoveGenerator(stz, 1);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1 ] ], [ [ 1, 1 ], [ 1 ] ] ]
```

### 15.5.5 StzSubstituteRelation

▷ StzSubstituteRelation(*stz*, *index*, *side*) (operation)

If *stz* is an StzPresentation (15.3.1) object and *index* is a positive integer less than or equal to the number of relations of *stz* and *side* is either 1 or 2, then StzSubstituteRelation will perform a sequence of Tietze transformations in order to replace, for the *index*th relation (say [u, v]), to replace all instances of the *side*th word of the relation in all other relations by the other side (so, for *side*=1, all instances of u in all other relations of *stz* are replaced by v). This requires two Tietze transformations per relation containing u, one to add a new redundant relation with each u replaced by v, and another to remove the original (now redundant) relation.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3], [F.2 ^ 2, F.2]];
<fp semigroup with 3 generators and 3 relations of length 22>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 3 relations
 with length 22>
gap> StzSubstituteRelation(stz, 3, 1);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1 ], [ 2, 2, 2, 3 ] ], [ [ 2, 2, 2 ], [ 2, 2 ] ],
  [ [ 2, 2 ], [ 2 ] ] ]
gap> StzSubstituteRelation(stz, 3, 1);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1 ], [ 2, 2, 3 ] ], [ [ 2, 2 ], [ 2 ] ], [ [ 2, 2 ], [ 2 ] ] ]
```

## 15.6 Converting a Tietze transformation object into a fp semigroup

Semigroup Tietze transformation objects (IsStzPresentation (15.3.2)) are not actual fp semigroups in the sense of IsFpSemigroup (**Reference: IsFpSemigroup**). This is because their generators and

relations can be modified (see section 15.5). However, an StzPresentation (15.3.1) can be converted back into an actual finitely presented semigroup using the methods described in this section.

The intended use of semigroup Tietze transformation objects is as follows: given an fp semigroup S, create a modifiable presentation using stz := StzPresentation(S), apply Tietze transformations to it (perhaps in order to simplify the presentation), then generate a new fp semigroup T given by stz which is isomorphic to S, but has a simpler presentation. Once T is obtained, it may be of interest to map elements of S over to T, where they may be represented by different combinations of generators. The isomorphism achieving this is described in this section (see StzIsomorphism (15.6.3)).

### 15.6.1 TietzeForwardMap

▷ TietzeForwardMap(*stz*) (attribute)

**Returns:** A list of lists of positive integers.

If *stz* is an StzPresentation (15.3.1) object, then TietzeForwardMap returns a list of lists of positive integers. There is an element of this list for every generator of the unreduced semigroup of the presentation (see UnreducedFpSemigroup (15.3.5)) that indicates the word (in LetterRep form) in the semigroup object currently defined by the presentation that the generator maps to.

This mapping is updated as the presentation object is transformed. It begins as a list of the form [[1], [2], [3], . . ., [n]] where n is the number of generators of the unreduced semigroup.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> S := F / [[F.1, F.3 ^ 3], [F.2 ^ 2, F.3 ^ 2]];
<fp semigroup with 3 generators and 2 relations of length 11>
gap> stz := StzPresentation(S);
<fp semigroup presentation with 3 generators and 2 relations
 with length 11>
gap> StzRemoveGenerator(stz, 1);
gap> TietzeForwardMap(stz);
[ [ 2, 2, 2 ], [ 1 ], [ 2 ] ]
```

### 15.6.2 TietzeBackwardMap

▷ TietzeBackwardMap(*stz*) (attribute)

**Returns:** A list of lists of positive integers.

If *stz* is an StzPresentation (15.3.1) object, then TietzeBackwardMap returns a list of lists of positive integers. There is an element of this list for every generator of the semigroup that the presentation currently defines that indicates the word (in LetterRep form) in the unreduced semigroup of the presentation (see UnreducedFpSemigroup (15.3.5)) that the generator maps to.

This mapping is updated as the presentation object is transformed. It begins as a list of the form [[1], [2], [3], . . ., [n]] where n is the number of generators of the unreduced semigroup.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> S := F / [[F.1, F.3 ^ 3], [F.2 ^ 2, F.3 ^ 2]];
<fp semigroup with 3 generators and 2 relations of length 11>
gap> stz := StzPresentation(S);
```

```
<fp semigroup presentation with 3 generators and 2 relations
 with length 11>
gap> StzRemoveGenerator(stz, 1);
gap> TietzeBackwardMap(stz);
[ [ 2 ], [ 3 ] ]
```

### 15.6.3 StzIsomorphism

▷ StzIsomorphism(*stz*) (operation)

**Returns:** A mapping object.

If *stz* is an StzPresentation (15.3.1) object, then StzIsomorphism returns a mapping object that maps the unreduced semigroup of the presentation (see UnreducedFpSemigroup (15.3.5)) to an FpSemigroup object that is defined by the generators and relations of the semigroup presentation at the moment this function is run.

If a StzIsomorphism is generated from *stz*, and the presentation *stz* is further modified afterwards (for example by applying more Tietze transformations or StzSimplifyOnce (15.7.1) to *stz*), then running StzIsomorphism(*stz*) a second time will produce a different result consistent with the *new* generators and relations of *stz*.

This mapping is built from the TietzeForwardMap (15.6.1) and TietzeBackwardMap (15.6.2) attributes from the presentation object, since if we know how to map the generators of the respective semigroups, then we know how to map any element of that semigroup.

This function is the primary way to obtain the simplified semigroup from the presentation object, by applying Range to the mapping that this function returns.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> S := F / [[F.1, F.3 ^ 3], [F.2 ^ 2, F.3 ^ 2]];
<fp semigroup with 3 generators and 2 relations of length 11>
gap> stz := StzPresentation(S);
<fp semigroup presentation with 3 generators and 2 relations
 with length 11>
gap> StzRemoveGenerator(stz, "a");
gap> map := StzIsomorphism(stz);
<fp semigroup with 3 generators and 2 relations of length 11> ->
<fp semigroup with 2 generators and 1 relation of length 6>
gap> S.1 ^ map;
c^3
```

## 15.7 Automatically simplifying a Tietze transformation object

It is possible to create a presentation object from an fp semigroup object and attempt to manually reduce it by applying Tietze transformations. However, there may be many different reductions that can be applied, so StzSimplifyOnce can be used to automatically check a number of different possible reductions and apply the best one. Then, StzSimplifyPresentation repeatedly applies StzSimplifyOnce to the presentation object until it fails to reduce the presentation any further. The metric with respect to which the IsStzPresentation object is reduced is Length (15.3.6).

### 15.7.1 StzSimplifyOnce

▷ StzSimplifyOnce(*stz*) (operation)

**Returns:** true or false.

If *stz* is an StzPresentation (15.3.1) object, then StzSimplifyOnce will check the possible reductions in length for a number of different possible Tietze transformations, and apply the choice which gives the least length. If a valid transformation was found then the function returns true, and if no transformation was performed because none would lead to a reduction in length, then the function returns false.

There are four different possible transformations that StzSimplifyOnce may apply. The function searches for redundant generators and checks if removing them would reduce the overall length of the presentation, it checks whether substituting one side of each relation throughout the rest of the relations would reduce the length, it checks whether there are any trivial relations (of the form $w = w$ for some word $w$) or any duplicated relations (relations which are formed from precisely the same words as another relation), and it checks whether any frequently occurring subwords in the relations can be replaced with a new generator to reduce the length. For more details, see 15.5

At InfoLevel 2 (which is the default value, and which can be set using SetInfoLevel(InfoFpSemigroup, 2)), the precise transformations performed are printed to the screen.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup with 3 generators and 2 relations of length 19>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 2 relations
 with length 19>
gap> StzSimplifyOnce(stz);
#I  <Removing redundant generator a using relation : 1. a = b^5*c>
true
gap> stz;
<fp semigroup presentation with 2 generators and 1 relation
 with length 11>
```

### 15.7.2 StzSimplifyPresentation

▷ StzSimplifyPresentation(*stz*) (operation)

If *stz* is an StzPresentation object, then StzSimplifyPresentation will repeatedly apply the best of a few possible reductions to *stz* until it can no longer reduce the length of the presentation.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup on the generators [ a, b, c ]>
gap> stz := StzPresentation(T);
<fp semigroup presentation with 3 generators and 2 relations
 with length 19>
```

```
gap> StzSimplifyPresentation(stz);
gap> RelationsOfStzPresentation(stz);
[ [ [ 1, 1, 1 ], [ 3 ] ], [ [ 3, 3 ], [ 3 ] ] ]
```

## 15.8 Automatically simplifying an fp semigroup

It may be the case that, rather than working with a Tietze transformation object, we want to start with an fp semigroup object and obtain the most simplified version of that fp semigroup that StzSimplifyPresentation can produce. In this case, SimplifyFpSemigroup can be applied to obtain a mapping from its argument to a reduced fp semigroup. If the mapping is not of interest, SimplifiedFpSemigroup can be used to directly obtain a new fp semigroup isomorphic to the first with reduced relations and generators (the mapping is stored as an attribute of the output). With these functions, the user never has to consider precisely what Tietze transformations to perform, and never has to worry about using the StzPresentation object or its associated operations. They can start with an fp semigroup and obtain a simplified fp semigroup.

### 15.8.1 SimplifyFpSemigroup

▷ SimplifyFpSemigroup(*S*) (operation)

**Returns:** A mapping object.

If *S* is a finitely presented semigroup, then SimplifyFpSemigroup will return a mapping object which will map *S* to a finitely presented semigroup which has had its presentation simplified. SimplifyFpSemigroup creates an StzPresentation object *stz* from *S*, which is then reduced using Tietze transformations until the presentation cannot be reduced in length any further.

SimplifyFpSemigroup applies the function StzSimplifyPresentation (15.7.2) to *stz*, which repeatedly checks whether a number of different possible transformations will cause a reduction in length, and if so applies the best one. This loop continues until no transformations cause any reductions, in which case the mapping is returned. The newly reduced FpSemigroup can be accessed either by taking the range of the mapping or calling SimplifiedFpSemigroup, which first runs SimplifyFpSemigroup and then returns the range of the mapping with the mapping held as an attribute.

For more information on how the mapping is created and used, go to 15.6.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup on the generators [ a, b, c ]>
gap> map := SimplifyFpSemigroup(T);;
#I  Applying StzSimplifyPresentation...
#I  StzSimplifyPresentation is verbose by default. Use SetInfoLevel(InfoFpSemigroup, 1) to hide
#I  output while maintaining ability to use StzPrintRelations, StzPrintGenerators, etc.
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 19>
#I  <Removing redundant generator a using relation : 1. a = b^5*c>
#I  Current: <fp semigroup presentation with 2 generators and 1 relation with length 11>
#I  <Creating new generator to replace instances of word: b^3>
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 10>
gap> IsMapping(map);
```

```
true
gap> T.1;
a
gap> T.1 ^ map;
b^5*c
gap> RelationsOfFpSemigroup(Range(map));
[ [ b^3, d ], [ d^2, d ] ]
```

### 15.8.2 SimplifiedFpSemigroup

▷ SimplifiedFpSemigroup(*S*) (operation)

**Returns:** A finitely presented semigroup.

If *S* is an fp semigroup object (IsFpSemigroup (**Reference: IsFpSemigroup**)), then SimplifiedFpSemigroup will return an FpSemigroup object *T* which is isomorphic to *S* which has been reduced to minimise its length. SimplifiedFpSemigroup applies SimplifyFpSemigroup (15.8.1) and assigns the Range of the isomorphism object which is returned to *T*, adding the isomorphism to *T* as an attribute. In this way, while *T* is a completely new FpSemigroup object, words in *S* can be mapped to *T* using the map obtained from the attribute FpTietzeIsomorphism (15.8.4).

For more information on the mapping between the semigroups and how it is created, see 15.6.

Example

```
gap> F := FreeSemigroup("a", "b", "c");;
gap> S := F / [[F.1 ^ 4, F.1], [F.1, F.1 ^ 44], [F.1 ^ 8, F.2 * F.3]];;
gap> T := SimplifiedFpSemigroup(S);;
#I  Applying StzSimplifyPresentation...
#I  StzSimplifyPresentation is verbose by default. Use SetInfoLevel(InfoFpSemigroup, 1) to hide
#I  output while maintaining ability to use StzPrintRelations, StzPrintGenerators, etc.
#I  Current: <fp semigroup presentation with 3 generators and 3 relations with length 63>
#I  <Replacing all instances in other relations of relation: 1. a^4 = a>
#I  Current: <fp semigroup presentation with 3 generators and 3 relations with length 24>
#I  <Replacing all instances in other relations of relation: 1. a^4 = a>
#I  Current: <fp semigroup presentation with 3 generators and 3 relations with length 18>
#I  <Replacing all instances in other relations of relation: 1. a^4 = a>
#I  Current: <fp semigroup presentation with 3 generators and 3 relations with length 15>
#I  <Replacing all instances in other relations of relation: 3. a = a^2>
#I  Current: <fp semigroup presentation with 3 generators and 3 relations with length 12>
#I  <Removing duplicate relation: 1. a = a^2>
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 9>
#I  <Removing redundant generator a using relation : 2. a = b*c>
#I  Current: <fp semigroup presentation with 2 generators and 1 relation with length 8>
gap> map := FpTietzeIsomorphism(T);;
gap> S.1 ^ map;
b*c
gap> S.1 ^ map = T.1 * T.2;
true
gap> invmap := InverseGeneralMapping(map);;
gap> T.1 ^ invmap = S.2;
true
gap> T.1 = S.2;
false
```

### 15.8.3 UnreducedFpSemigroup (for a semigroup)

▷ UnreducedFpSemigroup(*S*) (attribute)

**Returns:** *T*, an fp semigroup object.

If *S* is an fp semigroup object that has been obtained through calling SimplifiedFpSemigroup (15.8.2) on some fp semigroup *T* then UnreducedFpSemigroup returns the original semigroup object before simplification. These are unrelated semigroup objects, except that *S* will have a FpTietzeIsomorphism (15.8.4) attribute that returns an isomorphic mapping from *T* to *S*.

If SimplifyFpSemigroup (15.8.1) has been called on an fp semigroup *T*, then UnreducedFpSemigroup can be used on the Range of the resultant mapping to obtain the domain.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup on the generators [ a, b, c ]>
gap> S := SimplifiedFpSemigroup(T);
#I  Applying StzSimplifyPresentation...
#I  StzSimplifyPresentation is verbose by default. Use SetInfoLevel(InfoFpSemigroup, 1) to hide
#I  output while maintaining ability to use StzPrintRelations, StzPrintGenerators, etc.
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 19>
#I  <Removing redundant generator a using relation : 1. a = b^5*c>
#I  Current: <fp semigroup presentation with 2 generators and 1 relation with length 11>
#I  <Creating new generator to replace instances of word: b^3>
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 10>
<fp semigroup on the generators [ b, c, d ]>
gap> UnreducedFpSemigroup(S) = T;
true
```

### 15.8.4 FpTietzeIsomorphism

▷ FpTietzeIsomorphism(*S*) (attribute)

**Returns:** A mapping object.

If *S* is an fp semigroup object that has been obtained through calling SimplifiedFpSemigroup (15.8.2) on some fp semigroup *T*, then FpTietzeIsomorphism returns an isomorphism from *T* to *S*. Simplification produces an fp semigroup isomorphic to the original fp semigroup, and these two fp semigroup objects can interact with each other through the mapping given by this function.

Example

```
gap> F := FreeSemigroup("a", "b", "c");
<free semigroup on the generators [ a, b, c ]>
gap> T := F / [[F.1, F.2 ^ 5 * F.3],
>              [F.2 ^ 6, F.2 ^ 3]];
<fp semigroup on the generators [ a, b, c ]>
gap> S := SimplifiedFpSemigroup(T);
#I  Applying StzSimplifyPresentation...
#I  StzSimplifyPresentation is verbose by default. Use SetInfoLevel(InfoFpSemigroup, 1) to hide
#I  output while maintaining ability to use StzPrintRelations, StzPrintGenerators, etc.
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 19>
#I  <Removing redundant generator a using relation : 1. a = b^5*c>
#I  Current: <fp semigroup presentation with 2 generators and 1 relation with length 11>
```

```
#I  <Creating new generator to replace instances of word: b^3>
#I  Current: <fp semigroup presentation with 3 generators and 2 relations with length 10>
<fp semigroup on the generators [ b, c, d ]>
gap> T.2;
b
gap> S.1;
b
gap> T.2 = S.1;
false
gap> map := FpTietzeIsomorphism(S);;
gap> T.2 ^ map = S.1;
true
```

# Chapter 16

# Visualising semigroups and elements

There are two operations TikzString (16.3.1) and DotString (16.1.1) in Semigroups for creating LaTeX and dot (also known as GraphViz) format pictures of the Green's class structure of a semigroup and for visualising certain types of elements of a semigroup. There is also a function Splash (**Digraphs: Splash**) for automatically processing the output of TikzString (16.3.1) and DotString (16.1.1) and opening the resulting pdf file.

## 16.1 dot pictures

In this section, we describe the operations in Semigroups for creating pictures in dot format.

The operations described in this section return strings, which can be written to a file using the function FileString (**GAPDoc: FileString**) or viewed using Splash (**Digraphs: Splash**).

### 16.1.1 DotString

▷ DotString(*S[, options]*) (operation)

**Returns:** A string.

If the argument *S* is a semigroup, and the optional second argument *options* is a record, then this operation produces a graphical representation of the partial order of the $\mathscr{D}$-classes of the semigroup *S* together with the eggbox diagram of each $\mathscr{D}$-class. The output is in dot format (also known as GraphViz) format. For details about this file format, and information about how to display or edit this format see https://www.graphviz.org.

The string returned by DotString can be written to a file using the command FileString (**GAPDoc: FileString**).

The $\mathscr{D}$-classes are shown as eggbox diagrams with $\mathscr{L}$-classes as rows and $\mathscr{R}$-classes as columns; group $\mathscr{H}$-classes are shaded gray and contain an asterisk. The $\mathscr{L}$-classes and $\mathscr{R}$-classes within a $\mathscr{D}$-class are arranged to correspond to the normalization of the principal factor given by NormalizedPrincipalFactor (10.4.8). The $\mathscr{D}$-classes are numbered according to their index in GreensDClasses(*S*), so that an i appears next to the eggbox diagram of GreensDClasses(*S*)[i]. A line from one $\mathscr{D}$-class to another indicates that the higher $\mathscr{D}$-class is greater than the lower one in the $\mathscr{D}$-order on *S*.

If the optional second argument *options* is present, it can be used to specify some options for output.

**number**

if *options*.number is false, then the $\mathscr{D}$-classes in the diagram are not numbered according to their index in the list of $\mathscr{D}$-classes of *S*. The default value for this option is true.

**maximal**

if *options*.maximal is true, then the structure description of the group $\mathscr{H}$-classes is displayed; see StructureDescription (**Reference: StructureDescription**). Setting this attribute to true can adversely affect the performance of DotString. The default value for this option is false.

**normal**

if *options*.normal is false, then the $\mathscr{L}$- and $\mathscr{R}$-classes within each $\mathscr{D}$-class are arranged to correspond to PrincipalFactor (10.4.8). If *options*.normal is true, they are instead arranged to correspond to NormalizedPrincipalFactor (10.4.8). Setting this attribute to false may improve the performance of DotString as it avoids the computation of InjectionNormalizedPrincipalFactor (10.4.7). The default value for this option is true.

Example

```
gap> S := FullTransformationMonoid(3);
<full transformation monoid of degree 3>
gap> DotString(S);
"//dot\ndigraph  DClasses {\nnode [shape=plaintext]\nedge [color=blac\
k,arrowhead=none]\n1 [shape=box style=invisible label=<\n<TABLE BORDE\
R=\"0\" CELLBORDER=\"1\" CELLPADDING=\"10\" CELLSPACING=\"0\" PORT=\"\
1\">\n<TR BORDER=\"0\"><TD COLSPAN=\"1\" BORDER = \"0\" > 1</TD></TR>\
<TR><TD BGCOLOR=\"gray\">*</TD></TR>\n</TABLE>>];\n2 [shape=box style\
=invisible label=<\n<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLPADDING=\
\"10\" CELLSPACING=\"0\" PORT=\"2\">\n<TR BORDER=\"0\"><TD COLSPAN=\"\
3\" BORDER = \"0\" > 2</TD></TR><TR><TD BGCOLOR=\"gray\">*</TD><TD BG\
COLOR=\"gray\">*</TD><TD BGCOLOR=\"white\"></TD></TR>\n<TR><TD BGCOLO\
R=\"gray\">*</TD><TD BGCOLOR=\"white\"></TD><TD BGCOLOR=\"gray\">*</T\
D></TR>\n<TR><TD BGCOLOR=\"white\"></TD><TD BGCOLOR=\"gray\">*</TD><T\
D BGCOLOR=\"gray\">*</TD></TR>\n</TABLE>>];\n3 [shape=box style=invis\
ible label=<\n<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLPADDING=\"10\"\
 CELLSPACING=\"0\" PORT=\"3\">\n<TR BORDER=\"0\"><TD COLSPAN=\"1\" BO\
RDER = \"0\" > 3</TD></TR><TR><TD BGCOLOR=\"gray\">*</TD></TR>\n<TR><\
TD BGCOLOR=\"gray\">*</TD></TR>\n<TR><TD BGCOLOR=\"gray\">*</TD></TR>\
\n</TABLE>>];\n1 -> 2\n2 -> 3\n }"
gap> FileString("t3.dot", DotString(S));
1040
```

### 16.1.2 DotString (for a Cayley digraph)

▷ DotString(*digraph*) (operation)

**Returns:** A string.

If *digraph* is a Digraph (**Digraphs: Digraph**) in the category IsCayleyDigraph (**Digraphs: IsCayleyDigraph**), then DotString returns a graphical representation of *digraph*. 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 https://www.graphviz.org.

The string returned by DotString can be written to a file using the command FileString (**GAPDoc: FileString**).

See also DotLeftCayleyDigraph (16.1.4) and TikzLeftCayleyDigraph (16.3.2).

### 16.1.3 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 https://www.graphviz.org.

Example

```
gap> S := DualSymmetricInverseMonoid(4);
<inverse block bijection monoid of degree 4 with 3 generators>
gap> DotSemilatticeOfIdempotents(S);
"//dot\ngraph graphname {\n  node [shape=point]\nranksep=2;\nsubgraph \
cluster_1{\n15 \n}\nsubgraph cluster_2{\n5 11 14 12 13 8 \n}\nsubgraph\
 cluster_3{\n2 10 6 3 4 9 7 \n}\nsubgraph cluster_4{\n1 \n}\n2 -- 1\n3\
 -- 1\n4 -- 1\n5 -- 2\n5 -- 3\n5 -- 4\n6 -- 1\n7 -- 1\n8 -- 2\n8 -- 6\
\n8 -- 7\n9 -- 1\n10 -- 1\n11 -- 2\n11 -- 9\n11 -- 10\n12 -- 3\n12 -- \
6\n12 -- 9\n13 -- 3\n13 -- 7\n13 -- 10\n14 -- 4\n14 -- 6\n14 -- 10\n15\
 -- 5\n15 -- 8\n15 -- 11\n15 -- 12\n15 -- 13\n15 -- 14\n }"
```

### 16.1.4 DotLeftCayleyDigraph

▷ DotLeftCayleyDigraph(*S*) (operation)

▷ DotRightCayleyDigraph(*S*) (operation)

**Returns:** A string.

If *S* is a semigroup satisfying CanUseFroidurePin (6.1.4), then DotLeftCayleyDigraph is simply short for DotString(LeftCayleyDigraph(*S*)).

DotRightCayleyDigraph can be used to produce a dot string for the right Cayley graph of *S*.

See DotString (16.1.1) for more details, and see also TikzLeftCayleyDigraph (16.3.2).

Example

```
gap> DotLeftCayleyDigraph(Semigroup(IdentityTransformation));
"//dot\ndigraph hgn{\nnode [shape=circle]\n1 [label=\"a\"]\n1 -> 1\n}\
\n"
gap> DotRightCayleyDigraph(Semigroup(IdentityTransformation));
"//dot\ndigraph hgn{\nnode [shape=circle]\n1 [label=\"a\"]\n1 -> 1\n}\
\n"
```

## 16.2 tex output

In this section, we describe the operations in Semigroups for creating LaTeX representations of GAP objects. For pictures of GAP objects please see Section 16.3.

### 16.2.1 TexString

▷ `TexString(f[, n])` (operation)

**Returns:** A string.

This function produces a string containing LaTeX code for the transformation `f`. If the optional parameter `n` is used, then this is taken to be the degree of the transformation `f`, if the parameter `n` is not given, then `DegreeOfTransformation` (**Reference: DegreeOfTransformation**) is used by default. If `n` is less than the degree of `f`, then an error is given.

Example

```
gap> TexString(Transformation([6, 2, 4, 3, 6, 4]));
"\\begin{pmatrix}\n  1 & 2 & 3 & 4 & 5 & 6 \\\\\n  6 & 2 & 4 & 3 & 6 &\
 4\n\\end{pmatrix}"
gap> TexString(Transformation([1, 2, 1, 3]), 5);
"\\begin{pmatrix}\n  1 & 2 & 3 & 4 & 5 \\\\\n  1 & 2 & 1 & 3 & 5\n\\en\
d{pmatrix}"
```

## 16.3 `tikz` pictures

In this section, we describe the operations in Semigroups for creating pictures in `tikz` format.

The functions described in this section return a string, which can be written to a file using the function `FileString` (**GAPDoc: FileString**) or viewed using `Splash` (**Digraphs: Splash**).

### 16.3.1 TikzString

▷ `TikzString(obj[, options])` (operation)

**Returns:** A string.

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

Currently the following types of objects are supported:

**blocks**
: If `obj` is the left or right blocks of a bipartition, then `TikzString` returns a graphical representation of these blocks; see Section 3.6.

**bipartitions**
: If `obj` is a bipartition, then `TikzString` returns a graphical representation of `obj`.

  If the optional second argument `options` is a record with the component `colors` set to `true`, then the blocks of `obj` will be colored using the standard `tikz` colors. Due to the limited number of colors available in `tikz` this option only works when the degree of `obj` is less than 20. See Chapter 3 for more details about bipartitions.

**pbrs**
: If `obj` is a `PBR` (4.2.1), then `TikzString` returns a graphical representation of `obj`; see Chapter 4.

**Cayley graphs**
: If `obj` is a `Digraph` (**Digraphs: Digraph**) in the category `IsCayleyDigraph` (**Digraphs: IsCayleyDigraph**), then `TikzString` returns a picture of `obj`. No attempt is made whatsoever to

produce a sensible picture of the digraph *obj*, in fact, the vertices are all given the same coordinates. Human intervention is required to produce a meaningful picture from the value returned by this method. It is intended to make the task of drawing such a Cayley graph more straightforward by providing everything except the final layout of the graph. Please use DotString (16.1.1) if you want an automatically laid out diagram of the digraph *obj*.

Example

```
gap> x := Bipartition([[1, 4, -2, -3], [2, 3, 5, -5], [-1, -4]]);;
gap> TikzString(RightBlocks(x));
"%tikz\n\\documentclass{minimal}\n\\usepackage{tikz}\n\\begin{documen\
t}\n\\begin{tikzpicture}\n  \\draw[ultra thick](5,2)circle(.115);\n  \
\\draw(1.8,5) node [top] {{$1$}};\n  \\fill(4,2)circle(.125);\n  \\dr\
aw(1.8,4) node [top] {{$2$}};\n  \\fill(3,2)circle(.125);\n  \\draw(1\
.8,3) node [top] {{$3$}};\n  \\draw[ultra thick](2,2)circle(.115);\n \
 \\draw(1.8,2) node [top] {{$4$}};\n  \\fill(1,2)circle(.125);\n  \\d\
raw(1.8,1) node [top] {{$5$}};\n\n  \\draw (5,2.125) .. controls (5,2\
.8) and (2,2.8) .. (2,2.125);\n  \\draw (4,2.125) .. controls (4,2.6)\
 and (3,2.6) .. (3,2.125);\n\\end{tikzpicture}\n\n\\end{document}"
gap> x := Bipartition([[1, 5], [2, 4, -3, -5], [3, -1, -2], [-4]]);;
gap> TikzString(x);
"%tikz\n\\documentclass{minimal}\n\\usepackage{tikz}\n\\begin{documen\
t}\n\\begin{tikzpicture}\n\n  %block #1\n  %vertices and labels\n  \\\
fill(1,2)circle(.125);\n  \\draw(0.95, 2.2) node [above] {{ $1$}};\n \
 \\fill(5,2)circle(.125);\n  \\draw(4.95, 2.2) node [above] {{ $5$}};\
\n\n  %lines\n  \\draw(1,1.875) .. controls (1,1.1) and (5,1.1) .. (5\
,1.875);\n\n  %block #2\n  %vertices and labels\n  \\fill(2,2)circle(\
.125);\n  \\draw(1.95, 2.2) node [above] {{ $2$}};\n  \\fill(4,2)circ\
le(.125);\n  \\draw(3.95, 2.2) node [above] {{ $4$}};\n  \\fill(3,0)c\
ircle(.125);\n  \\draw(3, -0.2) node [below] {{ $-3$}};\n  \\fill(5,0\
)circle(.125);\n  \\draw(5, -0.2) node [below] {{ $-5$}};\n\n  %lines\
\n  \\draw(2,1.875) .. controls (2,1.3) and (4,1.3) .. (4,1.875);\n  \
\\draw(3,0.125) .. controls (3,0.7) and (5,0.7) .. (5,0.125);\n  \\dr\
aw(2,2)--(3,0);\n\n  %block #3\n  %vertices and labels\n  \\fill(3,2)\
circle(.125);\n  \\draw(2.95, 2.2) node [above] {{ $3$}};\n  \\fill(1\
,0)circle(.125);\n  \\draw(1, -0.2) node [below] {{ $-1$}};\n  \\fill\
(2,0)circle(.125);\n  \\draw(2, -0.2) node [below] {{ $-2$}};\n\n  %l\
ines\n  \\draw(1,0.125) .. controls (1,0.6) and (2,0.6) .. (2,0.125);\
\n  \\draw(3,2)--(2,0);\n\n  %block #4\n  %vertices and labels\n  \\f\
ill(4,0)circle(.125);\n  \\draw(4, -0.2) node [below] {{ $-4$}};\n\n \
%lines\n\\end{tikzpicture}\n\n\\end{document}"
gap> TikzString(UniversalPBR(2));
"%latex\n\\documentclass{minimal}\n\\usepackage{tikz}\n\\begin{docume\
nt}\n\\usetikzlibrary{arrows}\n\\usetikzlibrary{arrows.meta}\n\\newco\
mmand{\\arc}{\\draw[semithick, -{>[width = 1.5mm, length = 2.5mm]}]}\
\n\\begin{tikzpicture}[\n  vertex/.style={circle, draw, fill=black, i\
nner sep =0.04cm},\n  ghost/.style={circle, draw = none, inner sep = \
0.14cm},\n  botloop/.style={min distance = 8mm, out = -70, in = -110}\
,\n  toploop/.style={min distance = 8mm, out = 70, in = 110}]\n\n  % \
vertices and labels\n  \\foreach \\i in {1,...,2} {\n    \\node [vert\
ex] at (\\i/1.5, 3) {};\n    \\node [ghost] (\\i) at (\\i/1.5, 3) {};\
\n  }\n\n  \\foreach \\i in {1,...,2} {\n    \\node [vertex] at (\\i/\
1.5, 0) {};\n    \\node [ghost] (-\\i) at (\\i/1.5, 0) {};\n  }\n\n  \
% arcs from vertex 1\n  \\arc (1) to (-2);\n  \\arc (1) to (-1);\n  \
```

```
\\arc (1) edge [toploop] (1);\n  \\arc (1) .. controls (1.06666666666\
66667, 2.125) and (0.93333333333333324, 2.125) .. (2);\n\n  % arcs fr\
om vertex -1\n  \\arc (-1) .. controls (1.0666666666666667, 0.875) an\
d (0.93333333333333324, 0.875) .. (-2);\n  \\arc (-1) edge [botloop] \
(-1);\n  \\arc (-1) to (1);\n  \\arc (-1) to (2);\n\n  % arcs from ve\
rtex 2\n  \\arc (2) to (-2);\n  \\arc (2) to (-1);\n  \\arc (2) .. co\
ntrols (0.93333333333333324, 2.125) and (1.0666666666666667, 2.125) .\
. (1);\n  \\arc (2) edge [toploop] (2);\n\n  % arcs from vertex -2\n \
 \\arc (-2) edge [botloop] (-2);\n  \\arc (-2) .. controls (0.9333333\
3333333324, 0.875) and (1.0666666666666667, 0.875) .. (-1);\n  \\arc \
(-2) to (1);\n  \\arc (-2) to (2);\n\n\\end{tikzpicture}\n\\end{docum\
ent}"
```

### 16.3.2 TikzLeftCayleyDigraph

▷ TikzLeftCayleyDigraph(*S*) (operation)

▷ TikzRightCayleyDigraph(*S*) (operation)

**Returns:** A string.

If *S* is a semigroup satisfying CanUseFroidurePin (6.1.4), then TikzLeftCayleyDigraph is simply short for TikzString(LeftCayleyDigraph(*S*)).

TikzRightCayleyDigraph can be used to produce a tikz string for the right Cayley graph of *S*.

See TikzString (16.3.1) for more details, and see also DotLeftCayleyDigraph (16.1.4).

Example

```
gap> TikzLeftCayleyDigraph(Semigroup(IdentityTransformation));
"\\begin{tikzpicture}[scale=1, auto, \n  vertex/.style={c\
ircle, draw, thick, fill=white, minimum size=0.65cm},\n  \
edge/.style={arrows={-angle 90}, thick},\n  loop/.style={\
min distance=5mm,looseness=5,arrows={-angle 90},thick}]\n\
\n  % Vertices . . .\n  \\node [vertex] (a) at (0, 0) {};\
\n  \\node at (0, 0) {$a$};\n\n  % Edges . . .\n  \\path[\
->] (a) edge [loop]\n           node {$a$} (a);\n\\end{ti\
kzpicture}"
gap> TikzRightCayleyDigraph(Semigroup(IdentityTransformation));
"\\begin{tikzpicture}[scale=1, auto, \n  vertex/.style={c\
ircle, draw, thick, fill=white, minimum size=0.65cm},\n  \
edge/.style={arrows={-angle 90}, thick},\n  loop/.style={\
min distance=5mm,looseness=5,arrows={-angle 90},thick}]\n\
\n  % Vertices . . .\n  \\node [vertex] (a) at (0, 0) {};\
\n  \\node at (0, 0) {$a$};\n\n  % Edges . . .\n  \\path[\
->] (a) edge [loop]\n           node {$a$} (a);\n\\end{ti\
kzpicture}"
```

# Chapter 17

# IO

## 17.1 Reading and writing elements to a file

The functions ReadGenerators (17.1.1) and WriteGenerators (17.1.2) can be used to read or write, respectively, elements of a semigroup to a file.

### 17.1.1 ReadGenerators

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

**Returns:** A list of lists of semigroup elements.

If *filename* is an IO package file object or is the name of a file created using WriteGenerators (17.1.2), then ReadGenerators returns the contents of this file as a list of lists of elements of a semigroup.

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

Example

```
gap> file := Concatenation(SEMIGROUPS.PackageDir,
> "/data/tst/testdata");;
gap> ReadGenerators(file, 13);
[ <identity partial perm on [ 2, 3, 4, 5, 6 ]>,
  <identity partial perm on [ 2, 3, 5, 6 ]>, [1,2](5)(6) ]
```

### 17.1.2 WriteGenerators

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

**Returns:** IO_OK or IO_ERROR.

This function provides a method for writing collections of elements of a semigroup to a file. The resulting file can be further compressed using gzip or xz.

The argument *list* should be a list of lists of elements, or semigroups.

The argument *filename* should be a string containing the name of a file or an IO package file object where the entries in *list* will be written; see IO_File (**IO: IO_File mode**) and IO_CompressedFile (**IO: IO_CompressedFile**).

If the optional third argument *append* is not present or is given and equals "w", then the previous content of the file is deleted and overwritten. If the third argument is "a", then list is appended to the file.

If any element of *list* is a semigroup, then the generators of that semigroup are written to *filename*. More specifically, the list returned by GeneratorsOfSemigroup (**Reference: GeneratorsOfSemigroup**) is written to the file.

This function returns IO_OK (**IO: IO_OK**) if everything went well or IO_ERROR (**IO: IO_Error**) if something went wrong.

The file produced by WriteGenerators can be read using ReadGenerators (17.1.1).

From Version 3.0.0 onwards the Semigroups package used the IO package pickling functionality; see (**IO: Pickling and unpickling**) for more details. This approach is used because it is more general and more robust than the methods used by earlier versions of Semigroups , although the performance is somewhat worse, and the resulting files are somewhat larger.

### 17.1.3 IteratorFromGeneratorsFile

▷ IteratorFromGeneratorsFile(*filename*) (function)

**Returns:** An iterator.

If *filename* is a file or a string containing the name of a file created using WriteGenerators (17.1.2), 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.

If you want to get an iterator for a file written using WriteGenerators from a version of Semigroups before version 3.0.0, then you can use IteratorFromOldGeneratorsFile.

## 17.2 Reading and writing multiplication tables to a file

The functions ReadMultiplicationTable (17.2.1) and WriteMultiplicationTable (17.2.2) can be used to read or write, respectively, multiplication tables to a file.

### 17.2.1 ReadMultiplicationTable

▷ ReadMultiplicationTable(*filename[, nr]*) (function)

**Returns:** A list of multiplication tables.

If *filename* is a file or is the name of a file created using WriteMultiplicationTable (17.2.2), then ReadMultiplicationTable returns the contents of this file as a list of multiplication tables.

If the optional second argument *nr* is present, then ReadMultiplicationTable returns the multiplication table stored in the *nr*th line of *filename*.

Example

```
gap> file := Concatenation(SEMIGROUPS.PackageDir,
> "/data/tst/tables.gz");;
gap> tab := ReadMultiplicationTable(file, 12);
[ [ 1, 1, 3, 4, 5, 6, 7, 8, 9, 6 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ],
  [ 3, 3, 1, 5, 4, 7, 6, 9, 8, 7 ], [ 4, 4, 9, 6, 3, 8, 5, 1, 7, 8 ],
  [ 5, 5, 8, 7, 1, 9, 4, 3, 6, 9 ], [ 6, 6, 7, 8, 9, 1, 3, 4, 5, 1 ],
  [ 7, 7, 6, 9, 8, 3, 1, 5, 4, 3 ], [ 8, 8, 5, 1, 7, 4, 9, 6, 3, 4 ],
  [ 9, 9, 4, 3, 6, 5, 8, 7, 1, 5 ], [ 6, 10, 7, 8, 9, 1, 3, 4, 5, 2 ]
 ]
```

### 17.2.2 WriteMultiplicationTable

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

**Returns:** `IO_OK` or `IO_ERROR`.

This function provides a method for writing collections of multiplication tables to a file. The resulting file can be further compressed using `gzip` or `xz`. This function applies to square arrays with a maximum of 255 rows where the entries are integers from `[1, 2, .., n]` (where `n` is the number of rows in the array).

The argument *`list`* should be a list of multiplication tables.

The argument *`filename`* should be a file or a string containing the name of a file where the entries in *`list`* will be written or an IO package file object; see `IO_File` (**IO: IO_File mode**) and `IO_CompressedFile` (**IO: IO_CompressedFile**).

If the optional third argument *`append`* is not present or is given and equals `"w"`, then the previous content of the file is deleted and overwritten. If the third argument is given and equals `"a"` then `list` is appended to the file. This function returns `IO_OK` (**IO: IO_OK**) if everything went well or `IO_ERROR` (**IO: IO_Error**) if something went wrong.

The multiplication tables saved in *`filename`* can be recovered from the file using `ReadMultiplicationTable` (17.2.1).

### 17.2.3 IteratorFromMultiplicationTableFile

▷ `IteratorFromMultiplicationTableFile(`*`filename`*`)` (function)

**Returns:** An iterator.

If *`filename`* is a file or a string containing the name of a file created using `WriteMultiplicationTable` (17.2.2), then `IteratorFromMultiplicationTableFile` returns an iterator `iter` such that `NextIterator(iter)` returns the next multiplication table stored in the file *`filename`*.

This function is a convenient way of, for example, looping over a collection of multiplication tables 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 18

# Translations

In this chapter we describe the functionality in Semigroups for working with translations of semigroups. The notation used (as well as a number of results) is based on [Pet70]. Translations are of interest mainly due to their role in ideal extensions. A description of this role can also be found in [Pet70]. The implementation of translations in this package is only applicable to finite semigroups satisfying `CanUseFroidurePin` (6.1.4).

For a semigroup $S$, a transformation $\lambda$ of $S$ (written on the left) is a left translation if for all $s,t$ in $S$, $\lambda(s)t = \lambda(st)$. A right translation $\rho$ (written on the right) is defined dually.

The set $L$ of left translations of $S$ is a semigroup under composition of functions, as is the set $R$ of right translations. The associativity of $S$ guarantees that left [right] multiplication by any element $s$ of $S$ represents a left [right] translation; this is the *inner* left [right] translation $\lambda_s$ [$\rho_s$]. The inner left [right] translations form an ideal in $L$ [$R$].

A left translation $\lambda$ and right translation $\rho$ are *linked* if for all `s, t` in $S$, $s\lambda(t) = (s)\rho t$. A pair of linked translations is called a *bitranslation*. The set of all bitranslations forms a semigroup $H$ called the *translational hull* of $S$ where the operation is componentwise. If the components are inner translations corresponding to multiplication by the same element, then the bitranslation is *inner*. The inner bitranslations form an ideal of the translational hull.

Translations of a normalized Rees matrix semigroup $T$ (see `RMSNormalization` (6.5.7)) over a group $G$ can be represented through certain tuples, which can be computed very efficiently compared to arbitrary translations. For left translations these tuples consist of a function from the row indices of $T$ to G and a transformation on the row indices of $T$; the same is true of right translations and columns. More specifically, given a normalised Rees matrix semigroup $S$ over a group $G$ with sandwich matrix $P$, rows $I$ and columns $J$, the left translations are characterised by pairs $(\theta,\chi)$ where $\theta$ is a function from $I$ to $G$ and $\chi$ is a transformation of $I$. The left translation $\lambda$ defined by $(\theta,\chi)$ acts on $S$ via

$$\lambda((i,a,\mu)) = ((i)\chi,(i)\theta\cdot a,\mu),$$

where $i \in I$, $a \in G$, and $\mu \in J$ Dually, right translations $\rho$ are characterised by pairs $(\omega,\psi)$ where $\omega$ is a function from $J$ to $G$ and $\psi$ is a transformation of $J$, with action given by

$$((i,a,\mu))\rho = (i,a\cdot(\mu)\omega,(\mu)\psi).$$

Similarly, bitranslations $(\lambda,\rho)$ of $S$ can be characterised by triples $(g,\chi,\psi)$ such that $g \in G$, $\chi$ and $\psi$ are transformations of $I,J$ respectively, and

$$p_{\mu,(i)\chi}\cdot g\cdot p_{(1)\psi,i} = p_{\mu,(1)\chi}\cdot g\cdot p_{(\mu)\psi,i}.$$

The action of $\lambda$ on $S$ is then given by

$$\lambda((i,a,\mu)) = ((i)\chi, g \cdot p_{(1)\psi,i} \cdot a, \mu),$$

and of $\rho$ on $S$ by

$$((i,a,\mu))\rho = (i, a \cdot p_{\mu,(1)\chi} \cdot g, (\mu)\psi).$$

Further details may be found in [CP77].

## 18.1 Methods for translations

### 18.1.1 IsXTranslation

▷ IsSemigroupTranslation(*arg*) (Category)

**Returns:** true or false

All, and only, left [right] translations belong to IsLeftTranslation [IsRightTranslation]. These are both subcategories of IsSemigroupTranslation, which itself is a subcategory of IsAssociativeElement.

Example

```
gap> S := RectangularBand(3, 4);;
gap> l := Representative(LeftTranslations(S));;
gap> IsSemigroupTranslation(l);
true
gap> IsLeftTranslation(l);
true
gap> IsRightTranslation(l);
false
gap> l = One(LeftTranslations(S));
true
gap> l * l = l;
true
```

### 18.1.2 IsBitranslation (for IsAssociativeElement and IsMultiplicativeElementWithOne)

▷ IsBitranslation(*arg*) (Category)

**Returns:** true or false

All, and only, bitranslations belong to IsBitranslation. This is a subcategory of IsAssociativeElement (**Reference: IsAssociativeElement**).

Example

```
gap> G := Group((1, 2), (3, 4));;
gap> A := AsList(G);;
gap> mat := [[A[1], 0, A[1]],
> [A[2], A[2], A[4]]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> L := LeftTranslations(S);;
gap> R := RightTranslations(S);;
gap> l := OneOp(Representative(L));;
gap> r := OneOp(Representative(R));;
```

```
gap> H := TranslationalHull(S);;
gap> x := Bitranslation(H, l, r);;
gap> IsBitranslation(x);
true
gap> IsSemigroupTranslation(x);
false
```

### 18.1.3 IsXTranslationCollection

▷ IsSemigroupTranslationCollection(*obj*) (Collection)

**Returns:** true or false

Every collection of left-, right-, or bi-translations belongs to the respective category IsXTranslationCollection (18.1.3).

### 18.1.4 XPartOfBitranslation

▷ LeftPartOfBitranslation(*h*) (function)

**Returns:** a left or right translation

For a Bitranslation *h* consisting of a linked pair $(l,r)$, of left and right translations, LeftPartOfBitranslation(<A>h</A>) returns the left translation l, and RightPartOfBitranslation(<A>h</A>) returns the right translation r.

### 18.1.5 XTranslation

▷ LeftTranslation(*T, x[, y]*) (operation)

**Returns:** a left or right translation

For the semigroup *T* of left or right translations of a semigroup *S* and *x* is one of:

- a mapping on the underlying semigroup; note that in this case only the values of the mapping on the UnderlyingRepresentatives of *T* are checked and used, so mappings which do not define translations can be used to create translations if they are valid on that subset of *S*;
- a list of indices representing the images of the UnderlyingRepresentatives of *T*, where the ordering is that of PositionCanonical (11.1.2) on *S*;
- (for LeftTranslation) a list of length Length(Rows(S)) containing elements of UnderlyingSemigroup(S); in this case *S* must be a normalised Rees matrix semigroup and y must be a Transformation of Rows(S);
- (for RightTranslation) a list of length Length(Columns(S)) containing elements of UnderlyingSemigroup(S); in this case *S* must be a normalised Rees matrix semigroup and y must be a Transformation of Columns(S);

LeftTranslation and RightTranslation return the corresponding translations.

Example

```
gap> S := RectangularBand(3, 4);;
gap> L := LeftTranslations(S);;
gap> s := AsList(S)[1];;
gap> map := MappingByFunction(S, S, x -> s * x);;
gap> l := LeftTranslation(L, map);
```

```
<left translation on <regular transformation semigroup of size 12,
 degree 8 with 4 generators>>
gap> s ^ l;
Transformation( [ 1, 2, 1, 1, 5, 5, 5, 5 ] )
```

### 18.1.6 Bitranslation (for IsBitranslationsSemigroup, IsLeftTranslation, IsRightTranslation)

▷ `Bitranslation(H, l, r)` (operation)

**Returns:** a bitranslation

If $H$ is a translational hull over a semigroup $S$, and $l$ and $r$ are linked left and right translations respectively over $S$, then this function returns the bitranslation $(l, r)$. If $l$ and $r$ are not linked, then an error is produced.

Example

```
gap> G := Group((1, 2), (3, 4));;
gap> A := AsList(G);;
gap> mat := [[A[1], 0],
> [A[2], A[2]]];;
gap> S := ReesZeroMatrixSemigroup(G, mat);;
gap> L := LeftTranslations(S);;
gap> R := RightTranslations(S);;
gap> l := LeftTranslation(L, MappingByFunction(S, S, s -> S.1 * s));;
gap> r := RightTranslation(R, MappingByFunction(S, S, s -> s * S.1));;
gap> H := TranslationalHull(S);;
gap> x := Bitranslation(H, l, r);
<bitranslation on <regular semigroup of size 17, with 4 generators>>
```

### 18.1.7 UnderlyingSemigroup

▷ `UnderlyingSemigroup(S)` (attribute)

**Returns:** a semigroup

Given a semigroup of translations or bitranslations, returns the semigroup on which these translations act.

### 18.1.8 XTranslationsSemigroupOfFamily

▷ `LeftTranslationsSemigroupOfFamily(fam)` (attribute)

**Returns:** the semigroup of left or right translations, or the translational hull

Given a family *fam* of left-, right- or bi-translations, returns the translations semigroup or translational hull to which they belong.

Example

```
gap> S := RectangularBand(3, 3);;
gap> L := LeftTranslations(S);;
gap> l := Representative(L);;
gap> LeftTranslationsSemigroupOfFamily(FamilyObj(l)) = L;
true
gap> H := TranslationalHull(S);;
gap> h := Representative(H);;
gap> TranslationalHullOfFamily(FamilyObj(h)) = H;
true
```

### 18.1.9 TypeXTranslationSemigroupElements

▷ TypeLeftTranslationsSemigroupElements(*arg*) (attribute)

**Returns:** a type

Given a (bi)translations semigroup, returns the type of the elements that it contains.

### 18.1.10 XTranslations

▷ LeftTranslations(*S*) (attribute)

**Returns:** the semigroup of left or right translations

Given a finite semigroup *S* satisfying CanUseFroidurePin (6.1.4), returns the semigroup of all left or right translations of *S*.

Example

```
gap> S := Semigroup([Transformation([1, 4, 3, 3, 6, 5]),
> Transformation([3, 4, 1, 1, 4, 2])]);;
gap> L := LeftTranslations(S);
<the semigroup of left translations of <transformation semigroup of
 degree 6 with 2 generators>>
gap> Size(L);
361
```

### 18.1.11 TranslationalHull (for IsSemigroup and CanUseFroidurePin and IsFinite)

▷ TranslationalHull(*S*) (attribute)

**Returns:** the translational hull

Given a finite semigroup *S* satisfying CanUseFroidurePin (6.1.4), returns the translational hull of *S*.

Example

```
gap> S := Semigroup([Transformation([1, 4, 3, 3, 6, 5]),
> Transformation([3, 4, 1, 1, 4, 2])]);;
gap> H := TranslationalHull(S);
<translational hull over <transformation semigroup of degree 6 with 2
 generators>>
gap> Size(H);
38
```

### 18.1.12 NrXTranslations

▷ NrLeftTranslations(*S*) (attribute)

**Returns:** the number of left-, right-, or bi-translations

Given a finite semigroup *S* satisfying CanUseFroidurePin (6.1.4), returns the number of left-, right-, or bi-translations of *S*. This is typically more efficient than calling Size(LeftTranslations(<A>S</A>)), as the [bi]translations may not be produced.

Example

```
gap> S := Semigroup([Transformation([1, 4, 3, 3, 6, 5, 2]),
> Transformation([3, 4, 1, 1, 4, 2])]);;
gap> NrLeftTranslations(S);
1444
gap> NrRightTranslations(S);
398
```

```
gap> NrBitranslations(S);
69
```

### 18.1.13 InnerXTranslations

▷ `InnerLeftTranslations(`*S*`)` (attribute)

**Returns:** the monoid of inner left or right translations

For a finite semigroup *S* satisfying `CanUseFroidurePin` (6.1.4), `InnerLeftTranslations(`*S*`)` returns the inner left translations of S (i.e. the translations defined by left multiplication by a fixed element of *S*), and `InnerRightTranslations(`*S*`)` returns the inner right translations of *S* (i.e. the translations defined by right multiplication by a fixed element of *S*).

Example

```
gap> S := Semigroup([Transformation([1, 4, 3, 3, 6, 5]),
> Transformation([3, 4, 1, 1, 4, 2])]);;
gap> I := InnerLeftTranslations(S);
<left translations semigroup over <transformation semigroup
 of size 22, degree 6 with 2 generators>>
gap> Size(I) <= Size(S);
true
```

### 18.1.14 InnerTranslationalHull (for IsSemigroup and CanUseFroidurePin and IsFinite)

▷ `InnerTranslationalHull(`*S*`)` (attribute)

**Returns:** the inner translational hull

Given a finite semigroup *S* satisfying `CanUseFroidurePin` (6.1.4), returns the inner translational hull of *S*, i.e. the bitranslations whose left and right translation components are inner translations defined by left and right multiplication by the same fixed element of *S*.

Example

```
gap> S := Semigroup([Transformation([1, 4, 3, 3, 6, 5]),
> Transformation([3, 4, 1, 1, 4, 2])]);;
gap> I := InnerTranslationalHull(S);
<semigroup of bitranslations over <transformation semigroup
 of size 22, degree 6 with 2 generators>>
gap> L := LeftTranslations(S);;
gap> R := RightTranslations(S);;
gap> H := TranslationalHull(S);;
gap> inners := [];;
gap> for s in S do
> l := LeftTranslation(L, MappingByFunction(S, S, x -> s * x));
> r := RightTranslation(R, MappingByFunction(S, S, x -> x * s));
> AddSet(inners, Bitranslation(H, l, r));
> od;
gap> Set(I) = inners;
true
```

### 18.1.15 UnderlyingRepresentatives (for IsTranslationsSemigroup)

▷ `UnderlyingRepresentatives(`*T*`)` (attribute)

**Returns:** a set of representatives

For efficiency, we typically store translations on a semigroup $S$ as their actions on a small subset of $S$. For left translations, this is a set of representatives of the maximal $\mathscr{R}$-classes of $S$; dually for right translations we use representatives of the maximal $\mathscr{L}$-classes. You can use `UnderlyingRepresentatives` to access these representatives.

Example

```
gap> G := Range(IsomorphismPermGroup(SmallGroup(12, 1)));;
gap> mat := [[G.1, G.2], [G.1, G.1],
> [G.2, G.3], [G.1 * G.2, G.1 * G.3]];;
gap> S := ReesMatrixSemigroup(G, mat);;
gap> L := LeftTranslations(S);;
gap> R := RightTranslations(S);;
gap> UnderlyingRepresentatives(L);
[ (1,(),1), (2,(),2) ]
gap> UnderlyingRepresentatives(R);
[ (1,(),1), (2,(),2), (1,(),3), (1,(),4) ]
```

### 18.1.16 ImageSetOfTranslation (for IsSemigroupTranslation)

▷ `ImageSetOfTranslation(t)` (operation)

**Returns:** a set of elements

Given a left or right translation `t` on a semigroup $S$, returns the set of elements of $S$ lying in the image of `t`.

Example

```
gap> S := Semigroup([Transformation([1, 3, 3, 4]),
> Transformation([3, 4, 1, 2])]);;
gap> t := Set(LeftTranslations(S))[4];
<left translation on <transformation semigroup of size 8, degree 4
 with 2 generators>>
gap> ImageSetOfTranslation(t);
[ Transformation( [ 1, 2, 3, 1 ] ), Transformation( [ 1, 3, 3, 1 ] ),
  Transformation( [ 3, 1, 1, 3 ] ), Transformation( [ 3, 4, 1, 3 ] ) ]
```

# References


[ABMS15] J. Araújo, W. Bentz, J. D. Mitchell, and C. Schneider. The rank of the semigroup of transformations stabilising a partition of a finite set. *Math. Proc. Camb. Phil. Soc.*, 159:339 - 353, 2015. 96

[AMM23] M. Anagnostopoulou-Merkouri, Z. Mesyan, and J. D. Mitchell. Properties of congruence lattices of graph inverse semigroups. *International Journal of Algebra and Computation*, 2023. 262, 263

[Aui12] K. Auinger. Krohn–Rhodes complexity of Brauer type semigroups. *Portugaliae Mathematica*, 69(4):341–360, 2012. 15

[Bac92] Q. J. P. Baccelli F. Cohen G. Olsder G. J. *Synchronisation and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992. 72, 74

[BDF15] A. E. Brouwer, J. Draisma, and B. J. Frenk. Lossy gossip and composition of metrics. *Discrete & Computational Geometry*, 53(4):890–913, 2015. 110

[Bur16] S. A. Burrell. The Order Problem for Natural and Tropical Matrix Semigroups. MMath project, University of St Andrews, United Kingdom, 2016. 74

[Cha25] B. Charles. Computing character tables and cartan matrices of finite monoids with fixed point counting. *Journal of Symbolic Computation*, 127:1--29, 2025. 206

[CP77] A. H. Clifford and M. Petrich. Some classes of completely regular semigroups. *Journal of Algebra*, 46(2):462--480, 1977. 312

[DMW18] C. Donoven, J. D. Mitchell, and W. A. Wilson. Computing maximal subsemigroups of a finite semigroup. *Journal of Algebra*, 505:559-596, July 2018. 188, 190

[Eas19] J. East. Presentations for rook partition monoids and algebras and their singular ideals. *J. Pure and Applied Algebra*, 223:1097-1122, March 2019. 101

[EEMP19] J. East, A. Egri-Nagy, J. D. Mitchell, and Y. Péresse. Computing finite semigroups. *J. Symbolic Computation*, 92:110 - 155, 2019. 75, 76, 161

[Far09] K. G. Farlow. Max-Plus Algebra. Master's thesis, Virginia Polytechnic Institute and State University, United States, 2009. 71, 74

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

[FP97] V. Froidure and J.-E. Pin. Algorithms for computing finite semigroups. In *Foundations of computational mathematics (Rio de Janeiro, 1997)*, page 112–126. Springer, Berlin, 1997. 49, 76, 77

[Gau96] S. Gaubert. On the Burnside problem for Semigroups of Matrices over the (max, +) Algebra. *Semigroup Forum*, 5:271-292, 1996. 73, 74

[GGR68] N. Graham, R. Graham, and J. Rhodes. Maximal subsemigroups of finite semigroups. *J. Combinatorial Theory*, 4:203–209, 1968. 188, 190

[Gra68] R. Graham. On finite 0-simple semigroups and graph theory. *Mathematical systems theory*, 2(4):325–339, 1968. 89

[Gro06] C. Grood. The rook partition algebra. *J. Combin. Theory Ser. A*, 113(2):325–351, 2006. 101

[How95] J. 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. 118, 126, 133, 136, 215, 229, 254, 255, 258, 259, 260, 261, 278, 279

[HR05] T. Halverson and A. Ram. Partition algebras. *European Journal of Combinatorics*, 26(6):869–921, 2005. 15, 101

[JK07] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, page 135–149. SIAM, 2007. 64

[KM11] G. Kudryavtseva and V. Maltcev. Two generalisations of the symmetric inverse semigroups. *Publ. Math. Debrecen*, 78(2):253–282, 2011. 104

[KMU15] G. Kudryavtseva, V. Maltcev, and A. Umar. Presentation for the partial dual symmetric inverse monoid. *Comm. Algebra*, 43(4):1621–1639, 2015. 104

[MM13] P. Martin and V. Mazorchuk. Partitioned binary relations. *Mathematica Scandinavica*, 113:30-52, 2013. 37, 76

[MM16] Z. Mesyan and J. D. Mitchell. The structure of a graph inverse semigroup. *Semigroup Forum*, 93(1):111–130, March 2016. 122

[Pet70] M. Petrich. The translational hull in semigroups and rings. *Semigroup Forum*, 1(1):283--360, 1970. 311

[RR10] J. Radoszewski and W. Rytter. Efficient testing of equivalence of words in a free idempotent semigroup. *SOFSEM 2010: Theory and Practice of Computer Science*, page 663--671, 2010. 121

[Sch92] B. M. Schein. The minimal degree of a finite inverse semigroup. *Trans. Amer. Math. Soc.*, 333(2):877–888, 1992. 202

[Sim78] I. Simon. Limited subsets of a free monoid. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, page 143–150, Washington, DC, USA, 1978. IEEE Computer Society. 73

[Wan19] Z.-P. Wang. Congruences on graph inverse semigroups. *Journal of Algebra*, 534:51--64, sep 2019. 262, 263

# Index


* (for PBRs), 42
* (for Rees (0-)matrix semigroup isomorphisms by triples), 280
* (for bipartitions), 23
* (for matrices over a semiring), 58
< (for PBRs), 42
< (for Rees (0-)matrix semigroup isomorphisms by triples), 280
< (for bipartitions), 23
< (for matrices over a semiring), 58
= (for Rees (0-)matrix semigroup isomorphisms by triples), 280
= (for PBRs), 42
= (for bipartitions), 23
= (for matrices over a semiring), 58
\in, 62
\<
    for Green's classes, 157
^ (for Rees (0-)matrix semigroup isomorphisms by triples), 280

AnnularJonesMonoid, 102
AntiIsomorphismDualFpMonoid, 91
AntiIsomorphismDualFpSemigroup, 91
AntiIsomorphismDualSemigroup, 132
ApsisMonoid, 105
AsBipartition, 19
AsBlockBijection, 21
AsBooleanMat, 59
AsCongruenceByWangPair, 261
AsInverseSemigroupCongruenceByKernelTrace, 258
AsList, 56
AsListCanonical, 166
AsMatrix
    for a filter and a matrix, 52
    for a filter, matrix, and threshold, 52
    for a filter, matrix, threshold, and period, 52
AsMonoid, 87
AsMutableList, 56
AsPartialPerm
    for a bipartition, 22
    for a PBR, 41
AsPBR, 39
AsPermutation
    for a bipartition, 23
    for a PBR, 41
AsSemigroup, 86
AsSemigroupCongruenceByGeneratingPairs, 256
AsSemigroupHomomorphismByFunction
    for a semigroup homomorphism by images, 269
AsSemigroupHomomorphismByImages
    for a semigroup homomorphism by function, 268
AsSemigroupIsomorphismByFunction
    for a semigroup homomorphism by images, 275
AsTransformation
    for a bipartition, 22
    for a PBR, 41
AutomorphismGroup
    for a semigroup, 273

Bipartition, 16
BipartitionByIntRep, 16
Bitranslation
    for IsBitranslationsSemigroup, IsLeftTranslation, IsRightTranslation, 313
BlistNumber, 63
BlockDiagonalMatrixOfCharacterTables
    for a semigroup, 207
BLOCKS_NC, 32
BooleanMat, 58
BooleanMatNumber, 63
BrandtSemigroup, 116
BrauerMonoid, 100

`CanonicalBlocks`, 31
`CanonicalBooleanMat`, 64
  for a perm group and boolean matrix, 64
  for a perm group, perm group and boolean matrix, 64
`CanonicalForm`
  for a free inverse semigroup element, 127
`CanonicalMultiplicationTable`, 271
`CanonicalMultiplicationTablePerm`, 271
`CanonicalReesMatrixSemigroup`, 280
`CanonicalReesZeroMatrixSemigroup`, 280
`CanonicalTransformation`, 193
`CanUseFroidurePin`, 77
`CanUseGapFroidurePin`, 77
`CanUseLibsemigroupsFroidurePin`, 77
`CatalanMonoid`, 94
`CayleyDigraphOfCongruences`
  for a semigroup, 245
  for a semigroup and a list or collection, 245
`CayleyDigraphOfLeftCongruences`
  for a semigroup, 245
  for a semigroup and a list or collection, 245
`CayleyDigraphOfRightCongruences`
  for a semigroup, 245
  for a semigroup and a list or collection, 245
`CharacterTableOfInverseSemigroup`, 202
`ClosureInverseMonoid`, 80
`ClosureInverseSemigroup`, 80
`ClosureMonoid`, 80
`ClosureSemigroup`, 80
`CodomainOfBipartition`, 28
`ComponentRepsOfPartialPermSemigroup`, 194
`ComponentRepsOfTransformationSemigroup`, 190
`ComponentsOfPartialPermSemigroup`, 195
`ComponentsOfTransformationSemigroup`, 190
`CompositionMapping2`
  for IsRMSIsoByTriple, 278
  for IsRZMSIsoByTriple, 278
`CongruenceByWangPair`, 261
`CongruencesOfPoset`, 247
`CongruencesOfSemigroup`
  for a semigroup, 241
  for a semigroup and a multiplicative element collection, 241
`ContentOfFreeBandElement`, 119
`ContentOfFreeBandElementCollection`, 119
`CrossedApsisMonoid`, 105
`CyclesOfPartialPerm`, 195
`CyclesOfPartialPermSemigroup`, 195
`CyclesOfTransformationSemigroup`, 191

`DClass`, 145
`DClassBicharacter`
  for a D-class, 206
`DClasses`, 147
`DClassNC`, 146
`DClassOfHClass`, 144
`DClassOfLClass`, 144
`DClassOfRClass`, 144
`DClassReps`, 149
`DegreeOfBipartition`, 25
`DegreeOfBipartitionCollection`, 25
`DegreeOfBipartitionSemigroup`, 36
`DegreeOfBlocks`, 33
`DegreeOfPBR`, 42
`DegreeOfPBRCollection`, 42
`DegreeOfPBRSemigroup`, 48
`DigraphOfAction`
  for a transformation semigroup, list, and action, 191
`DigraphOfActionOnPoints`
  for a transformation semigroup, 192
  for a transformation semigroup and an integer, 192
`DimensionOfMatrixOverSemiring`, 50
`DimensionOfMatrixOverSemiringCollection`, 51
`DirectProduct`, 129
`DirectProductOp`, 129
`DomainOfBipartition`, 28
`DotLeftCayleyDigraph`, 303
`DotRightCayleyDigraph`, 303
`DotSemilatticeOfIdempotents`, 303
`DotString`, 301
  for a Cayley digraph, 302
`DualSemigroup`, 130
`DualSymmetricInverseMonoid`, 102
`DualSymmetricInverseSemigroup`, 102

`ElementOfFpMonoid`, 284

ElementOfFpSemigroup, 283
ELM_LIST (for Rees (0-)matrix semigroup isomorphisms by triples), 280
ELM_LIST
  for IsRMSIsoByTriple, 278
EmbeddingFpMonoid, 91
EmptyPBR, 38
EndomorphismMonoid
  for a digraph, 97
  for a digraph and vertex coloring, 97
EndomorphismsPartition, 94
Enumerate, 167
EnumeratorCanonical, 166
EqualInFreeBand, 119
EquivalenceRelationCanonicalLookup
  for an equivalence relation over a finite semigroup, 239
EquivalenceRelationCanonicalPartition, 240
EquivalenceRelationLookup
  for an equivalence relation over a finite semigroup, 238
EUnitaryInverseCover, 203
EvaluateWord, 171
ExtRepOfObj
  for a bipartition, 25
  for a blocks, 32
  for a PBR, 43

FactorisableDualSymmetricInverseMonoid, 103
Factorization, 172
FixedPointsOfTransformationSemigroup
  for a transformation semigroup, 192
FpTietzeIsomorphism, 299
FreeBand
  for a given rank, 117
  for a list of distinct names, 117
  for various distinct names, 117
FreeInverseSemigroup
  for a given rank, 126
  for a list of names, 126
  for various names, 126
FreeMonoidAndAssignGeneratorVars, 284
FreeSemigroupAndAssignGeneratorVars, 284
FreeSemilattice, 113
FullBooleanMatMonoid, 108
FullMatrixMonoid, 107
FullPBRMonoid, 106
FullTropicalMaxPlusMonoid, 110
FullTropicalMinPlusMonoid, 111

GeneralizedConjugacyClasses
  for a semigroup, 205
GeneralizedConjugacyClassesRepresentatives
  for a semigroup, 205
GeneralLinearMonoid, 107
GeneratingCongruencesOfJoinSemilattice, 249
GeneratingCongruencesOfLattice, 262
Generators, 174
GeneratorsOfSemigroupIdeal, 141
GeneratorsOfStzPresentation, 286
GeneratorsSmallest
  for a semigroup, 179
GLM, 107
GossipMonoid, 109
GraphInverseSemigroup, 121
GraphOfGraphInverseSemigroup, 123
GreensDClasses, 147
GreensDClassOfElement, 145
  for a free band and element, 120
GreensDClassOfElementNC, 146
GreensHClasses, 147
GreensHClassOfElement, 145
  for a Rees matrix semigroup, 145
GreensHClassOfElementNC, 146
GreensJClasses, 147
GreensLClasses, 147
GreensLClassOfElement, 145
GreensLClassOfElementNC, 146
GreensRClasses, 147
GreensRClassOfElement, 145
GreensRClassOfElementNC, 146
GroupHClass, 158
GroupOfUnits, 183

HallMonoid, 109
HClass, 145
  for a Rees matrix semigroup, 145
HClasses, 147
HClassNC, 146

HClassReps, 149
HomomorphismsOfStrongSemilatticeOf-
Semigroups, 135

Ideals
for a semigroup, 141
IdempotentGeneratedSubsemigroup, 186
Idempotents, 184
IdentityBipartition, 16
IdentityPBR, 39
ImagesElm
for IsRMSIsoByTriple, 279
ImageSetOfTranslation
for IsSemigroupTranslation, 316
ImagesRepresentative
for IsRMSIsoByTriple, 279
IndecomposableElements, 180
IndexOfVertexOfGraphInverseSemigroup,
124
IndexPeriodOfSemigroupElement, 169
InfoSemigroups, 13
InjectionNormalizedPrincipalFactor, 162
InjectionPrincipalFactor, 162
InnerLeftTranslations
for IsSemigroup and CanUseFroidurePin and
IsFinite, 315
InnerTranslationalHull
for IsSemigroup and CanUseFroidurePin and
IsFinite, 315
IntRepOfBipartition, 26
InverseMonoidByGenerators, 78
InverseOp, 71
for an integer matrix, 70
InverseSemigroupByGenerators, 78
InverseSemigroupCongruenceByKernel-
Trace, 258
InverseSubsemigroupByProperty, 82
Irr
for a monoid character table, 208
IrreducibleComponentsOfBipartition, 27
IrredundantGeneratingSubset, 177
IsActingSemigroup, 76
IsAntiSymmetricBooleanMat, 67
IsAperiodicSemigroup, 219
IsBand, 211
IsBipartition, 15
IsBipartitionCollColl, 15
IsBipartitionCollection, 15
IsBipartitionMonoid, 35
IsBipartitionPBR, 44
IsBipartitionSemigroup, 35
IsBitranslation
for IsAssociativeElement and IsMultiplica-
tiveElementWithOne, 311
IsBlockBijection, 30
IsBlockBijectionMonoid, 35
IsBlockBijectionPBR, 44
IsBlockBijectionSemigroup, 35
IsBlockGroup, 212
IsBlocks, 32
IsBooleanMat, 55
IsBooleanMatCollColl, 55
IsBooleanMatCollection, 55
IsBooleanMatMonoid, 73
IsBooleanMatSemigroup, 73
IsBrandtSemigroup, 227
IsCayleyDigraphOfCongruences, 243
IsCliffordSemigroup, 227
IsColTrimBooleanMat, 65
IsCombinatorialSemigroup, 219
IsCommutativeSemigroup, 212
IsCompletelyRegularSemigroup, 213
IsCompletelySimpleSemigroup, 223
IsCongruenceByWangPair, 260
IsCongruenceClass, 237
IsCongruenceFreeSemigroup, 213
IsCongruencePoset, 243
IsConnectedTransformationSemigroup
for a transformation semigroup, 194
IsCryptoGroup, 214
IsDTrivial, 219
IsDualSemigroupElement, 131
IsDualSemigroupRep, 131
IsDualTransBipartition, 29
IsDualTransformationPBR, 45
IsEmptyPBR, 43
IsEnumerated, 168
IsEquivalenceBooleanMat, 68
IsEUnitaryInverseSemigroup, 228
IsFactorisableInverseMonoid, 230
IsFinite, 73
IsFInverseMonoid, 229
IsFInverseSemigroup, 229

IsFreeBand
for a given semigroup, 118
IsFreeBandCategory, 118
IsFreeBandElement, 118
IsFreeBandElementCollection, 118
IsFreeBandSubsemigroup, 118
IsFreeInverseSemigroup, 126
IsFreeInverseSemigroupCategory, 126
IsFreeInverseSemigroupElement, 127
IsFreeInverseSemigroupElement-
Collection, 127
IsFullMatrixMonoid, 107
IsGeneralLinearMonoid, 107
IsGraphInverseSemigroup, 122
IsGraphInverseSemigroupElement, 122
IsGraphInverseSemigroupElement-
Collection, 123
IsGraphInverseSubsemigroup, 123
IsGreensClassNC, 158
IsGreensDGreaterThanFunc, 154
IsGroupAsSemigroup, 215
IsHTrivial, 219
IsIdempotentGenerated, 215
IsIdentityPBR, 44
IsIntegerMatrixMonoid, 73
IsIntegerMatrixSemigroup, 73
IsInverseSemigroupCongruenceByKernel-
Trace, 257
IsInverseSemigroupCongruenceClassBy-
KernelTrace, 260
IsIrreducibleBipartition, 31
IsIsomorphicSemigroup, 270
IsJoinIrreducible, 230
IsLeftCongruenceClass, 237
IsLeftSemigroupCongruence, 234
IsLeftSimple, 215
IsLeftZeroSemigroup, 216
IsLinkedTriple, 256
IsLTrivial, 219
IsMajorantlyClosed, 231
IsMatrixOverFiniteFieldCollColl, 55
IsMatrixOverFiniteFieldCollection, 55
IsMatrixOverFiniteFieldMonoid, 73
IsMatrixOverFiniteFieldSemigroup, 73
IsMatrixOverSemiring, 50
IsMatrixOverSemiringCollColl, 50
IsMatrixOverSemiringCollection, 50
IsMatrixOverSemiringMonoid, 73
IsMatrixOverSemiringSemigroup, 73
IsMaximalSubsemigroup, 189
IsMaxPlusMatrix, 55
IsMaxPlusMatrixCollColl, 55
IsMaxPlusMatrixCollection, 55
IsMaxPlusMatrixMonoid, 73
IsMaxPlusMatrixSemigroup, 73
IsMcAlisterTripleSemigroup, 136
IsMcAlisterTripleSemigroupElement, 138
IsMinPlusMatrix, 55
IsMinPlusMatrixCollColl, 55
IsMinPlusMatrixCollection, 55
IsMinPlusMatrixMonoid, 73
IsMinPlusMatrixSemigroup, 73
IsMonogenicInverseMonoid, 232
IsMonogenicInverseSemigroup, 232
IsMonogenicMonoid, 217
IsMonogenicSemigroup, 217
IsMonoidAsSemigroup, 218
IsMTSE, 138
IsNTPMatrix, 55
IsNTPMatrixCollColl, 56
IsNTPMatrixCollection, 56
IsNTPMatrixMonoid, 73
IsNTPMatrixSemigroup, 73
IsomorphismMonoid, 85
IsomorphismPermGroup, 87
IsomorphismReesMatrixSemigroup
for a D-class, 162
for a semigroup, 90
IsomorphismReesMatrixSemigroupOver-
PermGroup, 90
IsomorphismReesZeroMatrixSemigroup, 90
IsomorphismReesZeroMatrixSemigroup-
OverPermGroup, 90
IsomorphismSemigroup, 84
IsomorphismSemigroups, 272
IsOntoBooleanMat, 67
IsOrthodoxSemigroup, 218
IsPartialOrderBooleanMat, 68
IsPartialPermBipartition, 30
IsPartialPermBipartitionMonoid, 36
IsPartialPermBipartitionSemigroup, 36
IsPartialPermPBR, 46

IsPBR, 37
IsPBRCollColl, 37
IsPBRCollection, 37
IsPBRMonoid, 47
IsPBRSemigroup, 47
IsPermBipartition, 30
IsPermBipartitionGroup, 36
IsPermPBR, 46
IsRectangularBand, 219
IsRectangularGroup, 220
IsReesCongruenceClass, 263
IsReflexiveBooleanMat, 66
IsRegularGreensClass, 157
IsRegularSemigroup, 220
IsRightCongruenceClass, 238
IsRightSemigroupCongruence, 235
IsRightSimple, 215
IsRightZeroSemigroup, 221
IsRMSCongruenceByLinkedTriple, 254
IsRMSCongruenceClassByLinkedTriple, 255
IsRMSIsoByTriple, 277
IsRowTrimBooleanMat, 65
IsRTrivial, 219
IsRZMSCongruenceByLinkedTriple, 254
IsRZMSCongruenceClassByLinkedTriple, 255
IsRZMSIsoByTriple, 277
IsSelfDualSemigroup, 221
IsSemiband, 215
IsSemigroupCongruence, 233
IsSemigroupHomomorphismByFunction, 268
IsSemigroupHomomorphismByImages, 268
IsSemigroupIsomorphismByFunction, 275
IsSemigroupTranslation
  for IsAssociativeElement and IsMultiplicativeElementWithOne, 311
IsSemigroupTranslationCollection, 312
IsSemigroupWithAdjoinedZero, 221
IsSemigroupWithClosedIdempotents, 222
IsSemigroupWithCommutingIdempotents, 222
IsSemilattice, 223
IsSimpleSemigroup, 223
IsSSSE, 134
IsStrongSemilatticeOfSemigroups, 134
IsStzPresentation, 285
IsSubrelation, 252
IsSubsemigroupOfFpMonoid, 284
IsSuperrelation, 252
IsSurjectiveSemigroup, 224
IsSymmetricBooleanMat, 65
IsSynchronizingSemigroup
  for a transformation semigroup, 224
IsTorsion, 74
  for an integer matrix, 70
IsTotalBooleanMat, 67
IsTransBipartition, 29
IsTransformationBooleanMat, 68
IsTransformationPBR, 45
IsTransitive
  for a transformation semigroup and a pos int, 192
  for a transformation semigroup and a set, 192
IsTransitiveBooleanMat, 66
IsTrimBooleanMat, 65
IsTropicalMatrix, 55
IsTropicalMatrixCollection, 55
IsTropicalMatrixMonoid, 73
IsTropicalMatrixSemigroup, 73
IsTropicalMaxPlusMatrix, 55
IsTropicalMaxPlusMatrixCollColl, 56
IsTropicalMaxPlusMatrixCollection, 56
IsTropicalMaxPlusMatrixMonoid, 73
IsTropicalMaxPlusMatrixSemigroup, 73
IsTropicalMinPlusMatrix, 55
IsTropicalMinPlusMatrixCollColl, 56
IsTropicalMinPlusMatrixCollection, 56
IsTropicalMinPlusMatrixMonoid, 73
IsTropicalMinPlusMatrixSemigroup, 73
IsUniformBlockBijection, 31
IsUnitRegularMonoid, 225
IsUniversalPBR, 44
IsUniversalSemigroupCongruence, 264
IsUniversalSemigroupCongruenceClass, 264
IsVertex
  for a graph inverse semigroup element, 122
IsZeroGroup, 225
IsZeroRectangularBand, 225
IsZeroSemigroup, 226
IsZeroSimpleSemigroup, 226
IteratorCanonical, 166

IteratorFromGeneratorsFile, 308
IteratorFromMultiplicationTableFile, 309
IteratorOfDClasses, 155
IteratorOfDClassReps, 155
IteratorOfHClassReps, 155
IteratorOfLeftCongruences
  for a semigroup, 251
  for a semigroup, and a positive integer, 251
  for a semigroup, positive integer, and list or collection, 251
IteratorOfRClasses, 155
IteratorOfRightCongruences
  for a semigroup, 251
  for a semigroup, and a positive integer, 251
  for a semigroup, positive integer, and list or collection, 251

JClasses, 147
JoinIrreducibleDClasses, 197
JoinLeftSemigroupCongruences, 253
JoinRightSemigroupCongruences, 253
JoinSemigroupCongruences, 253
JoinSemilatticeOfCongruences, 248
JonesMonoid, 101

KernelOfSemigroupCongruence, 259
KernelOfSemigroupHomomorphism, 269

LargestElementSemigroup, 193
LatticeOfCongruences
  for a semigroup, 244
  for a semigroup and a list or collection, 245
LatticeOfLeftCongruences
  for a semigroup, 244
  for a semigroup and a list or collection, 245
LatticeOfRightCongruences
  for a semigroup, 245
  for a semigroup and a list or collection, 245
LClass, 145
LClasses, 147
LClassNC, 146
LClassOfHClass, 144
LClassReps, 149
LeftBlocks, 26
LeftCayleyDigraph, 168
LeftCongruencesOfSemigroup
  for a semigroup, 241
  for a semigroup and a multiplicative element collection, 241
LeftGreensMultiplier, 164
LeftInverse
  for a matrix over finite field, 69
LeftOne
  for a bipartition, 17
  for a graph inverse semigroup element, 124
LeftPartOfBitranslation, 312
LeftProjection, 17
LeftSemigroupCongruence, 236
LeftTranslation
  for IsLeftTranslationsSemigroup, IsGeneralMapping, 312
LeftTranslations
  for IsSemigroup and CanUseFroidurePin and IsFinite, 314
LeftTranslationsSemigroupOfFamily
  for IsFamily, 313
LeftZeroSemigroup, 115
Length, 287
LengthOfLongestDClassChain, 153

MajorantClosure, 198
Matrix
  for a filter and a matrix, 51
  for a semiring and a matrix, 51
MaximalDClasses, 150
MaximalLClasses, 150
MaximalRClasses, 150
MaximalSubsemigroups
  for a finite semigroup, 187
  for a finite semigroup and a record, 187
McAlisterTripleSemigroup, 136
McAlisterTripleSemigroupAction, 138
McAlisterTripleSemigroupElement, 138
McAlisterTripleSemigroupGroup, 137
McAlisterTripleSemigroupPartialOrder, 137
McAlisterTripleSemigroupSemilattice, 138
MeetLeftSemigroupCongruences, 252
MeetRightSemigroupCongruences, 252
MeetSemigroupCongruences, 252
MinimalCongruences
  for a congruence poset, 249

for a list or collection, 249
`MinimalCongruencesOfSemigroup`
for a semigroup, 242
for a semigroup and a multiplicative element collection, 242
`MinimalDClass`, 150
`MinimalFactorization`, 173
`MinimalFaithfulTransformationDegree`, 276
`MinimalIdeal`, 180
`MinimalIdealGeneratingSet`, 142
`MinimalInverseMonoidGeneratingSet`, 178
`MinimalInverseSemigroupGeneratingSet`, 178
`MinimalLeftCongruencesOfSemigroup`
for a semigroup, 242
for a semigroup and a multiplicative element collection, 242
`MinimalMonoidGeneratingSet`, 178
`MinimalRightCongruencesOfSemigroup`
for a semigroup, 242
for a semigroup and a multiplicative element collection, 242
`MinimalSemigroupGeneratingSet`, 178
`MinimalWord`
for free inverse semigroup element, 127
`MinimumGroupCongruence`, 260
`Minorants`, 199
`ModularPartitionMonoid`, 105
`MonogenicSemigroup`, 112
`MonoidCartanMatrix`, 209
`MonoidCharacterTable`, 207
`MotzkinMonoid`, 102
`MTSE`, 138
`MultiplicativeNeutralElement`
for an H-class, 161
`MultiplicativeZero`, 182
`MunnSemigroup`, 98

`NambooripadLeqRegularSemigroup`, 204
`NambooripadPartialOrder`, 204
`NaturalLeqBlockBijection`, 24
`NaturalLeqInverseSemigroup`, 197
`NaturalLeqPartialPermBipartition`, 23
`NonTrivialEquivalenceClasses`, 238
`NonTrivialFactorization`, 174
`NormalizedPrincipalFactor`, 163
`NormalizeSemigroup`, 74
`NrBlocks`
for a bipartition, 28
for blocks, 28
`NrDClasses`, 151
`NrHClasses`, 151
`NrIdempotents`, 185
`NrLClasses`, 151
`NrLeftBlocks`, 27
`NrLeftTranslations`
for IsSemigroup and CanUseFroidurePin and IsFinite, 314
`NrMaximalSubsemigroups`, 189
`NrRClasses`, 151
`NrRegularDClasses`, 151
`NrRightBlocks`, 27
`NrTransverseBlocks`
for a bipartition, 25
for blocks, 33
`NumberBlist`, 63
`NumberBooleanMat`, 63
`NumberOfLeftCongruences`
for a semigroup, 250
for a semigroup, and a positive integer, 250
for a semigroup, positive integer, and list or collection, 250
`NumberOfRightCongruences`
for a semigroup, 250
for a semigroup, and a positive integer, 250
for a semigroup, positive integer, and list or collection, 250
`NumberPBR`, 43

`OnBlist`, 62
`OneInverseOfSemigroupElement`, 170
`OnePseudoInverseOfSemigroupElement`, 170
`OnLeftBlocks`, 34
`OnLeftCongruenceClasses`, 240
`OnMultiplicationTable`, 272
`OnRightBlocks`, 34
`OnRightCongruenceClasses`, 241
`Order`, 70
`OrderAntiEndomorphisms`, 95
`OrderEndomorphisms`
monoid of order preserving transformations, 95

ParseRelations, 283
PartialBrauerMonoid, 100
PartialDualSymmetricInverseMonoid, 102
PartialJonesMonoid, 101
PartialOrderAntiEndomorphisms, 95
PartialOrderEndomorphisms, 95
PartialOrderOfDClasses, 152
PartialOrderOfLClasses, 152
PartialOrderOfRClasses, 152
PartialPermLeqBipartition, 23
PartialTransformationMonoid, 95
PartialUniformBlockBijectionMonoid, 103
PartitionMonoid, 99
PBR, 38
PBRNumber, 43
PeriodNTPMatrix, 57
PermLeftQuoBipartition, 24
PermuteMultiplicationTable, 272
PermuteMultiplicationTableNC, 272
Pims
  for a monoid cartan matrix, 209
PlanarModularPartitionMonoid, 105
PlanarPartitionMonoid, 104
PlanarUniformBlockBijectionMonoid, 103
PODI
  monoid of order preserving or reversing partial perms, 98
POI
  monoid of order preserving partial perms, 98
POPI
  monoid of orientation preserving partial perms, 98
PORI
  monoid of orientation preserving or reversing partial perms, 98
PosetOfCongruences, 248
PosetOfPrincipalCongruences
  for a semigroup, 246
  for a semigroup and a multiplicative element collection, 246
PosetOfPrincipalLeftCongruences
  for a semigroup, 246
  for a semigroup and a multiplicative element collection, 246
PosetOfPrincipalRightCongruences
  for a semigroup, 246
  for a semigroup and a multiplicative element collection, 246
PositionCanonical, 167
PrimitiveIdempotents, 199
PrincipalCongruencesOfSemigroup
  for a semigroup, 242
  for a semigroup and a multiplicative element collection, 243
PrincipalFactor, 163
PrincipalLeftCongruencesOfSemigroup
  for a semigroup, 242
  for a semigroup and a multiplicative element collection, 243
PrincipalRightCongruencesOfSemigroup
  for a semigroup, 243
  for a semigroup and a multiplicative element collection, 243
ProjectionFromBlocks, 33

RadialEigenvector, 71
Random
  for a semigroup, 168
RandomBipartition, 19
RandomBlockBijection, 19
RandomInverseMonoid, 92
RandomInverseSemigroup, 92
RandomMatrix
  for a filter and a matrix, 54
  for a semiring and a matrix, 54
RandomMonoid, 92
RandomPBR, 38
RandomSemigroup, 92
RandomWord
  for two integers, 281
Range
  for a graph inverse semigroup element, 122
RankOfBipartition, 25
RankOfBlocks, 33
RClass, 145
RClassBicharacterOfGroupHClass
  for a group H-class, 207
RClasses, 147
RClassNC, 146
RClassOfHClass, 144
RClassReps, 149
ReadGenerators, 307
ReadMultiplicationTable, 308

RectangularBand, 112
ReflexiveBooleanMatMonoid, 109
RegularBooleanMatMonoid, 108
RegularDClasses, 151
RegularRepresentationBicharacter
  for a semigroup, 206
RelationsOfStzPresentation, 286
RepresentativeOfMinimalDClass, 181
RepresentativeOfMinimalIdeal, 181
RightBlocks, 26
RightCayleyDigraph, 168
RightCongruencesOfSemigroup
  for a semigroup, 241
  for a semigroup and a multiplicative element collection, 241
RightCosetsOfInverseSemigroup, 200
RightGreensMultiplier, 164
RightInverse
  for a matrix over finite field, 69
RightOne
  for a bipartition, 17
  for a graph inverse semigroup element, 125
RightProjection, 17
RightSemigroupCongruence, 236
RightZeroSemigroup, 115
RMSCongruenceByLinkedTriple, 254
RMSCongruenceClassByLinkedTriple, 256
RMSIsoByTriple, 277
RMSNormalization, 89
RookMonoid, 98
RookPartitionMonoid, 99
RowSpaceBasis
  for a matrix over finite field, 69
RowSpaceTransformation
  for a matrix over finite field, 69
RowSpaceTransformationInv
  for a matrix over finite field, 69
RZMSCongruenceByLinkedTriple, 254
RZMSCongruenceClassByLinkedTriple, 256
RZMSConnectedComponents, 196
RZMSDigraph, 196
RZMSIsoByTriple, 277
RZMSNormalization, 88

SameMinorantsSubgroup, 200
SchutzenbergerGroup, 159
SemigroupCongruence, 235
SemigroupHomomorphismByFunction, 266
SemigroupHomomorphismByFunctionNC, 266
SemigroupHomomorphismByImages
  for a semigroup and two lists, 266
  for two semigroups, 266
  for two semigroups and a list, 266
  for two semigroups and two lists, 266
SemigroupIdeal, 140
SemigroupIdealOfReesCongruence, 263
SemigroupIsomorphismByFunction, 274
SemigroupIsomorphismByFunctionNC, 274
SemigroupIsomorphismByImages
  for a semigroup and two list, 274
  for two semigroups, 274
  for two semigroups and a list, 274
  for two semigroups and two lists, 274
Semigroups package overview, 8
SEMIGROUPS.DefaultOptionsRec, 80
SemigroupsOfStrongSemilatticeOf-Semigroups, 134
SemigroupsTestAll, 12
SemigroupsTestExtreme, 12
SemigroupsTestInstall, 12
SemigroupsTestStandard, 12
SemilatticeOfStrongSemilatticeOf-Semigroups, 134
SimplifiedFpSemigroup, 298
SimplifyFpSemigroup, 297
SingularApsisMonoid, 105
SingularBrauerMonoid, 100
SingularCrossedApsisMonoid, 105
SingularDualSymmetricInverseMonoid, 102
SingularFactorisableDualSymmetric-InverseMonoid, 103
SingularJonesMonoid, 101
SingularModularPartitionMonoid, 105
SingularOrderEndomorphisms, 95
SingularPartitionMonoid, 100
SingularPlanarModularPartitionMonoid, 105
SingularPlanarPartitionMonoid, 104
SingularPlanarUniformBlockBijection-Monoid, 103
SingularTransformationMonoid, 95
SingularTransformationSemigroup, 95
SingularUniformBlockBijectionMonoid,

103
SLM, 107
SmallerDegreePartialPerm-
Representation, 201
SmallerDegreeTransformation-
Representation, 276
SmallestElementSemigroup, 193
SmallestIdempotentPower, 169
SmallestMultiplicationTable, 270
SmallGeneratingSet, 176
SmallInverseMonoidGeneratingSet, 176
SmallInverseSemigroupGeneratingSet, 176
SmallMonoidGeneratingSet, 176
SmallSemigroupGeneratingSet, 176
Source
for a graph inverse semigroup element, 122
SpecialLinearMonoid, 107
SpectralRadius, 72
SSSE, 133
StandardiseWord, 282
StandardizeWord, 282
Star
for a bipartition, 18
for a PBR, 42
StarOp
for a bipartition, 18
for a PBR, 42
StringToWord
for a string, 282
StrongSemilatticeOfSemigroups, 133
StructureDescription
for an H-class, 162
StructureDescriptionMaximalSubgroups,
161
StructureDescriptionSchutzenberger-
Groups, 160
StzAddGenerator, 292
StzAddRelation, 291
StzIsomorphism, 295
StzPresentation, 285
StzPrintGenerators, 289
StzPrintPresentation, 289
StzPrintRelation, 288
StzPrintRelations, 288
StzRemoveGenerator, 292
StzRemoveRelation, 291
StzSimplifyOnce, 296
StzSimplifyPresentation, 296
StzSubstituteRelation, 293
SubsemigroupByProperty
for a semigroup and function, 82
for a semigroup, function, and limit on the
size of the subsemigroup, 82
Successors, 62
SupersemigroupOfIdeal, 142

TemperleyLiebMonoid, 101
TensorBipartitions
for a list of bipartitions, 18
for a pair of bipartitions, 18
TexString, 304
ThresholdNTPMatrix, 57
ThresholdTropicalMatrix, 56
TietzeBackwardMap, 294
TietzeForwardMap, 294
TikzLeftCayleyDigraph, 306
TikzRightCayleyDigraph, 306
TikzString, 304
TraceOfSemigroupCongruence, 259
TranslationalHull
for IsSemigroup and CanUseFroidurePin and
IsFinite, 314
TriangularBooleanMatMonoid, 110
TrivialCongruence, 265
TrivialSemigroup, 111
TypeLeftTranslationsSemigroupElements
for IsLeftTranslationsSemigroup, 314

UnderlyingRepresentatives
for IsTranslationsSemigroup, 315
UnderlyingSemigroup
for IsTranslationsSemigroup, 313
UnderlyingSemigroupOfCongruencePoset,
247
UnderlyingSemigroupOfSemigroupWith-
AdjoinedZero, 182
UnderlyingSemilatticeOfSemigroups, 135
UniformBlockBijectionMonoid, 103
UniformRandomPartialPerm, 196
UnitriangularBooleanMatMonoid, 110
UniversalPBR, 39
UniversalSemigroupCongruence, 264
UnreducedFpSemigroup

for a presentation, 287
for a semigroup, 299
`UnweightedPrecedenceDigraph`, 72

`VagnerPrestonRepresentation`, 201
`VerticesOfGraphInverseSemigroup`, 124

`WordToString`
for a string and a list, 281
`WreathProduct`, 130
`WriteGenerators`, 307
`WriteMultiplicationTable`, 309

`ZeroSemigroup`, 114