Functionally recursive groups

Self-similar groups

Version 2.4.12

03/12/2022

Laurent Bartholdi

Groups generated by automata or satisfying functional recursions

Laurent Bartholdi Email: laurent.bartholdi@gmail.com
Homepage: https://www.math.uni-sb.de/ag/bartholdi/

Address: Mathematisches Institut
Bunsenstraße 3-5
D-37073 Göttingen
Germany

Abstract

This document describes the package FR, which implements in GAP the basic objects of Mealy machines and functional recursions; and handles groups that they generate.

Groups defined by a recursive action on a rooted tree can be defined in GAP via their recursion. Various algorithms are implemented to manipulate these groups and their elements.

For comments or questions on FR please contact the author; this package is still under development.

Copyright

Acknowledgements

Colophon

This project started in the mid-1990s, when, as a PhD student I did many calculations with groups generated by automata, and realized the similarities between all calculations; it quickly became clear that these calculations could be done much better by a computer than by a human.

The first routines I wrote constructed finite representations of the groups considered, so as to get insight from fast calculations within GAP. The results then had to be proved correct within the infinite group under consideration, and this often involved guessing appropriate words in the infinite group with a given image in the finite quotient.

Around 2000, I had developed quite a few routines, which I assembled in a GAP package, that dealt directly with infinite groups. This package was primitive at its core, but was extended with various routines as they became useful.

I decided in late 2005 to start a new package from scratch, that would encorporate as much functionality as possible in a uniform manner; that would handle semigroups as well as groups; that could be easily extended; and with a complete, understandable documentation. I hope I am not too far from these objectives.

Contents

Chapter 1

Licensing

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program, in the file COPYING. If not, see `https://www.gnu.org/licenses/`.

Chapter 2

FR package

2.1 A brief mathematical introduction

This chapter assumes that you have no familiarity with groups generated by automata. If you do, and wish to see their usage within GAP through a sample session, please skip to Section 2.2. For a more thourough introduction on self-similar groups see [BGN03] or [BGŠ03].

We shall here be interested in groups G defined by their action on a regular rooted tree. Let X be a finite set; and let X^* denote the set of words (free monoid) over X. Then X^* naturally has the structure of a regular rooted tree: the root is the empty word, and vertex $v \in X^*$ is connected to vertex vx for all choices of $x \in X$. Each vertex except the root therefore has $\#X + 1$ neighbours.

Let W denote the automorphism group of the graph X^*. Given $a \in W$, we may restrict its action to $X \subset X^*$, and obtain a permutation π_a on X, called the *activity* of a. We may also obtain, for all $x \in X$, a tree automorphism $a_x \in W$, called the *state of a at x*, by the formula

$$(v)a_x = w \quad \text{if} \quad (xv)a = x^{\pi_a}w.$$

The data (a_x, π_a) determine uniquely the automorphism a, and any choice of a_x and π_a defines a tree isometry. We therefore have a group isomorphism

$$\phi : W \to W \wr Sym(X),$$

called the *Wreath recursion*. The image of ϕ is the permutational wreath product $W^X \rtimes Sym(X)$.

The state a_x should be interpreted as the restriction of the action of a on the subtree xX^*; the automorphism a is defined by acting first on each of the subtrees of the form xX^* by its respective state, and then permuting these subtrees according to π_a. The wreath recursion can be iterated on the states of a, to define states a_v for any $v \in X^*$.

The automorphism $a \in W$ may be represented by a graph, as follows. There is one vertex for each state a_v of a, labeled π_{a_v}; and for each $x \in X$ there is one edge from state a_v to state a_{vx}, labeled x. This graph is nothing but a quotient of the regular rooted tree X^*, where vertices v and w are identified if $a_v = a_w$. Again, this graph, with a choice of initial vertex, determines uniquely the automorphism a.

This graph may be conveniently encoded in what is called a *Moore machine*: it consists of a set Q, the vertex set of the graph; an alphabet, X; a 'transition' function $\phi : Q \times X \to Q$, where $\phi(q, x)$ is the endpoint of the edge starting at q and labeled x; and a labeling π of X by the symmetric group on X. We will use the equivalent *Mealy machines*, given by a 'transition' function $\phi : Q \times X \to X \times Q$, encoding both ϕ and π together.

Of particular interest are *finite-state automorphisms*: these are automorphisms whose Mealy machine has finitely many states. The product and inverse of finite-state automorphisms is again finite-state.

A subgroup $G \leq W$ is *self-similar* if $G^\phi \subset G \wr Sym(X)$. This is equivalent to asking, for every $a \in G$, that all of its states a_x also belong to G.

The following important properties have also been considered. A subgroup $G \leq W$ is *level-transitive* if its action is transitive on all the G-subsets X^n. It is *weakly branched* if it is level-transitive, and for every $v \in X^*$ there is a non-trivial $a_v \in G$ that fixes $X^* \setminus vX^*$. It is *branched* if furthermore for each $n \in \mathbb{N}$ the group generated by all such a_v for all v of length n has finite index in G.

A self-similar finitely generated group $G \leq W$ is *contracting* if there are constants $K, n \in \mathbb{N}$ and $\lambda < 1$ such that $|a_v| \leq \lambda |a| + K$ for all $a \in G$ and $v \in X^n$; here $|a|$ denotes the minimal number of generators needed to express a. It then follows that there exists a finite set $N \subset G$ such that for all $a \in G$, all but finitely many of the states of a belong to N. The minimal such N is called the *nucleus* of G. Since the states of elements of the nucleus are again in the nucleus, we see that the nucleus is naturally a Mealy machine. By considering all elements of W obtained from this Mealy machine by choosing all possible initial states, we obtain a generating set for G made of all states of a single machine; this is the *group generated* by the machine.

In this package, we are mainly interested in self-similar groups of finite-state automorphisms. The reason is historical: Aleshin [Ale83], and later Grigorchuk [Gri80] and Gupta and Sidki [GS83] constructed peculiar examples of groups using self-similar finite-state automorphisms. All these groups can be defined by drawing a small machine (at most five vertices) and considering the group that they generate.

We assumed for simplicity that the elements a were invertible. Actually, in the definition of Mealy machines it makes sense to accept arbitrary maps, and not necessarily bijections of X as a label at each vertex. One may in this way define peculiar semigroups.

2.2 An example session

This is a brief introduction describing some of the simpler features of the FR package. It assumes you have some familiarity with the theory of groups defined by automata; if not, a brief mathematical introduction may be found in Section 2.1. We show here and comment a typical use of the package.

The package is installed by unpacking the archive in the pkg/ directory of your GAP installation. It can also be placed in a local directory, which must be added to the load-path by invoking gap with the -l option.

```
————————————————————————————— Example —————————————————————————————
gap> LoadPackage("fr");
----------------------------------------------------------------
Loading FR 0.857142p5 (Functionally recursive and automata groups)
by Laurent Bartholdi (http://www.uni-math.gwdg.de/laurent)
----------------------------------------------------------------
true
```

Many FR groups are predefined by FR, see Chapter 9. We consider here the *Basilica group*, considered in [GŻ02] and [BV05].

We may start by defining a group: it has two generators *a* and *b*, satisfying the specified recursions.

```
————————————————————————————— Example —————————————————————————————
gap> B := FRGroup("a=<1,b>(1,2)","b=<1,a>",IsFRMealyElement);
<self-similar group over [ 1 .. 2 ] with 2 generators>
```

```
gap> AssignGeneratorVariables(B);
#I  Assigned the global variables [ a, b ]
```

We have just created the group $B = \langle a, b \rangle$.

Note that this group is predefined as `BasilicaGroup`. We now compute the decompositions of the generators:

```
─────────────── Example ───────────────
gap> DecompositionOfFRElement(a); DecompositionOfFRElement(b);
[ [ <2|identity ...>, <2|b> ], [ 2, 1 ] ]
[ [ <2|identity ...>, <2|a> ], [ 1, 2 ] ]
```

Elements are described as words in the generators; they are printed as `<2|a>`, where the 2 reminds of the degree of the tree on which *a* acts.

The optional argument `IsFRElement` (10.2.11) tells FR to store elements in this way. This representation is always possible, but it is usually inefficient for calculations. The argument `IsMealyElement` (10.2.4) forces FR to use a more efficient representation, which in some cases may take an infinite time to set up. With no extra argument, FR does what it thinks is best. The advantages of both representations are sometimes obtained by the argument `IsFRMealyElement` (10.2.12), which stores both representations.

Elements act on sequences over $\{1, 2\}$. The action is computed in the standard manner:

```
─────────────── Example ───────────────
gap> 1^a; [1]^a; [1,1]^a;
2
[ 2 ]
[ 2, 1 ]
```

Periodic sequences are also implemented in FR; they are constructed by giving the period and preperiod. The period is printed by preceding it with a "/":

```
─────────────── Example ───────────────
gap> v := PeriodicList([1],[2]);
[ 1, / 2 ]
gap> v^a; v^(a^2);
[/ 2 ]
[/ 1, 2 ]
gap> last{[1..10]};
[ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ]
```

Most computations are much more efficient if *B*'s elements are converted to *Mealy representation*,

```
─────────────── Example ───────────────
gap> Bm := Image(IsomorphismMealyGroup(B));
<recursive group over [ 1 .. 2 ] with 2 generators>
gap> a := Bm.1; b := Bm.2;
<Mealy element on alphabet [ 1, 2 ] with 3 states>
<Mealy element on alphabet [ 1, 2 ] with 3 states>
```

This could have been done automatically by specifying `IsMealyElement` as last argument in the call to `FRGroup`.

The group *B* is torsion-free, and its elements are bounded automata. Although torsion-freeness is difficult to check for FR, it can be checked on individual elements:

```
────────────────────────── Example ──────────────────────────
gap> IsBoundedFRSemigroup(Bm);
true
gap> Order(a); Order(b);
infinity
infinity
gap> g := PseudoRandom(B);; Length(InitialState(g));
4679
gap> Order(g); time;
infinity
2599
```

The group B is weakly branched; more precisely, the derived subgroup B' contains $B' \times B'$. To prove that, it suffices to check $[a,b] \times 1 \in B'$ and $1 \times [a,b] \in B'$. These elements are constructed using VertexElement (4.1.5):

```
────────────────────────── Example ──────────────────────────
gap> c := Comm(a,b);
<Mealy element on alphabet [ 1, 2 ] with 9 states>
gap> K := NormalClosure(Bm,Group(c));
<self-similar group over [ 1 .. 2 ] with 3 generators>
gap> VertexElement(1,c) in K; VertexElement(1,c) in K;
true
true
gap> DecompositionOfFRElement(VertexElement(1,c))=[[c,One(Bm)],[1,2]];
true
gap> VertexElement(2,c)=Comm(b,a^2);
true
```

Note that we had to guess the form of the element VertexElement(2,c) above. This could have been found out by GAP using ShortGroupWordInSet (11.4.2).

We may also check the relations $[b^p, (b^p)^{a^p}] = 1$ and $[a^{2p}, (a^{2p})^{b^p}]$ for p any power of 2:

```
────────────────────────── Example ──────────────────────────
gap> ForAll([0..10],i->IsOne(Comm(b^(2^i),(b^(2^i))^((a^(2^i)))))); time;
true
1361
```

Since the group B is bounded, it is contracting. We compute its nucleus:

```
────────────────────────── Example ──────────────────────────
gap> NucleusOfFRSemigroup(B);
[ <2|identity ...>, <2|b>, <2|b^-1>, <2|a>, <2|a^-1>, <2|b^-1*a>, <2|a^-1*b> ]
```

We then compute the Mealy machine with stateset this nucleus, and draw it graphically (this requires the external programs graphviz and imagemagick):

```
────────────────────────── Example ──────────────────────────
gap> N := NucleusMachine(B);
<Mealy machine on alphabet [ 1, 2 ] with 7 states>
gap> Draw(N);
```

We may also draw powers of the dual automaton: these are approximations to the Schreier graph of B. However, we also construct a smaller Mealy machine with states only a and b, which give better images:

```
                              ── Example ──
gap> Draw(DualMachine(N)^3);
gap> M := AsMealyMachine(FRMachine(a))[1];
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> Draw(DualMachine(M)^4);
```

These Schreier graphs are orbits of the group; they can be displayed as follows:

```
                              ── Example ──
gap> WordGrowth(B:point:=[1,1,1,1],draw);
```

More properties of B can be checked, or experimented with, on its finite quotients obtained by truncating the tree on which B acts at a given length. `PermGroup(B,n)` constructs a permutation group which is the natural quotient of B acting on 2^n points:

```
                              ── Example ──
gap> G := PermGroup(B,7);
<permutation group with 2 generators>
gap> Size(G); LogInt(last,2);
309485009821345068724781056
88
```

We may "guess" the structure of the Lie algebra of B by examining the ranks of the successive quotients along its Jennings series:

```
                              ── Example ──
gap> J := JenningsLieAlgebra(G); time;
<Lie algebra of dimension 88 over GF(2)>
18035
gap> List([1..15],i->Dimension(Grading(J).hom_components(i)));
[ 2, 3, 1, 4, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1 ]
```

The "4" in position 8 of that list should really be a "5"; computations on finite quotients of B usually give lower bounds for invariants of B. In that case, we guess that the ranks behave like a "ruler" function, i.e. that the rank of the homogeneous component of degree i is $2 + v_2(i)$ if i is a power of 2 and is $1 + v_2(i)$ otherwise; here $v_2(i)$ is the number of times 2 divides i.

Chapter 3

Functionally recursive machines

At the core of this package are *functionally recursive machines*. The internals of specific machines will be described later, but each machine M has an associated *alphabet X*, a *set of states Q*, and a *transition function* $\phi : Q \times X \rightarrow X \times Q$. An *element*, as we will see in Section 4, is given by a machine and an initial state $q \in Q$.

The element (M, q) defines a transformation on the set X^* of strings (finite or infinite) over the alphabet X, as follows: the empty string is always fixed. Given a string $x_1 x_2 \ldots x_n$ with $n \geq 0$, compute $\phi(q, x_1) = (y_1, r)$; then compute the action of (M, r) on the string $x_2 \ldots x_n$, and call the result $y_2 \ldots y_n$. Then the action of (M, q) on $x_1 x_2 \ldots x_n$ yields $y_1 y_2 \ldots y_n$.

This can be understood more formally as follows. The transition function ϕ induces a map $\phi^n : Q \times X^n \rightarrow X^n \times Q$, defined by successively applying ϕ to move the Q from the left to the right. Similarly, ϕ can be extended to a map $Q^m \times X^n \rightarrow X^n \times Q^m$.

We see that the action on finite strings preserves their length, and also preserves prefixes: if (M, q) maps $x_1 \ldots x_n$ to $y_1 \ldots y_m$, then necessarily $m = n$; and if $k < n$ then T maps $x_1 \ldots x_k$ to $y_1 \ldots y_k$.

The strings over the alphabet X can be naturally organised into a rooted tree. The root represents the empty string, and the strings $x_1 \ldots x_n$ and $x_1 \ldots x_{n+1}$ are connected by an edge, for all $x_i \in X$.

3.1 Types of machines

Machines must be accessible to computation; therefore it is reasonable to assume that their alphabet X is finite.

If the stateset Q is also finite, the machine is called a *Mealy machine*, and its transition function ϕ can be stored as a table.

More general machines are obtained if one allows the stateset Q to be a free group/semigroup/monoid generated by a finite set S, and the transition function ϕ to be specified on $S \times X$. Its values are then extended naturally by composition.

Machines store their transitions (second coordinate of ϕ), and their output, (first coordinate of ϕ) in a matrix indexed by state and letter. In particular, `PermList(output[state])` gives the action on the first level.

Because of the way GAP handles permutations and transformations, a permutation is never equal to a transformation, even though both can answer `true` to *IsOne*. Therefore, FR stores the output as a list, which can be then accessed (e.g. in commands such as `ActivityPerm` and `ActivityTransformation` either as a permutation or as a transformation. The command `Activity` itself will return a permutation if possible, and otherwise a transformation.

3.2 Products of machines

Machines can be combined in different manners. If two machines act on the same alphabet, then their *sum* and *product* are defined as machines acting again on the same alphabet; the statesets are the free products (which is also their sum, in the category of semigroups) of the respective statesets. The sum or product of machines has a stateset of highest possible category, i.e. is a group unless some argument is a monoid, and a monoid unless some argument is a semigroup. The transition and output functions are the union of the respective functions of their arguments.

If a non-empty collection of machines have same stateset, then their *tensor sum* and *tensor product* are machines again with same stateset; the alphabets on which the machines act are the disjoint union, respectively cartesian product, of the arguments' alphabets. The transition and output functions are again the union or composition of the respective functions of their arguments.

The *direct sum* and *direct product* of a collection of machines are always defined. Its stateset is generated by the union of the arguments' statesets, as for a sum or a product; its alphabet is the disjoint union, respectively cartesian product of its arguments' alphabets, as for a tensor sum or product. The transition and output functions are again the union of the respective functions of their arguments.

3.3 Creators for `FRMachine`s

3.3.1 FRMachineNC (family,free,listlist,list)

▷ FRMachineNC(`fam, free, transitions, outputs`)   (operation)
    Returns: A new FR machine.

This function constructs a new FR machine, belonging to family `fam`. It has stateset the free group/semigroup/monoid `free`, and transitions described by `states` and `outputs`.

`transitions` is a list of lists; `transitions[s][x]` is a word in `free`, which is the state reached by the machine when started in state s and fed input x.

`outputs` is also a list of lists; `outputs[s][x]` is the output produced by the machine is in state s and inputs x.

```
——————————————— Example ———————————————
gap> f := FreeGroup(2);
<free group on the generators [ f1, f2 ]>
gap> m := FRMachineNC(FRMFamily([1,2]),f,[[One(f),f.1],[One(f),f.2^-1]],
                      [[2,1],[1,2]]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1, f2 ] )>
```

3.3.2 FRMachine ([list,]list,list)

▷ FRMachine(`[names, ]transitions, outputs`)   (operation)
▷ FRMachine(`free, transitions, outputs`)   (operation)
    Returns: A new FR machine.

This function constructs a new FR machine. It has stateset a free group/semigroup/monoid, and structure described by `transitions` and `outputs`.

If there is an argument `free`, it is the free group/monoid/semigroup to be used as stateset. Otherwise, the stateset will be guessed from the `transitions` and `outputs`; it will be a free group if all states are invertible, and a monoid otherwise. `names` is then an optional list, with at position s a

string naming generator `s` of the stateset. If `names` contains too few entries, they are completed by the names `__1,__2,...`.

transitions is a list of lists; `transitions[s][x]` is either an associative word, or a list of integers describing the state reached by the machine when started in state `s` and fed input `x`. Positive integers indicate a generator index, negative integers its inverse, the empty list in the identity state, and lists of length greater than one indicate a product of states. If an entry is an FR element, then its machine is incorporated into the newly constructed one.

outputs is a list; at position `s` it contains a permutation, a transformation, or a list of integers (the images of a transformation), describing the activity of state `s`. If all states are invertible, the outputs are all converted to permutations, while if there is a non-invertible state then the outputs are all converted to transformations.

```
 _____ Example _____
  gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);
  <FR machine with alphabet [ 1, 2 ] on Group( [ tau, mu ] )>
  gap> m=n;
  true
  gap> Display(n);
       |    1          2
  -----+--------+---------+
   tau | <id>,2      tau,1
    mu | <id>,2    mu^-1,1
  -----+--------+---------+
  gap> m := FRMachine([[[],[FRElement(n,1)]]],[()]);
  <FR machine with alphabet [ 1, 2 ] on Group( [ f1, f2, f3 ] )>
  gap> Display(m);
      |    1          2
  ----+--------+---------+
   f1 | <id>,1      f2,2
   f2 | <id>,2      f2,1
   f3 | <id>,2   f1^-1,1
  ----+--------+---------+
  gap> f := FreeGroup(2);
  <free group on the generators [ f1, f2 ]>
  gap> p := FRMachine(f,[[One(f),f.1],[One(f),f.2^-1]],[(1,2),(1,2)]);
  <FR machine with alphabet [ 1, 2 ] on Group( [ f1, f2 ] )>
  gap> n=p;
  true
```

3.3.3 UnderlyingFRMachine

▷ UnderlyingFRMachine(*obj*)   (attribute)

Returns: An FR machine underlying *obj*.

FR elements, FR groups etc. often have an underlying FR machine, which is returned by this command.

```
 _____ Example _____
  gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
  <FR machine with alphabet [ 1, 2 ] on Group( [ a, b ] )>
  gap> a := FRElement(m,1); b := FRElement(m,2);
  <2|a>
  <2|b>
```

```
gap> UnderlyingFRMachine(a)=m;
true
```

3.3.4 AsGroupFRMachine

▷ AsGroupFRMachine(*m*)   (attribute)
▷ AsMonoidFRMachine(*m*)   (attribute)
▷ AsSemigroupFRMachine(*m*)   (attribute)

Returns: An FR machine isomorphic to *m*, on a free group/monoid/semigroup.

This function constructs, from the FR machine *m*, an isomorphic FR machine n with a free group/monoid/semigroup as stateset. The attribute Correspondence(n) is a mapping (homomorphism or list) from the stateset of *m* to the stateset of n.

m can be an arbitrary FR machine, or can be an free group/monoid/semigroup endomorphism. It is then converted to an FR machine on a 1-letter alphabet.

```
─────────────────────────── Example ───────────────────────────
gap> s := FreeSemigroup(1);;
gap> sm := FRMachine(s,[[GeneratorsOfSemigroup(s)[1],
                         GeneratorsOfSemigroup(s)[1]^2]],[(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Semigroup( [ s1 ] )>
gap> m := FreeMonoid(1);;
gap> mm := FRMachine(m,[[One(m),GeneratorsOfMonoid(m)[1]^2]],[(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Monoid( [ m1 ], ... )>
gap> g := FreeGroup(1);;
gap> gm := FRMachine(g,[[One(g),GeneratorsOfGroup(g)[1]^-2]],[(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
gap> AsGroupFRMachine(sm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
     |   1        2
----+------+--------+
 f1 | f1,2   f1^2,1
----+------+--------+
gap> Correspondence(last);
MappingByFunction( <free semigroup on the generators
[ s1 ]>, <free group on the generators [ f1 ]>, function( w ) ... end )
gap> AsGroupFRMachine(mm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
     |    1         2
----+--------+--------+
 f1 | <id>,2   f1^2,1
----+--------+--------+
gap> AsGroupFRMachine(gm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
     |    1         2
----+--------+---------+
 f1 | <id>,2   f1^-2,1
----+--------+---------+
gap> AsMonoidFRMachine(sm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Monoid( [ m1 ], ... )>
     |   1        2
----+------+--------+
 m1 | m1,2   m1^2,1
```

```
   ----+------+--------+
gap> AsMonoidFRMachine(mm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Monoid( [ m1 ], ... )>
    |    1        2
----+--------+--------+
 m1 | <id>,2   m1^2,1
----+--------+--------+
gap> AsMonoidFRMachine(gm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Monoid( [ m1, m2 ], ... )>
    |    1        2
----+--------+--------+
 m1 | <id>,2   m2^2,1
 m2 | m1^2,2   <id>,1
----+--------+--------+
gap> AsSemigroupFRMachine(sm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Semigroup( [ s1 ] )>
    |   1        2
----+------+--------+
 s1 | s1,2   s1^2,1
----+------+--------+
gap> AsSemigroupFRMachine(mm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Semigroup( [ s1, s2 ] )>
    |   1        2
----+------+--------+
 s1 | s2,2   s1^2,1
 s2 | s2,1    s2,2
----+------+--------+
gap> AsSemigroupFRMachine(gm); Display(last);
<FR machine with alphabet [ 1, 2 ] on Semigroup( [ s1, s2, s3 ] )>
    |    1        2
----+--------+--------+
 s1 |   s3,2    s2^2,1
 s2 | s1^2,2     s3,1
 s3 |   s3,1     s3,2
----+--------+--------+
gap>
gap> Display(GuptaSidkiMachines(3));
   | 1     2     3
---+-----+-----+-----+
 a | a,1   a,2   a,3
 b | a,2   a,3   a,1
 c | a,3   a,1   a,2
 d | b,1   c,2   d,3
---+-----+-----+-----+
gap> AsGroupFRMachine(GuptaSidkiMachines(3));
<FR machine with alphabet [ 1 .. 3 ] on Group( [ f1, f2 ] )>
gap> Display(last);
    |    1        2         3
----+--------+---------+--------+
 f1 | <id>,2    <id>,3    <id>,1
 f2 |   f1,1   f1^-1,2      f2,3
----+--------+---------+--------+
```

```
gap> Correspondence(last);
[ <identity ...>, f1, f1^-1, f2 ]
gap> AsGroupFRMachine(GroupHomomorphism(g,g,[g.1],[g.1^3]));
<FR machine with alphabet [ 1 ] on Group( [ f1 ] )>
gap> Display(last);
 G  |     1
----+--------+
 f1 | f1^3,1
----+--------+
```

3.3.5 AsGroupFRMachine (endomorphism)

▷ AsGroupFRMachine(*f*)   (attribute)

▷ AsMonoidFRMachine(*f*)   (attribute)

▷ AsSemigroupFRMachine(*f*)   (attribute)

   Returns: An FR machine.

   This function creates an FR machine on a 1-letter alphabet, that represents the endomorphism *f*. It is specially useful when combined with products of machines; indeed the usual product of machines corresponds to composition of endomorphisms.

```
————————————————— Example ———————————————————
gap> f := FreeGroup(2);;
gap> h := GroupHomomorphismByImages(f,f,[f.1,f.2],[f.2,f.1*f.2]);
[ f1, f2 ] -> [ f2, f1*f2 ]
gap> m := AsGroupFRMachine(h);
<FR machine with alphabet [ 1 ] on Group( [ f1, f2 ] )>
gap> mm := TensorProduct(m,m);
<FR machine with alphabet [ 1 ] on Group( [ f1, f2 ] )>
gap> Display(mm);
 G  |         1
----+------------+
 f1 |    f1*f2,1
 f2 | f2*f1*f2,1
----+------------+
```

3.4 Attributes for FRMachines

3.4.1 StateSet (FR machine)

▷ StateSet(*m*)   (attribute)

   Returns: The set of states associated with *m*.

   This function returns the stateset of *m*. It can be either a list (if the machine is of Mealy type), or a free group/semigroup/monoid (in all other cases).

```
————————————————— Example ———————————————————
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ tau, mu ] )>
gap> StateSet(n);
<free group on the generators [ tau, mu ]>
gap> StateSet(AsMealyMachine(n));
[ 1 .. 4 ]
```

3.4.2 GeneratorsOfFRMachine

▷ GeneratorsOfFRMachine(*m*)   (attribute)

    Returns: The generating set of the stateset of *m*.

    This function returns the generating set of the stateset of *m*. If *m* is a Mealy machine, it returs the stateset.

```
───────────────────── Example ─────────────────────
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ tau, mu ] )>
gap> GeneratorsOfFRMachine(n);
[ tau, mu ]
```

3.4.3 Output (FR machine)

▷ Output(*m*)   (operation)
▷ Output(*m, s*)   (operation)
▷ Output(*m, s, x*)   (operation)

    Returns: A transformation of *m*'s alphabet.

    In the first form, this function returns the output of *m*.

    In the second form, this function returns the transformation of *m*'s alphabet associated with state *s*. This transformation is returned as a list of images.

    s is also allowed to be a list, in which case it is interpreted as the corresponding product of states.

    In the third form, the result is actually the image of *x* under Output(m,s).

```
───────────────────── Example ─────────────────────
gap> n := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
<FR machine with alphabet [ 1, 2 ] on Group( [ a, b ] )>
gap> Output(n,[1,2]);
[2,1]
gap> Output(n,Product(GeneratorsOfFRMachine(n)));
[2,1]
```

3.4.4 Transition (FR machine,state,input)

▷ Transition(*m, s, i*)   (operation)

    Returns: An element of *m*'s stateset.

    This function returns the state reached by *m* when started in state *s* and fed input *i*. This input may be an alphabet letter or a sequence of alphabet letters.

```
───────────────────── Example ─────────────────────
gap> n := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
<FR machine with alphabet [ 1, 2 ] on Group( [ a, b ] )>
gap> Transition(n,[2,1],2);
a*b
gap> Transition(n,Product(GeneratorsOfFRMachine(n))^2,1);
a*b
```

3.4.5 Transitions (FR machine,state)

▷ Transitions(*m, s*)   (operation)

    Returns: A list of elements of *m*'s stateset.

This function returns the states reached by `m` when started in state `s` and fed inputs from the alphabet. The state may be expressed as a word or as a list of states.

```
─────────────────────── Example ───────────────────────
 gap> n := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
 <FR machine with alphabet [ 1, 2 ] on Group( [ a, b ] )>
 gap> Transitions(n,[2,1]);
 [ <identity ...>, a*b ]
 gap> Transitions(n,Product(GeneratorsOfFRMachine(n))^2);
 [ a*b, b*a ]
```

3.4.6 WreathRecursion

▷ WreathRecursion(*m*)   (attribute)

Returns: A function on the stateset of *m*.

This function returns a function on *m*'s stateset. This function, on receiving state *q* as input, returns a list. Its first entry is a list indexed by *m*'s alphabet, with in position *x* the state *m* would be in if it received input *x* when in state *q*. The second entry is the list of the permutation of *m*'s alphabet induced by *q*.

`WreathRecursion(machine)(q)[1][a]` is equal to `Transition(machine,q,a)` and `WreathRecursion(machine)(q)[2]` is equal to `Output(machine,q)`.

```
─────────────────────── Example ───────────────────────
 gap> n := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
 <FR machine with alphabet [ 1, 2 ] on Group( [ a, b ] )>
 gap> WreathRecursion(n)(GeneratorsOfFRMachine(n)[1]);
 [ [ <identity ...>, b ], [2,1] ]
 gap> WreathRecursion(n)(GeneratorsOfFRMachine(n)[2]);
 [ [ <identity ...>, a ], [1,2] ]
```

3.5 Operations for FRMachines

3.5.1 StructuralGroup

▷ StructuralGroup(*m*)   (operation)
▷ StructuralMonoid(*m*)   (operation)
▷ StructuralSemigroup(*m*)   (operation)

Returns: A finitely presented group/monoid/semigroup capturing the structure of *m*.

This function returns a finitely presented group/monoid/semigroup, with generators the union of the AlphabetOfFRObject (10.1.3) and GeneratorsOfFRMachine (3.4.2) of *m*, and relations all $qa' = aq'$ whenever $\phi(q,a) = (a',q')$.

```
─────────────────────── Example ───────────────────────
 gap> n := FRMachine(["a","b","c"],[[[2],[3]],[[3],[2]],[[1],[1]]],[(1,2),(1,2),()]);
 <FR machine with alphabet [ 1, 2 ] on Group( [ a, b, c ] )>
 gap> StructuralGroup(n);
 <fp group on the generators [ a, b, c, 1, 2 ]>
 gap> RelatorsOfFpGroup(last);
 [ a*2*b^-1*1^-1, a*1*c^-1*2^-1, b*2*c^-1*1^-1,
   b*1*b^-1*2^-1, c*1*a^-1*1^-1, c*2*a^-1*2^-1 ]
 gap> SimplifiedFpGroup(last2);
 <fp group on the generators [ a, 1 ]>
```

```
gap> RelatorsOfFpGroup(last);
[ 1^-1*a^2*1^4*a^-2*1^-1*a*1^-2*a^-1, 1*a*1^-1*a*1^2*a^-1*1*a^-2*1^-3*a,
  1^-1*a^2*1^2*a^-1*1^-1*a*1^2*a^-2*1^-2 ]
```

3.5.2 \+

▷ \+(*m1, m2*)   (method)

Returns: A new FR machine, in the same family as its arguments.

This function returns a new FR machine r, with stateset generated by the union of the statesets of its arguments. The arguments m1 and m2 must operate on the same alphabet. If the stateset of m1 is free on n_1 letters and the stateset of m2 is free on n_2 letters, then the stateset of their sum is free on $n_1 + n_2$ letters, with the first n_1 identified with m1's states and the next n_2 with m2's.

The transition and output functions are naturally extended to the sum.

The arguments may be free group, semigroup or monoid machines. The sum is in the weakest containing category: it is a group machine if both arguments are group machines; a monoid if both are either group of monoid machines; and a semigroup machine otherwise.

The maps from the stateset of m1 and m2 to the stateset of r can be recovered as Correspondence(r)[1] and Correspondence(r)[2]; see Correspondence (3.5.12).

```
———————————————————————— Example ————————————————————————
gap> tau := FRMachine([[[],[1]]],[(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
gap> mu := FRMachine([[[],[-1]]],[(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
gap> sum := tau+mu;; Display(sum);
     |    1          2
-----+--------+----------+
 f11 | <id>,2       f11,1
 f12 | <id>,2    f12^-1,1
-----+--------+----------+
gap> Correspondence(sum)[1];
[ f1 ] -> [ f11 ]
gap> GeneratorsOfFRMachine(tau)[1]^last;
f11
```

3.5.3 *

▷ *(*machine1, machine2*)   (method)

Returns: A new FR machine, in the same family as its arguments.

The product of two FR machines coincides with their sum, since the natural free object mapping to the product of the statesets is generated by the union of the statesets. See therefore \+ (3.5.2).

3.5.4 **TensorSumOp (FR Machines)**

▷ TensorSumOp(*FR_machines, machine*)   (method)

Returns: A new FR machine on the disjoint union of the arguments' alphabets.

The tensor sum of FR machines with same stateset is defined as the FR machine acting on the disjoint union of the alphabets; if these alphabets are [1..n1] up to [1..nk], then the alphabet of their sum is [1..n1+...+nk] and the transition functions are similarly concatenated.

The first argument is a list; the second argument is any element of that list, and is used only to improve the method selection algorithm.

```
————————————————————————— Example —————————————————————————
gap> m := TensorSum(AddingMachine(2),AddingMachine(3),AddingMachine(4));
AddingMachine(2)(+)AddingMachine(3)(+)AddingMachine(4)
gap> Display(m);
   |  1     2     3     4     5     6     7     8     9
---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 a | a,1   a,2   a,3   a,4   a,5   a,6   a,7   a,8   a,9
 b | a,2   b,1   a,4   a,5   b,3   a,7   a,8   a,9   b,6
---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

3.5.5   TensorProductOp (FR Machines)

▷ TensorProductOp(*FR, machines, machine*)   (method)

   Returns: A new FR machine on the cartesian product of the arguments' alphabets.

   The tensor product of FR machines with same stateset is defined as the FR machine acting on the cartesian product of the alphabets. The transition function and output function behave as if a single letter, in the tensor product's alphabet, were a word (read from left to right) in the machines' alphabets.

   The first argument is a list; the second argument is any element of that list, and is used only to improve the method selection algorithm.

```
————————————————————————— Example —————————————————————————
gap> m := TensorProduct(AddingMachine(2),AddingMachine(3));
AddingMachine(2)(*)AddingMachine(3)
gap> Display(last);
   |  1     2     3     4     5     6
---+-----+-----+-----+-----+-----+-----+
 a | a,1   a,2   a,3   a,4   a,5   a,6
 b | a,4   a,5   a,6   a,2   a,3   b,1
---+-----+-----+-----+-----+-----+-----+
```

3.5.6   DirectSumOp (FR Machines)

▷ DirectSumOp(*FR, machines, machine*)   (method)

   Returns: A new FR machine on the disjoint union of the arguments' alphabets.

   The direct sum of FR machines is defined as the FR machine with stateset generated by the disjoint union of the statesets, acting on the disjoint union of the alphabets; if these alphabets are [1..n1] up to [1..nk], then the alphabet of their sum is [1..n1+...+nk] and the output and transition functions are similarly concatenated.

   The first argument is a list; the second argument is any element of that list, and is used only to improve the method selection algorithm.

```
————————————————————————— Example —————————————————————————
gap> m := DirectSum(AddingMachine(2),AddingMachine(3),AddingMachine(4));
AddingMachine(2)#AddingMachine(3)#AddingMachine(4)
gap> Display(m);
   |  1     2     3     4     5     6     7     8     9
---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 a | a,1   a,2   a,3   a,4   a,5   a,6   a,7   a,8   a,9
 b | a,2   b,1   b,3   b,4   b,5   b,6   b,7   b,8   b,9
```

```
  c | c,1   c,2   a,3   a,4   a,5   c,6   c,7   c,8   c,9
  d | d,1   d,2   a,4   a,5   b,3   d,6   d,7   d,8   d,9
  e | e,1   e,2   e,3   e,4   e,5   a,6   a,7   a,8   a,9
  f | f,1   f,2   f,3   f,4   f,5   a,7   a,8   a,9   b,6
  ---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

3.5.7 DirectProductOp (FR Machines)

▷ DirectProductOp(*FR, machines, machine*) (method)

Returns: A new FR machine on the cartesian product of the arguments' alphabets.

The direct product of FR machines is defined as the FR machine with stateset generated by the product of the statesets, acting on the product of the alphabets; if these alphabets are [1..n1] up to [1..nk], then the alphabet of their product is [1..n1*...*nk] and the output and transition functions act component-wise.

The first argument is a list; the second argument is any element of that list, and is used only to improve the method selection algorithm.

```
 ────────────────────────────── Example ──────────────────────────────
 gap> m := DirectProduct(AddingMachine(2),AddingMachine(3));
 AddingMachine(2)xAddingMachine(3)
 gap> Display(last);
    |  1     2     3     4     5     6
 ---+-----+-----+-----+-----+-----+-----+
  a | a,1   a,2   a,3   a,4   a,5   a,6
  b | a,2   a,3   b,1   a,5   a,6   b,4
  c | a,4   a,5   a,6   c,1   c,2   c,3
  d | a,5   a,6   b,4   c,2   c,3   d,1
 ---+-----+-----+-----+-----+-----+-----+
```

3.5.8 TreeWreathProduct (FR machine)

▷ TreeWreathProduct(*m, n, x0, y0*) (method)

Returns: A new FR machine on the cartesian product of the arguments' alphabets.

The *tree-wreath product* of two FR machines is a machine acting on the product of its arguments' alphabets X, Y, in such a way that many images of the first machine's states under conjugation by the second commute.

It is introduced (in lesser generality, and with small variations) in [Sid05], and may be described as follows: one takes two copies of the stateset of m, one copy of the stateset of n, and, if necessary, an extra identity state.

The first copy of m fixes the alphabet $X \times Y$; its state $\tilde{s}$ has transitions to the identity except $\tilde{s}$ at $(x0, y0)$ and s at $(*, y0)$ for any other $*$. The second copy of m is also trivial except that, on input $(x, y0)$, its state s goes to state s' with output $(x', y0)$ whenever s originally went, on input x, to state s' with output x'. This copy of m therefore acts only in the X direction, on the subtree $(X \times \{y0\})^\infty$, on subtrees below vertices of the form $(x0, y0)^t(x, y0)$.

A state t in the copy of n maps the input (x, y) to (x, y') and proceeds to state t' if $y = y0$, and to the identity state otherwise, when on input y the original machine mapped state t to output t' and output y'.

```
 ────────────────────────────── Example ──────────────────────────────
 gap> m := TreeWreathProduct(AddingMachine(2),AddingMachine(3),1,1);
 AddingMachine(2)~AddingMachine(3)
```

```
gap> Display(last);
   |  1     2     3     4     5     6
---+-----+-----+-----+-----+-----+-----+
 a | c,2   c,3   a,1   c,5   c,6   c,4
 b | c,4   c,2   c,3   b,1   c,5   c,6
 c | c,1   c,2   c,3   c,4   c,5   c,6
 d | d,1   c,2   c,3   b,4   c,5   c,6
---+-----+-----+-----+-----+-----+-----+
```

3.5.9   SubFRMachine

▷ SubFRMachine(*machine1, machine2*)   (operation)
▷ SubFRMachine(*machine1, f*)   (operation)
   Returns: Either `fail` or an embedding of the states of *machine2* in the states of *machine1*.

   In its first form, this function attempts to locate a copy of *machine2* in *machine1*. If is succeeds, it returns a homomorphism from the stateset of *machine2* into the stateset of *machine1*; otherwise it returns `fail`.

   In its second form, this function attempts to construct a machine with stateset the source of *f*, that could be identified as a submachine of *machine1* via *f*.

```
─────────────────── Example ───────────────────
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ tau, mu ] )>
gap> tauinv := FRMachine([[[1],[]]],[(1,2)]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
gap> SubFRMachine(n,tauinv);
[ f1 ] -> [ tau^-1 ]
gap> SubFRMachine(n,last);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
```

3.5.10   ChangeFRMachineBasis

▷ ChangeFRMachineBasis(*m[, l][, p]*)   (attribute)
   Returns: An equivalent FR machine, in a new basis.

   This function constructs a new group FR machine, given a group FR machine *m* and, optionally, a list of states *l* (as elements of the free object `StateSet(m)`) and a permutation *p*, which defaults to the identity permutation.

   The new machine has the following transitions: if alphabet letter `a` is mapped to `b` by state `s` in *m*, leading to state `t`, then, in the new machine, the input letter `a^p` is mapped to `b^p` by state `s`, leading to state `l[a]^-1*t*l[b]`.

   The group generated by the new machine is isomorphic to the group generated by *m*. This command amounts to a change of basis of the associated bimodule (see [Nek05, Section 2.2]). It amounts to conjugation by the automorphism `c=FRElement("c",[l[1]*c,...,l[n]*c],[()],1)`.

   If the second argument is absent, this command attempts to choose a list that makes many entries of the recursion trivial.

```
─────────────────── Example ───────────────────
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);;
gap> Display(n);
 G   |    1          2
-----+--------+---------+
```

```
   tau | <id>,2      tau,1
    mu | <id>,2    mu^-1,1
  -----+--------+---------+
gap> nt := ChangeFRMachineBasis(n,GeneratorsOfFRMachine(n){[1,1]});;
gap> Display(nt);
   G  |     1                   2
  -----+--------+-------------------+
   tau | <id>,2                 tau,1
    mu | <id>,2   tau^-1*mu^-1*tau,1
  -----+--------+-------------------+
```

3.5.11 Minimized (FR machine)

▷ Minimized(*m*) (operation)

Returns: A minimized machine equivalent to *m*.

This function attempts to construct a machine equivalent to *m*, but with a stateset of smaller rank. Identical generators are collapsed to a single generator of the stateset; if *m* is a group or monoid machine then trivial generators are removed; if *m* is a group machine then mutually inverse generators are grouped. This function sets as `Correspondence(result)` a mapping between the stateset of *m* and the stateset of the result; see `Correspondence` (3.5.12).

```
————— Example —————
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);;
gap> m := FRMachine(["tauinv"],[[[1],[]]],[(1,2)]);;
gap> sum := n+m+n;
<FR machine with alphabet [ 1, 2 ] on Group( [ tau1, mu1, tauinv1, tau2, mu2 ] )>
gap> min := Minimized(sum);
<FR machine with alphabet [ 1, 2 ] on Group( [ tau1, mu1 ] )>
gap> Correspondence(min);
[ tau1, mu1, tauinv1, tau2, mu2 ] -> [ tau1, mu1, tau1^-1, tau1, mu1 ]
```

3.5.12 Correspondence (FR machine)

▷ Correspondence(*m*) (attribute)

Returns: A mapping between statesets of FR machines.

If a machine *m* was created as a minimized group/monoid/semigroup machine, then `Correspondence(m)` is a mapping between the stateset of the original machine and the stateset of *m*. See `Minimized` (3.5.11) for an example.

If *m* was created as a minimized Mealy machine, then `Correspondence(m)` is a list identifying, for each state of the original machine, a state of the new machine. If the original state is inaccessible, the corresponding list entry is unbound. See `Minimized` (5.2.2) for an example.

If *m* was created using `AsGroupFRMachine` (3.3.4), `AsMonoidFRMachine` (3.3.4), `AsSemigroupFRMachine` (3.3.4), or `AsMealyMachine` (5.2.18), then `Correspondence(m)` is a list or a homomorphism identifying for each generator of the original machine a generator, or word in the generators, of the new machine. It is a list if either the original or the final machine is a Mealy machine, and a homomorphism in other cases.

If *m* was created as a sum of two machines, then *m* has a mapping `Correspondence(m)[i]` between the stateset of machine i=1,2 and its own stateset. See `\+` (3.5.2) for an example.

Chapter 4

Functionally recursive elements

A *functionally recursive element* is given by a functionally recursive machine and an initial state q. Many functions for FR machines, which accept a state as an argument, apply to FR elements. In that case, no state is passed to the function.

The main function of FR elements is to serve as group/monoid/semigroup elements: they can be multiplied and divided, and they act naturally on sequences. They can also be tested for equality, and can be sorted.

FR elements are stored as free group/monoid/semigroup words. They are printed as <n|w>, where n is the degree of their alphabet.

Equality of FR elements is tested as follows. Given FR elements (m, q) and (m, r), we set up a "rewriting system" for m, which records a purported set of relations among states of m. We start by an empty rewriting system, and we always ensure that the rewriting system is reduced and shortlex-reducing. Then, to compare q and r, we first compare their activities. If they differ, the elements are distinct. Otherwise, we reduce q and r using the rewriting system. If the resulting words are graphically equal, then they are equal. Otherwise, we add the rule $q \rightarrow r$ or $r \rightarrow q$ to the rewriting system, and proceed recursively to compare coordinatewise the states of these reduced words. As a bonus, we keep the rewriting system to speed up future comparisons.

Efficient comparison requires lookup in sorted lists, aka "Sets". Given two FR elements x and y, we declare that $x < y$ if, for the shortlex-first sequence l such that Output(Transition(x,l)) and Output(Transition(y,l)) differ, the former is less than the latter (compared as lists). In fact, a single internal function compares x and y and returns $-1, 0, 1$ depending on whether $x < y$ or $x = y$ or $x > y$. It traverses, in breadth first fashion, the alphabet sequences, and stops either when provably $x = y$ or if different outputs appear.

4.1   Creators for `FRElements`

4.1.1   FRElementNC (family,free,listlist,list,assocword)

▷ FRElementNC(*fam, free, transitions, outputs, init*)   (operation)
    Returns: A new FR element.

This function constructs a new FR element, belonging to family *fam*. It has stateset the free group/semigroup/monoid *free*, and transitions described by *states* and *outputs*, and initial states *init*.

transitions is a list of lists; transitions[s][x] is a word in free, which is the state reached by the machine when started in state s and fed input x.

outputs is a list of lists; outputs[s][x] is a output letter of the machine when it receives input x in state s.

init is a word in free.

```
──────── Example ────────
gap> f := FreeGroup(2);
<free group on the generators [ f1, f2 ]>
gap> e := FRElementNC(FREFamily([1,2]),f,[[One(f),f.1],[One(f),f.2^-1]],
                      [[2,1],[2,1]],f.1);
<2|f1>
```

4.1.2 FRElement ([list,]list,list,list)

▷ FRElement([names,]transitions, outputs, init)   (operation)
▷ FRElement(free, transitions, outputs, init)   (operation)

Returns: A new FR element.

This function constructs a new FR element. It has stateset a free group/semigroup/monoid, structure described by transitions and outputs, and initial state init. If the stateset is not passed as argument free, then it is determined by transitions and outputs; it is a free group if all states are invertible, and a free monoid otherwise. In that case, names is an optional list; at position s it contains a string describing generator s.

transitions is a list of lists; transitions[s][x] is either an associative word, or a list of integers or FR elements describing the state reached by the machine when started in state s and fed input x. Positive integers indicate a generator, negative integers its inverse, the empty list in the identity state, and lists of length greater than one indicate a product of states. If an entry is an FR element, then its machine is incorporated into the newly constructed element.

outputs is a list; at position s it contains a permutation, a transformation, or a list of images, describing the activity of state s.

init is either an associative word, an integer, or a list of integers describing the inital state of the machine.

```
──────── Example ────────
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);
<2|tau>
gap> tau1 := FRElement(["tau1","tau"],[[[],[2]],[[],[2]]],[(),(1,2)],1);
<2|tau1>
gap> (tau/tau1)^2;
<2|tau1*tau2^-1*tau1*tau2^-1>
gap> IsOne(last);
true
```

```
──────── Example ────────
gap> f := FreeGroup("tau","tau1");
<free group on the generators [ tau, tau1 ]>
gap> tau := FRElement(f,[[One(f),f.1],[One(f),f.1]],[(1,2),()],f.1);
<2|tau>
gap> tau1 := FRElement(f,[[One(f),f.1],[One(f),f.1]],[(1,2),()],f.2);
<2|tau1>
gap> (tau/tau1)^2;
<2|tau1*tau2^-1*tau1*tau2^-1>
```

```
gap> IsOne(last);
true
gap> tauX := FRElement(f,[[One(f),f.1],[One(f),f.1]],[(1,2),()],1);;
gap> tauY := FRElement(f,[[One(f),f.1],[One(f),f.1]],[(1,2),()],f.1);;
gap> Size(Set([tau,tauX,tauY]));
1
```

4.1.3 FRElement (machine/element,list)

▷ FRElement(*m, q*) (operation)

Returns: A new FR element.

This function constructs a new FR element. If *m* is an FR machine, it creates the element (m,q) whose FRMachine is *m* and whose initial state is *q*.

If *m* is an FR element, this command creates an FR element with the same FR machine as *m*, and with initial state *q*.

```
——————————— Example ———————————
gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
<FR machine with alphabet [ 1 .. 2 ] on Group( [ a, b ] )>
gap> a := FRElement(m,1); b := FRElement(m,2);
<2|a>
<2|b>
gap> Comm(b,b^a);
<2|b^-1*a^-1*b^-1*a*b*a^-1*b*a>
gap> IsOne(last);
true
gap> last2=FRElement(m,[-2,-1,-2,1,2,-1,2,1]);
true
```

4.1.4 ComposeElement (elementcoll,perm)

▷ ComposeElement(*l, p*) (operation)

Returns: A new FR element.

This function constructs a new FR element. *l* is a list of FR elements, and *p* is a permutation, transformation or list. In that last case, the resulting element g satisfies `DecompositionOfFRElement(g)=[l,p]`.

If all arguments are Mealy elements, the result is a Mealy element. Otherwise, it is a Monoid-FRElement.

```
——————————— Example ———————————
gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);;
gap> a := FRElement(m,1); b := FRElement(m,2);
<2|a>
<2|b>
gap> ComposeElement([b^0,b],(1,2));
<2|f1>
gap> last=a;
true
gap> DecompositionOfFRElement(last2);
[ [ <2|identity ...>, <2|f5> ], [ 2, 1 ] ]
```

4.1.5 VertexElement

▷ VertexElement(*v, e*)   (operation)

   Returns: A new FR element.

   This function constructs a new FR element. *v* is either an integer or a list of integers, and represents a vertex. *e* is an FR element. The resulting element acts on the subtree below vertex *v* as *e* acts on the whole tree, and fixes all other subtrees.

```
————————————————— Example —————————————————
  gap> e := FRElement([[[],[]]],[(1,2)],[1]);
  <2|f1>
  gap> f := VertexElement(1,e);;
  gap> g := VertexElement(2,f);;
  gap> g = VertexElement([2,1],e);
  true
  gap> 1^e;
  2
  gap> [1,1]^f;
  [ 1, 2 ]
  gap> [2,1,1]^g;
  [ 2, 1, 2 ]
```

4.1.6 DiagonalElement

▷ DiagonalElement(*n, e*)   (operation)

   Returns: A new FR element.

   This function constructs a new FR element. *n* is either an integer or a list of integers, representing a sequence of operations to be performed on *e* starting from the last.

   `DiagonalElement(n,e)` is an element with trivial output, and with $e^{(-1)^i \, binomial(n,i)}$ in coordinate $i+1$ of the alphabet, assumed to be of the form `[1..d]`.

   In particular, `DiagonalElement(0,e)` is the same as `VertexElement(1,e)`; `DiagonalElement(1,e)` is the commutator of `VertexElement(1,e)` with any cycle mapping 1 to 2; and `DiagonalElement(-1,e)` has a transition to *e* at all inputs.

```
————————————————— Example —————————————————
  gap> e := FRElement([[[],[],[1]]],[(1,2,3)],[1]);
  <3|f1>
  gap> Display(e);
       |    1        2       3
  ----+--------+--------+------+
   f1 | <id>,2   <id>,3   f1,1
  ----+--------+--------+------+
  Initial state: f1
  gap> Display(DiagonalElement(0,e));
       |    1        2        3
  ----+--------+--------+--------+
   f1 |   f2,1    <id>,2   <id>,3
   f2 | <id>,2   <id>,3     f2,1
  ----+--------+--------+--------+
  Initial state: f1
  gap> Display(DiagonalElement(1,e));
       |    1        2         3
  ----+--------+---------+--------+
```

```
  f1 |   f2,1   f2^-1,2   <id>,3
  f2 | <id>,2   <id>,3    f2,1
 ----+--------+---------+--------+
Initial state: f1
gap> Display(DiagonalElement(2,e));
     |    1        2        3
 ----+--------+---------+------+
  f1 |   f2,1   f2^-2,2   f2,3
  f2 | <id>,2   <id>,3    f2,1
 ----+--------+---------+------+
Initial state: f1
gap> Display(DiagonalElement(-1,e));
     |    1        2        3
 ----+--------+--------+------+
  f1 |   f2,1     f2,2    f2,3
  f2 | <id>,2   <id>,3    f2,1
 ----+--------+--------+------+
Initial state: f1
gap> DiagonalElement(-1,e)=DiagonalElement(2,e);
true
gap> Display(DiagonalElement([0,-1],e));
  G  |    1        2        3
 ----+--------+--------+--------+
  f1 |   f2,1   <id>,2   <id>,3
  f2 |   f3,1    f3,2     f3,3
  f3 | <id>,2   <id>,3    f3,1
 ----+--------+--------+--------+
Initial state: f1
gap> Display(DiagonalElement([-1,0],e));
  G  |    1        2        3
 ----+--------+--------+--------+
  f1 |   f2,1     f2,2     f2,3
  f2 |   f3,1   <id>,2   <id>,3
  f3 | <id>,2   <id>,3    f3,1
 ----+--------+--------+--------+
Initial state: f1
```

4.1.7 AsGroupFRElement

▷ AsGroupFRElement(e)   (operation)
▷ AsMonoidFRElement(e)   (operation)
▷ AsSemigroupFRElement(e)   (operation)

Returns: An FR element isomorphic to *m*, with a free group/monoid/semigroup as stateset.

This function constructs, from the FR element *e*, an isomorphic FR element f with a free group/monoid/semigroup as stateset. *e* may be a Mealy, group, monoid or semigroup FR element.

```
————— Example —————
gap> e := AsGroupFRElement(FRElement(GuptaSidkiMachines(3),4));
<3|f1>
gap> Display(e);
  G  |    1        2        3
 ----+--------+---------+--------+
```

```
  f1 |   f2,1   f2^-1,2     f1,3
  f2 | <id>,2   <id>,3   <id>,1
 ----+--------+---------+--------+
Initial state: f1
gap> e=FRElement(GuptaSidkiMachines(3),4);
#I  \=: converting second argument to FR element
true
```

---- Example ----
```
gap> e := AsMonoidFRElement(FRElement(GuptaSidkiMachines(3),4));
<3|m1>
gap> Display(e);
  M  |    1        2        3
 ----+--------+--------+--------+
  m1 |   m2,1     m3,2     m1,3
  m2 | <id>,2   <id>,3   <id>,1
  m3 | <id>,3   <id>,1   <id>,2
 ----+--------+--------+--------+
Initial state: m1
gap> e=FRElement(GuptaSidkiMachines(3),4);
#I  \=: converting second argument to FR element
true
```

---- Example ----
```
gap> e := AsSemigroupFRElement(FRElement(GuptaSidkiMachines(3),4));
<3|s1>
gap> Display(e);
  S  |  1       2       3
 ----+------+------+------+
  s1 | s2,1   s3,2   s1,3
  s2 | s4,2   s4,3   s4,1
  s3 | s4,3   s4,1   s4,2
  s4 | s4,1   s4,2   s4,3
 ----+------+------+------+
Initial state: s1
gap> e=FRElement(GuptaSidkiMachines(3),4);
#I  \=: converting second argument to FR element
true
```

4.2   Operations and Attributes for `FRElement`s

4.2.1   Output (FR element)

▷ Output(*e*)   (operation)

   Returns: A transformation of *e*'s alphabet.

   This function returns the transformation of *e*'s alphabet, i.e. the action on strings of length 1 over the alphabet. This transformation is a permutation if `machine` is a group machine, and a transformation otherwise.

---- Example ----
```
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> Output(tau);
(1,2)
```

```
  zap := FRElement(["zap"],[[[],[1]]],[[1,1]],[1]);;
gap> Output(zap);
<1,1>
```

4.2.2 Activity

▷ Activity(*e[, l]*) (operation)
▷ ActivityInt(*e[, l]*) (operation)
▷ ActivityTransformation(*e[, l]*) (operation)
▷ ActivityPerm(*e[, l]*) (operation)

Returns: The transformation induced by *e* on the *l*th level of the tree.

This function returns the transformation induced by *e* on the *l*th level of the tree, i.e. on the strings of length *l* over *e*'s alphabet.

This set of strings is identified with the set $L = \{1, \ldots, d^l\}$ of integers, where the alphabet of *e* has *d* letters. Changes of the first letter of a string induce changes of a multiple of d^{l-1} on the position in *L*, while changes of the last letter of a string induce changes of 1 on the position in *L*.

In its first form, this command returns a permutation (for group elements) or a `Transformation` (**Reference: Transformation for an image list**) (for other elements). In the second form, it returns the unique integer `i` such that the transformation *e* acts on `[1..Length(AlphabetOfFRObject(e))^n]` as adding `i` in base `Length(alphabet(e))`, or `fail` if no such `i` exists. In the third form, it returns a GAP transformation. In the fourth form, it returns a permutation, or `fail` if *e* is not invertible.

```
────────────────────────── Example ──────────────────────────
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> Output(tau); PermList(last)=Activity(tau);
[ 2, 1 ]
true
gap> Activity(tau,2); ActivityInt(tau,2);
(1,3,2,4)
1
gap> Activity(tau,3); ActivityInt(tau,3);
(1,5,3,7,2,6,4,8)
1
gap> zap := FRElement(["zap"],[[[1],[]]],[[1,1]],[1]);
<2|zap>
gap> Output(zap);
[ 1, 1 ]
gap> Activity(zap,3);
<1,1,1,2,1,2,3,4>
```

4.2.3 Transition (FR element,input)

▷ Transition(*e, i*) (operation)

Returns: An element of *machine*'s stateset.

This function returns the state reached by *e* when fed input *i*. This input may be an alphabet letter or a sequence of alphabet letters.

```
────────────────────────── Example ──────────────────────────
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> Transition(tau,2);
tau
```

```
gap> Transition(tau,[2,2]);
tau
gap> Transition(tau^2,[2,2]);
tau
```

4.2.4 Transitions (FR element)

▷ Transitions(e)   (operation)

Returns: A list of elements of `machine`'s stateset.

This function returns the states reached by `e` when fed the alphabet as input.

```
————————————————————— Example —————————————————————
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> Transitions(tau);
[ <identity ...>, tau ]
gap> Transition(tau^2);
[ tau, tau ]
```

4.2.5 Portrait

▷ Portrait(e, l)   (operation)
▷ PortraitPerm(e, l)   (operation)
▷ PortraitTransformation(e, l)   (operation)
▷ PortraitInt(e, l)   (operation)

Returns: A nested list describing the action of `e`.

This function returns a sequence of $l + 1$ lists; the ith list in the sequence is an `i-1`-fold nested list. The entry at position $(x_1, \ldots, x_i)$ is the transformation of the alphabet induced by `e` under vertex $x_1 \ldots x_i$.

The difference between the commands is the following: Portrait returns transformations, PortraitPerm returns permutations, and and PortraitInt returns integers, the power of the cycle $x \mapsto x + 1$ that represents the transformation, as for the function ActivityInt (4.2.2).

```
————————————————————— Example —————————————————————
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> Portrait(tau,0);
[ <2,1> ]
gap> Portrait(tau,3);
[ <2,1>, [ <>, <2,1> ], [ [ <>, <> ], [ <>, <2,1> ] ],
  [ [ [ <>, <> ], [ <>, <> ] ], [ [ <>, <> ], [ <>, <2,1> ] ] ] ]
gap> PortraitPerm(tau,0);
[ (1,2) ]
gap> PortraitInt(tau,0);
[ 1 ]
gap> PortraitInt(tau,3);
[ 1 , [ 0 , 1 ],
  [ [ 0 , 0 ], [ 0 , 1 ] ],
  [ [ [ 0 , 0 ], [ 0 , 0 ] ], [ [ 0 , 0 ], [ 0 , 1 ] ] ] ]
```

4.2.6 DecompositionOfFRElement

▷ DecompositionOfFRElement(e*[, n]*)   (operation)

Returns: A list describing the action and transitions of `e`.

This function returns a list. The second coordinate is the action of `e` on its alphabet, see `Output` (4.2.1). The first coordinate is a list, containing in position *i* the FR element inducing the action of `e` on strings starting with *i*.

If a second argument `n` is supplied, the decomposition is iterated *n* times.

This FR element has same underlying machine as `e`, and initial state given by `Transition` (4.2.3).

```
——————————————— Example ———————————————
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> DecompositionOfFRElement(tau);
[ [ <2|identity ...>, <2|tau> ], [ 2, 1 ] ]
```

4.2.7 StateSet (FR element)

▷ StateSet(e)   (operation)

Returns: The set of states associated with `e`.

This function returns the stateset of `e`. If `e` is of Mealy type, this is the list of all states reached by `e`.

If `e` is of group/semigroup/monoid type, then this is the stateset of the underlying FR machine, and not the minimal set of states of `e`, which is computed with `States` (4.2.9).

```
——————————————— Example ———————————————
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> StateSet(tau);
<free group on the generators [ tau ]>
```

4.2.8 State

▷ State(e, v)   (operation)

Returns: An FR element describing the action of `e` at vertex `v`.

This function returns the FR element with same underlying machine as `e`, acting on the binary tree as `e` acts on the subtree below `v`.

`v` is either an integer or a list. This function returns an FR element, but otherwise is essentially a call to `Transition` (4.2.3).

```
——————————————— Example ———————————————
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> State(tau,2);
<2|tau>
gap> State(tau,[2,2]);
<2|tau>
gap> State(tau^2,[2,2]);
<2|tau>
```

4.2.9 States

▷ States(e)   (operation)

Returns: A list of FR elements describing the action of `e` on all subtrees.

This function calls repeatedly State (4.2.8) to compute all the states of `e`; it returns the smallest list of FRElements that is closed under the function State (4.2.8).

`e` may be either an FR element, or a list of FR elements. In the latter case, it amounts to computing the list of all states of all elements of the list `e`.

The ordering of the list is as follows. First come `e`, or all elements of `e`. Then come the states reached by `e` in one transition, ordered by the alphabet letter leading to them; then come those reached in two transitions, ordered lexicographically by the transition; etc.

Note that this function is not guaranteed to terminate. There is currently no mechanism that detects whether an FR element is finite state, so in fact this function terminates if and only if `e` is finite-state.

```
 ────────────────────────── Example ──────────────────────────
 gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);;
 gap> a := FRElement(m,1);; b := FRElement(m,2);;
 gap> States(a);
 [ <2|a>, <2|identity ...>, <2|b> ]
 gap> States(b);
 [ <2|b>, <2|identity ...>, <2|a> ]
 gap> States(a^2);
 [ <2|a^2>, <2|b>, <2|identity ...>, <2|a> ]
```

4.2.10 FixedStates

▷ FixedStates(`e`)   (operation)

Returns: A list of FR elements describing the action of `e` at fixed vertices.

This function calls repeatedly State (4.2.8) to compute all the states of `e` at non-trivial fixed vertices.

`e` may be either an FR element, or a list of FR elements. In the latter case, it amounts to computing the list of all states of all elements of the list `e`.

The ordering of the list is as follows. First come `e`, or all elements of `e`. Then come the states reached by `e` in one transition, ordered by the alphabet letter leading to them; then come those reached in two transitions, ordered lexicographically by the transition; etc.

Note that this function is not guaranteed to terminate, if `e` is not finite-state.

```
 ────────────────────────── Example ──────────────────────────
 gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);;
 gap> a := FRElement(m,1);; b := FRElement(m,2);;
 gap> FixedStates(a);
 [ ]
 gap> FixedStates(b);
 [ <2|identity ...>, <2|a> ]
 gap> FixedStates(a^2);
 [ <2|b>, <2|identity ...>, <2|a> ]
```

4.2.11 LimitStates

▷ LimitStates(`e`)   (operation)

Returns: A set of FR element describing the recurring actions of `e` on all subtrees.

This function computes the States (4.2.9) *S* of `e`, and then repeatedly removes elements that are not *recurrent*, i.e. that do not appear as states of elements of *S* on subtrees distinct from the entire tree; and then converts the result to a set.

As for `States` (4.2.9), `e` may be either an FR element, or a list of FR elements.

Note that this function is not guaranteed to terminate. It currently terminates if and only if `States` (4.2.9) terminates.

```
———————————————— Example ————————————————
gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);;
gap> a := FRElement(m,1);; b := FRElement(m,2);;
gap> LimitStates(a);
[ <2|identity ...>, <2|b>, <2|a> ]
gap> LimitStates(a^2);
[ <2|identity ...>, <2|b>, <2|a> ]
```

4.2.12 IsFiniteStateFRElement

▷ `IsFiniteStateFRElement(e)`   (property)

▷ `IsFiniteStateFRMachine(e)`   (property)

    Returns: `true` if `e` is a finite-state element.

    This function tests whether `e` is a finite-state element.

    When applied to a Mealy element, it returns `true`.

```
———————————————— Example ————————————————
gap> m := GuptaSidkiMachines(3);; Display(m);
   |  1      2      3
---+-----+-----+-----+
 a | a,1    a,2    a,3
 b | a,2    a,3    a,1
 c | a,3    a,1    a,2
 d | b,1    c,2    d,3
---+-----+-----+-----+
gap> Filtered(StateSet(m),i->IsFiniteStateFRElement(FRElement(m,i)));
[ 1, 2, 3, 4 ]
gap> IsFiniteStateFRMachine(m);
true
```

4.2.13 NucleusOfFRMachine

▷ `NucleusOfFRMachine(m)`   (operation)

▷ `Nucleus(m)`   (operation)

    Returns: The nucleus of the machine `m`.

    This function computes the *nucleus* of the machine `m`. It is the minimal set `N` of states such that, for every word `s` in the states of `m`, all states of `s` of at large enough depth belong to .

    It may also be characterized as the minimal set `N` of states that contains the limit states of `m` and is such that the limit states of `N*m` belong to `N`.

    The elements of the nucleus form the stateset of a Mealy machine; this machine is created by `NucleusMachine` (5.2.27).

    This command is not guaranteed to terminate; though it will, if the semigroup generated by `m` is contracting. If the minimal such `N` is infinite, this command either returns `K` or runs forever.

```
———————————————— Example ————————————————
gap> m := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);;
gap> NucleusOfFRMachine(m);
[ <2|identity ...>, <2|b>, <2|a> ]
```

```
gap> m := FRMachine(["a","b"],[[[],[1]],[[1],[2]]],[(1,2),()]);;
gap> NucleusOfFRMachine(m);
fail
```

4.2.14 InitialState

▷ InitialState(e)   (operation)

Returns: The initial state of an FR element.

This function returns the initial state of an FR element. It is an element of the stateset of the underlying FR machine of `e`.

```
————————————————— Example —————————————————
gap> n := FRElement(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)],[1,2]);
<2|tau*mu>
gap> InitialState(n);
tau*mu
gap> last in StateSet(n);
true
```

4.2.15 \^ (POW)

▷ \^(e, v)   (method)

Returns: The image of a vertex `v` under `e`.

This function accepts an FR element and a vertex `v`, which is either an integer or a list. It returns the image of `v` under the transformation `e`, in the same format (integer/list) as `v`.

The list `v` can be a periodic list (see `PeriodicList` (11.2.2)). In that case, the result is again a periodic list. The computation will succeed only if the states along the period are again periodic.

```
————————————————— Example —————————————————
gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
gap> 1^tau;
2
gap> [1,1]^tau;
[ 2, 1 ]
gap> [2,2,2]^tau;
[ 1, 1, 1 ]
gap List([0..5],i->PeriodicList([],[2])^(tau^i));
[ [/ 2 ], [/ 1 ], [ 2, / 1 ], [ 1, 2, / 1 ], [ 2, 2, / 1 ],
  [ 1, 1, 2, / 1 ] ]
```

4.2.16 * (PROD)

▷ *(m, n)   (method)

Returns: The product of the two FR elements `m` and `n`.

This function returns a new FR element, which is the product of the FR elements `m` and `n`.

In case `m` and `n` have the same underlying machine, this is the machine of the result. In case the machine of `n` embeds in the machine of `m` (see `SubFRMachine` (3.5.9)), the machine of the product is the machine of `m`. In case the machine of `m` embeds in the machine of `n`, the machine of the product is the machine of `n`. Otherwise the machine of the product is the product of the machines of `m` and `n` (See * (3.5.3)).

```
 ──────────────── Example ────────────────
 gap> tau := FRElement(["tau"],[[[],[1]]],[(1,2)],[1]);;
 gap> tau*tau; tau^2;
 <2|tau^2>
 <2|tau^2>
 gap> [2,2,2]^(tau^2);
 [ 2, 1, 1 ]
```

4.2.17   \[\] (ELMLIST)

▷ \[\](*m, i*)   (method)
▷ \{\}(*m, l*)   (method)

Returns: A [list of] FR element[s] with initial state *i*.

These are respectively synonyms for FRElement(m,i) and List(l,s->FRElement(m,s)). The argument *m* must be an FR machine, *i* must be a positive integer, and *l* must be a list.

Chapter 5

Mealy machines and elements

Mealy machines form a special class of FR machines. They have as stateset a finite set, as opposed to a free group/monoid/semigroup. All commands available for FR machines are also available for Mealy machines, but the latter have added functionality.

There are currently two types of Mealy machines; one has as stateset an interval of integers of the form [1..m] and as alphabet a set of integers; the other has an arbitrary domain as stateset and alphabet. Almost no functionality is implemented for the latter type, but there is a function converting it to the former type (see AsMealyMachine (5.2.18)).

The internal representation of a Mealy machine of the first kind is quite different from that of FR machines. The alphabet is assumed to be an interval [1..n], and the stateset is assumed to be an interval [1..m]. The transitions are stored as a $m \times n$ matrix, and the outputs are stored in a list of length m, consisting of permutations or transformations.

Mealy machines have additional properties, in particular they can act on periodic sequences (see PeriodicList (11.2.2)). For example, the periodic sequence PeriodicList([1],[1,2]) describes the infinite ray [1,1,2,1,2,..] in the tree. In principle, Mealy machines could act on sequences accepted by an automaton, although this is not yet implemented.

Mealy elements are Mealy machines with an initial state. For efficiency reasons, Mealy elements are always minimized, and their states are ordered in a canonical top-down, left-to-right order of traversal of the tree. In particular, their initial state is always 1. In this implementation, multiplication of Mealy elements is slower than multiplication of group FR elements, while comparison of Mealy elements is faster than comparison of group FR elements. In practise, it is better to work with Mealy elements as often as possible.

Products of Mealy machines behave in the same way as products of general FR machines, see 3.2. The only difference is that now the sum and products of statesets are distinct; the sum of statesets being their disjoint union, and their product being their cartesian product.

Sometimes one would like to know how a Mealy element was obtained as a word in Mealy elements. This is possible within the representation IsFRMealyElement (10.2.12), which combines the representations IsMealyElement (10.2.4) and IsFRElement (10.2.11). On top of usual FR elements, they have an attribute UnderlyingMealyMachine, which is used for faster comparison of elements, and computation of the action.

Therefore, if L is a list of FR elements, the call List(L,UnderlyingElement);; will set these attributes, and all calculations made with elements of L will use and propagate the attributes. FR-Mealy elements are displayed in the form <d|w|n>, where d is the degree of the alphabet, w is a word in the stateset, and n is the number of states of the underlying Mealy element.

5.1 Creators for `MealyMachine`s and `MealyElement`s

5.1.1 MealyMachine ([list,]listlist,list)

▷ MealyMachine(*[alphabet,]transitions, output*)   (operation)
▷ MealyElement(*[alphabet,]transitions, output, init*)   (operation)

Returns: A new Mealy machine/element.

This function constructs a new Mealy machine or element, of integer type.

transitions is a list of lists; `transitions[s][x]` is an integer, which is the state reached by the machine when started in state *s* and fed input *x*.

output is a list; at position *s* it contains a permutation, a transformation describing the activity of state *s*, or a list describing the images of the transformation.

alphabet is an optional domain given as first argument; When present, it is assumed to be a finite domain, mapped bijectively to `[1..n]` by its enumerator. The indices `"[s]"` above are then understood with respect to this enumeration.

init is an integer describing the initial state the newly created Mealy element should be in.

```
───────────────────────── Example ─────────────────────────
gap> b := MealyMachine([[3,2],[3,1],[3,3]],[(1,2),(),()]);
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> Display(b);
   | 1      2
---+-----+-----+
 a | c,2    b,1
 b | c,1    a,2
 c | c,1    c,2
---+-----+-----+
gap> n := MealyMachine(Domain([11,12]),[[3,2],[3,1],[3,3]],[(1,2),(),()]);
<Mealy machine on alphabet [ 11, 12 ] with states [ 1 .. 3 ]>
gap> Display(n);
   | 11      12
---+------+------+
 a | c,12    b,11
 b | c,11    a,12
 c | c,11    c,12
---+------+------+
```

```
───────────────────────── Example ─────────────────────────
gap> tau := MealyElement([[2,1],[2,2]],[(1,2),()],1);
<Mealy machine on alphabet [ 1, 2 ] with 2 states, initial state 1>
gap> Display(tau);
   | 1      2
---+-----+-----+
 a | b,2    a,1
 b | b,1    b,2
---+-----+-----+
Initial state:  a
gap> [1,1]^tau; [[1]]^tau; [[2]]^tau;
[ 2, 1 ]
[ 2, [ 1 ] ]
[ [ 1 ] ]
```

5.1.2   MealyMachine (domain,domain,function,function)

▷ MealyMachine(*stateset, alphabet, transitions, output*)   (operation)
▷ MealyElement(*stateset, alphabet, transitions, output, init*)   (operation)
    Returns: A new Mealy machine/element.
    This function constructs a new Mealy machine or element, of domain type.
    stateset and *alphabet* are domains; they are not necessarily finite.
    transitions is a function; it takes as arguments a state and an alphabet letter, and returns a state.
    output is either a function, accepting as arguments a state and a letter, and returning a letter.
    init is an element of *stateset* describing the initial state the newly created Mealy element should be in.

```
                            ____ Example ____
  gap> g := Group((1,2));; n := MealyMachine(g,g,\*,\*);
  <Mealy machine on alphabet [ (), (1,2) ] with states Group( [ (1,2) ] )>
  gap> [(1,2),()]^FRElement(n,());
  [ (1,2), (1,2) ]
  gap> a := MealyElement(g,g,\*,\*,());
  <Mealy machine on alphabet [ (), (1,2) ] with states Group(
  [ (1,2) ] ), initial state ()>
  gap> [(1,2),()]^a;
  [ (1,2), (1,2) ]
```

5.1.3   MealyMachineNC (family,listlist,list)

▷ MealyMachineNC(*fam, transitions, output*)   (operation)
▷ MealyElementNC(*fam, transitions, output, init*)   (operation)
    Returns: A new Mealy machine/element.
    This function constructs a new Mealy machine or element, of integer type. No tests are performed to check that the arguments contain values within bounds, or even of the right type (beyond the simple checking performed by GAP's method selection algorithms). In particular, Mealy elements are always assumed to be minimized, but these functions leave this task to the user.
    fam is the family to which the newly created Mealy machine will belong.
    transitions is a list of lists; *transitions[s][x]* is an integer, which is the state reached by the machine when started in state *s* and fed input *x*.
    output is a list; at position *s* it contains a permutation or a transformation describing the activity of state *s*.
    init is an integer describing the initial state the newly created Mealy element should be in.

```
                            ____ Example ____
  gap> taum := MealyMachine([[2,1],[2,2]],[(1,2),()]);
  <Mealy machine on alphabet [ 1, 2 ] with 2 states>
  gap> tauminv := MealyMachineNC(FamilyObj(taum),[[1,2],[2,2]],[(1,2),()]);
  <Mealy machine on alphabet [ 1, 2 ] with 2 states>
  gap> tau := MealyElement([[2,1],[2,2]],[(1,2),()],1);
  <Mealy machine on alphabet [ 1, 2 ] with 2 states, initial state 1>
  gap> tauinv := MealyElementNC(FamilyObj(n),[[1,2],[2,2]],[(1,2),()],1);
  <Mealy machine on alphabet [ 1, 2 ] with 2 states, initial state 1>
  gap> tau=FRElement(taum,1); tauinv=FRElement(tauminv,1);
  true
  true
```

```
 gap> IsOne(tau*tauinv);
 true
```

5.1.4 AllMealyMachines

▷ AllMealyMachines(m, n[, filters])   (function)

Returns: A list of all Mealy machines with specified properties.

This function constructs all Mealy machines with alphabet [1..m], stateset [1..n] and specified properties.

These properties are specified as additional arguments. They can include IsInvertible (10.2.15), IsReversible (5.2.4), IsBireversible (5.2.7), and IsMinimized (5.2.5) to specify that the machines should have that property.

A group/monoid/semigroup p may also be passed as argument; this specifies the allowable vertex transformations of the machines. The property IsTransitive requires that the state-closed group/monoid/semigroup of the machine act transitively on its alphabet, and IsSurjective requires that its VertexTransformationsFRMachine (5.2.15) be precisely equal to p.

The argument EquivalenceClasses returns one isomorphism class of Mealy machine, under the permutations of the stateset and alphabet.

The argument InverseClasses returns one isomorphism class of Mealy machine under inversion of the stateset.

The following example constructs the two Mealy machines AleshinMachine (9.1.15) and BabyAleshinMachine (9.1.16):

```
 ───────────────────────── Example ─────────────────────────
 gap> l := AllMealyMachines(2,3,IsBireversible,IsSurjective,EquivalenceClasses);;
 gap> Length(l);
 20
 gap> Filtered(l,x->VertexTransformationsFRMachine(DualMachine(x))=SymmetricGroup(3)
 >                     and Size(StateSet(Minimized(x)))=3);
 [ <Mealy machine on alphabet [ 1, 2 ] with 3 states>,
   <Mealy machine on alphabet [ 1, 2 ] with 3 states> ]
 gap> Display(last[1]);
    | 1     2
 ---+-----+-----+
  a | a,1   b,2
  b | c,2   c,1
  c | b,1   a,2
 ---+-----+-----+
 gap> Display(last[2]);
    | 1     2
 ---+-----+-----+
  a | a,2   b,1
  b | c,1   c,2
  c | b,2   a,1
 ---+-----+-----+
```

5.2 Operations and Attributes for `MealyMachine`s and `MealyElement`s

5.2.1 Draw

▷ Draw(*m[, filename]*)   (operation)

   This function creates a graph description of the Mealy machine/element *m*. If a second argument `filename` is present, the graph is saved, in `dot` format, under that filename; otherwise it is converted to Postscript using the program `dot` from the graphviz package, and is displayed in a separate X window using the program display or rsvg-view. This works on UNIX systems.

   It is assumed, but not checked, that graphviz and display/rsvg-view are properly installed on the system. The option `usesvg` requests the use of rsvg-view; by default, display is used.

   A circle is displayed for every state of *m*, and there is an edge for every transition in *m*. It has label of the form x/y, where x is the input symbol and y is the corresponding output. Edges are coloured according to the input symbol, in the order "red", "blue", "green", "gray", "yellow", "cyan", "orange", "purple". If *m* has an initial state, it is indicated as a doubly circled state.

   If *m* is a FR machine, `Draw` first attempts to convert it to a Mealy machine (see `AsMealyMachine` (5.2.18)).

   The optional value "detach" detaches the drawing subprocess after it is started, in the syntax `Draw(M:detach)`.

   It is assumed that graphviz and display/rsvg-view are properly installed on the system. The option `usesvg` requests the use of rsvg-view; by default, display is used.

   For example, the command `Draw(NucleusMachine(BasilicaGroup));` produces (in a new
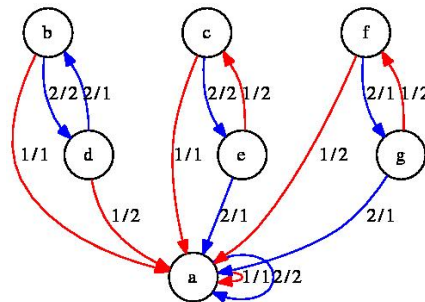
window) the following picture:

5.2.2 Minimized (Mealy machine)

▷ Minimized(*m*)   (operation)

   Returns: A minimized machine equivalent to *m*.

   This function contructs the minimized Mealy machine `r` corresponding to *m*, by identifying isomorphic states; and, if *m* is initial, by removing inaccessible states.

   If *m* is initial, the minimized automaton is such that its states are numbered first by distance to the initial state, and then lexicographically by input letter. (in particular, the initial state is 1). This makes comparison of minimized automata efficient.

   Furthermore, `Correspondence(r)` is a list describing, for each (accessible) state of *m*, its corresponding state in `r`; see `Correspondence` (3.5.12).

```
─────────────────────── Example ───────────────────────
  gap> GrigorchukMachine := MealyMachine([[2,3],[4,4],[2,5],[4,4],[4,1]],
                                   [(),(1,2),(),(),()]);
  <Mealy machine on alphabet [ 1, 2 ] with 5 states>
```

```
gap> g2 := GrigorchukMachine^2;
<Mealy machine on alphabet [ 1, 2 ] with 25 states>
gap> Minimized(g2);
<Mealy machine on alphabet [ 1, 2 ] with 11 states, minimized>
gap> Correspondence(last);
[ 2, 1, 4, 11, 9, 1, 2, 5, 7, 6, 4, 3, 2, 9, 11, 11, 10, 9, 2, 4, 9, 8, 11, 4, 2 ]
gap> e := FRElement(g2,11);
<Mealy element on alphabet [ 1, 2 ] with 25 states, initial state 11>
gap> Minimized(e);
<Mealy element on alphabet [ 1, 2 ] with 5 states, initial state 1, minimized>
gap> Correspondence(last);
[ 3, 2, 1, 4, 5, 2, 3,,,, 1,, 3, 5, 4, 4,, 5, 3, 1, 5,, 4, 1, 3 ]
```

5.2.3 DualMachine

▷ DualMachine(*m*) (operation)

    Returns: The dual Mealy machine of *m*.

    This function constructs the *dual* machine of *m*, i.e. the machine with stateset the alphabet of *m*, with alphabet the stateset of *m*, and similarly with transitions and output switched.

```
 ─────────────────────────────── Example ───────────────────────────────
 gap> b := MealyMachine([[3,2],[3,1],[3,3]],[(1,2),(),()]);
 <Mealy machine on alphabet [ 1, 2 ] with 3 states>
 gap> d := DualMachine(b)^4;
 <Mealy machine on alphabet [ 1, 2, 3 ] with 16 states>
 gap> Draw(d); # action on 2^4 points
 gap> DualMachine(d);
 <Mealy machine on alphabet [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
  ] with 3 states>
 gap> Output(last,1)=Activity(FRElement(b,1),4);
 true
```

5.2.4 IsReversible

▷ IsReversible(*m*) (property)

    Returns: true if *m* is a reversible Mealy machine.

    This function tests whether *m* is *reversible*, i.e. whether the DualMachine (5.2.3) of *m* is invertible. See [MNS00] for more details.

```
 ─────────────────────────────── Example ───────────────────────────────
 gap> IsReversible(MealyMachine([[1,2],[2,2]],[(1,2),()]));
 false
 gap> IsReversible(MealyMachine([[1,2],[2,1]],[(),(1,2)]));
```

5.2.5 IsMinimized

▷ IsMinimized(*m*) (property)

    Returns: true if *m* is a minimized Mealy machine.

    This function tests whether *m* is *minimized*, i.e. whether nono of its states can be removed or coalesced. All Mealy elements are automatically minimized.

```
 ───────────────────────────── Example ─────────────────────────────
  gap> AllMealyMachines(2, 2, IsBireversible,EquivalenceClasses);
  [ <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states>,
    <Mealy machine on alphabet [ 1, 2 ] with 2 states> ]
  gap> List(last,IsMinimized);
  [ false, true, false, false, false, false, true, false ]
```

5.2.6 AlphabetInvolution

▷ AlphabetInvolution(*m*)   (attribute)

Returns: A list giving, for each alphabet letter, its inverse.

If *m* is a bireversible machine, it may happen that the stateset of the dual of *m* (see DualMachine (5.2.3)) is closed under taking inverses. If this happens, then this list records the mapping from an alphabet letter of *m* to its inverse.

```
 ───────────────────────────── Example ─────────────────────────────
  gap> m := GammaPQMachine(3,5);; AlphabetOfFRObject(m);
  [ 1 .. 6 ]
  gap> IsBireversible(m); AlphabetInvolution(GammaPQMachine(3,5));
  true
  [ 6, 5, 4, 3, 2, 1 ]
```

5.2.7 IsBireversible

▷ IsBireversible(*m*)   (property)

Returns: true if *m* is a bireversible Mealy machine.

This function tests whether *m* is *bireversible*, i.e. whether all eight machines obtained from *m* using DualMachine (5.2.3) and Inverse are well-defined. See [MNS00] for more details.

```
 ───────────────────────────── Example ─────────────────────────────
  gap> IsBireversible(MealyMachine([[1,2],[2,1]],[(),(1,2)]));
  false
  gap> IsBireversible(MealyMachine([[1,1],[2,2]],[(),(1,2)]));
  true
```

5.2.8 StateGrowth

▷ StateGrowth(*m[, x]*)   (operation)

Returns: The state growth of the Mealy machine or element *m*.

This function computes, as a rational function, the power series in *x* whose coefficient of degree *n* is the number of non-trivial states at level *n* of the tree.

If *x* is absent, it is assumed to be Indeterminate(Rationals).

If *m* is a Mealy machine, this function is computed with respect to all possible starting states. If *m* is a Mealy element, this function is computed with respect to the initial state of *m*.

```
─────────────────────── Example ───────────────────────
gap> b := MealyMachine([[3,2],[3,1],[3,3]],[(1,2),(),()]);
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> StateGrowth(b,Indeterminate(Rationals));
(2)/(-x_1+1)
gap> StateGrowth(FRElement(b,1),Indeterminate(Rationals));
(1)/(-x_1+1)
```

5.2.9 Degree (FR element)

▷ Degree(*m*)   (operation)

▷ DegreeOfFRMachine(*m*)   (operation)

▷ DegreeOfFRElement(*m*)   (operation)

Returns: The growth degree of the Mealy machine or element *m*.

This function computes the order of the pole at $x = 1$ of StateGrowth(m,x), in case its denominator is a product of cyclotomics; and returns infinity otherwise.

This attribute of Mealy machines was studied inter alia in [Sid00].

```
─────────────────────── Example ───────────────────────
gap> m := MealyMachine([[2,1],[3,2],[3,3]],[(),(1,2),()]);
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> StateGrowth(m,Indeterminate(Rationals));
(-x_1+2)/(x_1^2-2*x_1+1)
gap> List(StateSet(m),i->Degree(FRElement(m,i)));
[ 2, 1, -1 ]
gap> a := MealyMachine(Group((1,2)),Group((1,2)),\*,\*);
<Mealy machine on alphabet [ (), (1,2) ] with states Group( [ (1,2) ] )>
gap> Degree(a);
infinity
```

5.2.10 IsFinitaryFRElement

▷ IsFinitaryFRElement(*e*)   (property)

▷ IsFinitaryFRMachine(*e*)   (property)

Returns: true if *e* is a finitary element.

This function tests whether *e* is a finitary element. These are by definition the elements of growth degree at most 0.

When applied to a Mealy machine, it returns true if all states of *e* are finitary.

```
─────────────────────── Example ───────────────────────
gap> m := GuptaSidkiMachines(3);; Display(m);
   | 1     2     3
---+-----+-----+-----+
 a | a,1   a,2   a,3
 b | a,2   a,3   a,1
 c | a,3   a,1   a,2
 d | b,1   c,2   d,3
---+-----+-----+-----+
gap> Filtered(StateSet(m),i->IsFinitaryFRElement(FRElement(m,i)));
[ 1, 2, 3 ]
gap> IsFinitaryFRElement(m);
false
```

5.2.11 Depth (FR element)

▷ Depth(*m*)   (attribute)
▷ DepthOfFRMachine(*m*)   (attribute)
▷ DepthOfFRElement(*m*)   (attribute)

    Returns: The depth of the finitary Mealy machine or element `m`.

    This function computes the maximal level at which the `m` has an non-trivial state. In particular the identity has depth 0, and FR elements acting only at the root vertex have depth 1. The value `infinity` is returned if `m` is not finitary (see `IsFinitaryFRElement` (5.2.10)).

```
——————————————————— Example ———————————————————
gap> m := MealyMachine([[2,1],[3,3],[4,4],[4,4]],[(),(),(1,2),()]);
<Mealy machine on alphabet [ 1, 2 ] with 4 states>
gap> DepthOfFRMachine(m);
infinity
gap> List(StateSet(m),i->DepthOfFRElement(FRElement(m,i)));
[ infinity, 2, 1, 0 ]
```

5.2.12 IsBoundedFRElement

▷ IsBoundedFRElement(*e*)   (property)
▷ IsBoundedFRMachine(*e*)   (property)

    Returns: `true` if `e` is a finitary element.

    This function tests whether `e` is a bounded element. These are by definition the elements of growth degree at most 1.

    When applied to a Mealy machine, it returns `true` if all states of `e` are bounded.

```
——————————————————— Example ———————————————————
gap> m := GuptaSidkiMachines(3);; Display(m);
   |  1     2     3
---+-----+-----+-----+
 a | a,1   a,2   a,3
 b | a,2   a,3   a,1
 c | a,3   a,1   a,2
 d | b,1   c,2   d,3
---+-----+-----+-----+
gap> Filtered(StateSet(m),i->IsBoundedFRElement(FRElement(m,i)));
[ 1, 2, 3, 4 ]
gap> IsBoundedFRMachine(m);
true
```

5.2.13 IsPolynomialGrowthFRElement

▷ IsPolynomialGrowthFRElement(*e*)   (property)
▷ IsPolynomialGrowthFRMachine(*e*)   (property)

    Returns: `true` if `e` is an element of polynomial growth.

    This function tests whether `e` is a polynomial element. These are by definition the elements of polynomial growth degree.

    When applied to a Mealy machine, it returns `true` if all states of `e` are of polynomial growth.

```
——————————————————— Example ———————————————————
gap> m := GuptaSidkiMachines(3);; Display(m);
   |  1     2     3
```

```
 ---+-----+-----+-----+
  a | a,1   a,2   a,3
  b | a,2   a,3   a,1
  c | a,3   a,1   a,2
  d | b,1   c,2   d,3
 ---+-----+-----+-----+
gap> Filtered(StateSet(m),i->IsPolynomialGrowthFRElement(FRElement(m,i)));
[ 1, 2, 3, 4 ]
gap> IsPolynomialGrowthFRMachine(m);
true
```

5.2.14 Signatures

▷ Signatures(e)   (operation)

Returns: A list describing the product of the activities on each level.

This function computes the product of the activities of e on each level, and returns a periodic list describing it (see PeriodicList (11.2.2)).

The entries pi are permutations, and their values are meaningful only when projected in the abelianization of VertexTransformationsFRElement(e).

```
 ———————————————————————————— Example ————————————————————————————
gap> Signatures(GrigorchukGroup.1);
[ (1,2), / () ]
gap> Signatures(GrigorchukGroup.2);
[/ (), (1,2), (1,2) ]
gap> last[50];
(1,2)
gap> Signatures(AddingMachine(3)[2]);
[/ (1,2,3) ]
```

5.2.15 VertexTransformationsFRMachine

▷ VertexTransformationsFRMachine(m)   (operation)
▷ VertexTransformationsFRElement(e)   (operation)

Returns: The group/monoid generated by all vertex transformations of states of m.

The first function computes the finite permutation group / transformation monoid generated by all outputs of states of m.

The second command is a short-hand for VertexTransformationsFRMachine(UnderlyingFRMachine(e)).

```
 ———————————————————————————— Example ————————————————————————————
gap> m := MealyMachine([[1,3,2],[3,2,1],[2,1,3]],[(2,3),(1,3),(1,2)]);
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> VertexTransformationsFRMachine(m);
Group([ (2,3), (1,3), (1,2) ])
```

5.2.16 FixedRay (FR element)

▷ FixedRay(e)   (operation)

Returns: The lexicographically first ray fixed by e.

This function computes the lexicographically first infinite sequence that is fixed by the FR element e, and returns it as a periodic list (see PeriodicList (11.2.2)). It returns fail if no such ray exists.

```
──────── Example ────────
gap> m := MealyMachine([[1,3,2],[3,2,1],[2,1,3]],[(2,3),(1,3),(1,2)]);
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> FixedRay(FRElement(m,1));
[/ 1 ]
gap> last^FRElement(m,1);
[/ 1 ]
gap> FixedRay(FRElement(m,[1,2]));
fail
```

5.2.17 IsLevelTransitiveFRElement

▷ IsLevelTransitiveFRElement(*e*)   (property)

Returns: true if *e* acts transitively on each level of the tree.

This function tests whether *e* acts transitively on each level of the tree. It is implemented only if `VertexTransformationsFRElement(e)` is abelian.

This function is used as a simple test to detect whether an element has infinite order: if *e* has a fixed vertex *v* such that the `State(e,v)` is level-transitive, then *e* has infinite order.

This function can be abbreviated as `IsLevelTransitive`.

```
──────── Example ────────
gap> m := AddingMachine(3);; Display(m);
   |   1      2      3
---+-----+-----+-----+
 a | a,1    a,2    a,3
 b | a,2    a,3    b,1
---+-----+-----+-----+
Initial state:  b
gap> IsLevelTransitiveFRElement(m);
true
gap> IsLevelTransitiveFRElement(Product(UnderlyingFRMachine(GrigorchukOverGroup){[2..5]}));
true
```

5.2.18 AsMealyMachine (FR machine)

▷ AsMealyMachine(*m*)   (attribute)

Returns: A Mealy machine isomorphic to *m*.

This function constructs a Mealy machine r, which is as close as possible to the FR machine *m*. Furthermore, `Correspondence(r)` is a list identifying, for every generator of the stateset of *m*, a corresponding state in the new Mealy machine; see `Correspondence` (3.5.12).

m may be a group/monoid/semigroup FR machine, or a Mealy machine; in which case the result is returned unchanged.

In particular, FRElement(m,s) and FRElement(AsMealyMachine(m),s) return the same tree automorphism, for any FR machine m and any state s.

This function is not guaranteed to return; if *m* does not have finite states, then it will loop forever.

```
──────── Example ────────
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);
<FR machine with alphabet [ 1 .. 2 ] on Group( [ tau, mu ] )>
gap> Display(n);
     |     1            2
```

```
-----+--------+---------+
 tau | <id>,2     tau,1
  mu | <id>,2   mu^-1,1
-----+--------+---------+
gap> AsMealyMachine(n);
<Mealy machine on alphabet [ 1, 2 ] with 4 states>
gap> Display(last);
   |  1     2
---+-----+-----+
 a | c,2   a,1
 b | c,2   d,1
 c | c,1   c,2
 d | b,2   c,1
---+-----+-----+
gap> Correspondence(last);
[ 1, 2 ]
```

5.2.19 AsMealyMachine (List)

▷ AsMealyMachine(*l*)   (attribute)

Returns: A Mealy machine constructed out of the FR elements in *l*.

This function constructs a Mealy machine r, with states *l* (which must be a state-closed set). Its outputs are the outputs of its elements, and its transitions are the transitions of its elements; in particular, FRElement(r,i) is equal to l[i] as an FR element.

Correspondence(r) records the argument *l*.

This function returns fail if *l* is not state-closed.

```
————————————————————— Example —————————————————————
gap>  mu := FRElement([[[],[-1]]],[(1,2)],[1]);
<2|f1>
gap>
gap> States(mu);
[ <2|f1>, <2|identity ...>, <2|f1^-1> ]
gap> AsMealyMachine(last);
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> Display(last);
   |  1     2
---+-----+-----+
 a | b,2   c,1
 b | b,1   b,2
 c | a,2   b,1
---+-----+-----+
```

5.2.20 AsMealyElement

▷ AsMealyElement(*m*)   (attribute)

Returns: A Mealy element isomorphic to *m*.

This function constructs a Mealy element, which induces the same tree automorphism as the FR element *m*.

m may be a group/monoid/semigroup FR element, or a Mealy element; in which case the result is returned unchanged.

This function is not guaranteed to return; if *m* does not have finite states, then it will loop forever.

```
──────────────── Example ────────────────
gap> mu := FRElement([[[],[-1]]],[(1,2)],[1]);
<2|f1>
gap> AsMealyElement(mu);
<Mealy machine on alphabet [ 1, 2 ] with 3 states, initial state 1>
gap> [[2,1]]^last;
[ [ 1, 2 ] ]
gap> [2,1,2,1]^mu;
[ 1, 2, 1, 2 ]
```

5.2.21 AsIntMealyMachine

▷ AsIntMealyMachine(*m*)   (attribute)

▷ AsIntMealyElement(*m*)   (attribute)

   Returns: A Mealy machine in integer format, isomorphic to *m*.

   This function constructs a Mealy machine r, which has similar behaviour as *m* while having state-set [1..n] for some natural n. Most FR commands operate efficiently only on Mealy machines of this type.

   This function is not guaranteed to return; if *m* does not have finite states, then it will loop forever.

```
──────────────── Example ────────────────
gap> g := Group((1,2));; n := MealyMachine(g,g,\*,\*);
<Mealy machine on alphabet [ (), (1,2) ] with states Group( [ (1,2) ] )>
gap> Display(n);
       |      ()            (1,2)
-------+-------------+-------------+
   () |    (),()        (1,2),(1,2)
 (1,2) | (1,2),(1,2)      (),()
-------+-------------+-------------+
gap> AsIntMealyMachine(n);
<Mealy machine on alphabet [ 1, 2 ] with 2 states>
gap> Display(last);
   | 1     2
---+-----+-----+
 a | a,1   b,2
 b | b,2   a,1
---+-----+-----+
gap> Correspondence(last);
[ 1, 2 ]
```

5.2.22 TopElement

▷ TopElement(*p[, n]*)   (attribute)

   Returns: A Mealy machine in integer format, acting on the first symbol of sequences.

   This function constructs a Mealy machine r, which acts as *p* on the first letter of sequences and fixes the other letters. The argument *n* is the size of the alphabet of r; if it is ommitted, then it is assumed to be the degree of the transformation *p*, or the largest moved point of the permutation or trans *p*.

```
                                ──── Example ────
  gap> a := TopElement((1,2));
  <Mealy element on alphabet [ 1, 2 ] with 2 states>
  gap> last=GrigorchukGroup.1;
  true
  gap> a := TopElement((1,2),3);
  <Mealy element on alphabet [ 1, 2, 3 ] with 2 states>
  gap> last in GuptaSidkiGroup;
  false
```

5.2.23   ConfinalityClasses

▷ ConfinalityClasses(*e*)   (attribute)

▷ IsWeaklyFinitaryFRElement(*e*)   (attribute)

   Returns: A list describing the non-trivial confinality classes of *e*.

   If *e* is a bounded element (see IsBoundedFRElement (5.2.12)), there are finitely many infinite sequences that have confinality class larger that one; i.e. ultimately periodic sequences that are mapped by *e* to a sequence with different period. This function returns a list of equivalence classes of periodic lists, see PeriodicList (11.2.2), which are related under *e*.

   By definition, an element is *weakly finitary* if it has no non-singleton confinality classes.

```
                                ──── Example ────
  gap> g := FRGroup("t=<,,t>(2,3)","u=<u,,>(1,2)","v=<u,t,>");;
  gap> ConfinalityClasses(g.1);
  [ {PeriodicList([  ],[ 2 ])} ]
  gap> List(GeneratorsOfGroup(g),x->Elements(ConfinalityClasses(x)[1]));
  [ [ [/ 2 ], [/ 3 ] ],
    [ [/ 1 ], [/ 2 ] ],
    [ [/ 1 ], [/ 2 ], [/ 3 ] ] ]
  gap> IsWeaklyFinitaryFRElement(BinaryAddingElement);
  false
  gap> IsWeaklyFinitaryFRElement(GuptaSidkiGroup.2);
  true
```

5.2.24   Germs

▷ Germs(*e*)   (attribute)

▷ NormOfBoundedFRElement(*e*)   (attribute)

   Returns: The germs of the bounded element *e*.

   The *germs* of a bounded element are the finitely many ultimately periodic sequences on which the state of *e* does not vanish. This function returns the germs of *e*, as a list of pairs; the first entry is a ray described as a periodic sequence of integers (see PeriodicList (11.2.2)), and the second entry is the periodic sequence of states that appear along that ray.

   The *norm* of a bounded element is the length of its list of germs.

```
                                ──── Example ────
  gap> Germs(BinaryAddingElement);
  [ [ [/ 2 ], [/ 1 ] ] ]
  gap> Germs(GrigorchukGroup.1);
  [  ]
  gap> Germs(GrigorchukGroup.2);
```

```
   [ [ [/ 2 ], [/ 1, 3, 5 ] ] ]
gap> Display(GrigorchukGroup.2);
   |  1      2
---+-----+-----+
 a | b,1    c,2
 b | d,2    d,1
 c | b,1    e,2
 d | d,1    d,2
 e | d,1    a,2
---+-----+-----+
Initial state: a
```

5.2.25 HasOpenSetConditionFRElement

▷ HasOpenSetConditionFRElement(e)   (property)

Returns: true if e has the open set condition.

An FR element e has the *open set condition* if for every infinite ray in the tree which is fixed by e, there is an open set around that ray which is also fixed by e. This function tests for e to have the open set condition. It currently is implemented only for bounded elements.

```
───────────────────── Example ─────────────────────
gap> HasOpenSetConditionFRElement(GrigorchukGroup.1);
true
gap> HasOpenSetConditionFRElement(GrigorchukGroup.2);
false
```

5.2.26 LimitFRMachine

▷ LimitFRMachine(m)   (attribute)

Returns: The submachine of m on all recurrent states.

This command creates a new Mealy machine, with stateset the limit states of m.

```
───────────────────── Example ─────────────────────
gap> m := MealyMachine([[2,2,3],[2,3,3],[3,3,3]],[(),(),(1,2,3)]);
<Mealy machine on alphabet [ 1 .. 3 ] with 3 states>
gap> Display(m);
   |  1      2      3
---+-----+-----+-----+
 a | b,1    b,2    c,3
 b | b,1    c,2    c,3
 c | c,2    c,3    c,1
---+-----+-----+-----+
gap> LimitStates(m);
[ <Mealy element on alphabet [ 1 .. 3 ] with 2 states>,
  <Mealy element on alphabet [ 1 .. 3 ] with 1 state> ]
gap> LimitFRMachine(m);
<Mealy machine on alphabet [ 1 .. 3 ] with 2 states>
gap> Display(last);
   |  1      2      3
---+-----+-----+-----+
 a | a,1    b,2    b,3
 b | b,2    b,3    b,1
---+-----+-----+-----+
```

5.2.27 NucleusMachine (FR machine)

▷ NucleusMachine(*m*)   (attribute)

Returns: The nucleus of *m*.

This command creates a new Mealy machine n, with stateset the nucleus (see NucleusOfFRMachine (4.2.13)) of *m*.

This nucleus machine is characterized as the smallest machine n such that Minimized(LimitFRMachine(m*n)) is isomorphic to n. It is also isomorphic to the NucleusMachine (7.2.20) of the state closure of the SCSemigroup (7.1.3) of *m*.

Note that the ordering of the states in the resulting machine is not necessarily the same as in *m*; however, if *m* and n are isomorphic, then this command returns *m*.

```
———————————————————————— Example ————————————————————————
gap> m := MealyMachine([[2,1,1],[2,2,2]],[(1,2,3),()]);
<Mealy machine on alphabet [ 1, 2, 3 ] with 2 states>
gap> Display(m);
   |  1     2     3
---+-----+-----+-----+
 a | b,2   a,3   a,1
 b | b,1   b,2   b,3
---+-----+-----+-----+
gap> NucleusMachine(m);
<Mealy machine on alphabet [ 1, 2, 3 ] with 3 states>
gap> Display(last);
   |  1     2     3
---+-----+-----+-----+
 a | a,1   a,2   a,3
 b | c,3   b,1   c,2
 c | a,2   c,3   c,1
---+-----+-----+-----+
```

5.2.28 GuessMealyElement

▷ GuessMealyElement(*p, d, n*)   (operation)

Returns: A Mealy element that probably has the same activity as *p*.

This function receives a permutation or transformation *p*, a degree *d* and a level *n*, and attempts to find a Mealy element on the alphabet [1..d] whose activity on level *n* is *p*.

This function returns fail if it thinks that the given level is not large enough to make a reasonable guess. In all cases, the function is not guaranteed to return the correct Mealy machine.

```
———————————————————————— Example ————————————————————————
gap> GuessMealyElement(Activity(GrigorchukGroup.2,6),2,6);
<Mealy element on alphabet [ 1, 2 ] with 5 states>
gap> last=GrigorchukGroup.2;
true
gap> GuessMealyElement(Activity(GrigorchukGroup.2,5),2,5);
fail
gap> ComposeElement([GrigorchukGroup.2,One(GrigorchukGroup)],());
<Mealy element on alphabet [ 1, 2 ] with 6 states>
gap> last=GuessMealyElement(Activity(GrigorchukGroup.2,6),2,7);
true
```

Chapter 6

Linear machines and elements

Linear machines are a special class of FR machines, in which the stateset Q and the alphabet X are vector spaces over a field $\Bbbk$, and the transition map $\phi : Q \otimes X \to X \otimes Q$ is a linear map; furthermore, there is a functional $\pi : Q \to \Bbbk$ called the *output*.

As before, a choice of initial state $q \in Q$ induces a linear map $q : T(X) \to T(X)$, where $T(X) = \bigoplus X^{\otimes n}$ is the tensor algebra generated by X. This map is defined as follows: given $x = x_1 \otimes \ldots \otimes x_n \in T(X)$, rewrite $q \otimes x$ as a sum of expressions of the form $y \otimes r$ with $y \in T(X)$ and $r \in Q$; then q, by definition, maps x to the sum of the $\pi(r)y$.

There are two sorts of linear machines: *vector machines*, for which the state space is a finite-dimensional vector space over a field; and *algebra machines*, for which the state space is a free algebra in a finite set of variables.

In a vector machine, the transition and output maps are stored as a matrix and a vector respectively. Minimization algorithms are implemented, as for Mealy machines.

In an algebra machine, the transition and output maps are stored as words in the algebra. These machines are natural extensions of group/monoid/semigroup machines.

Linear elements are given by a linear machine and an initial state. They can be added and multiplied, and act on the tensor algebra of the alphabet, admitting natural representations as matrices.

6.1 Methods and operations for `LinearFRMachine`s and `LinearFRElement`s

6.1.1 VectorMachine

▷ VectorMachine(*domain, transitions, output*)   (operation)
▷ VectorElement(*domain, transitions, output, init*)   (operation)
▷ VectorMachineNC(*fam, transitions, output*)   (operation)
▷ VectorElementNC(*fam, transitions, output, init, category*)   (operation)

Returns: A new vector machine/element.

This function constructs a new linear machine or element, of vector type.

transitions is a matrix of matrices; for `a,b` indices of basis vectors of the alphabet, `transitions[a][b]` is a square matrix indexed by the stateset, which is the transition to be effected on the stateset upon the output $a \to b$.

The optional last argument *category* specifies a category (IsAssociativeElement (**Reference: IsAssociativeElement**), IsJacobianElement (**Reference: IsJacobianElement**),...) to which the

new element should belong.

   `output` and `init` are vectors in the stateset.

   In the "NC" version, no tests are performed to check that the arguments contain values within bounds, or even of the right type (beyond the simple checking performed by GAP's method selection algorithms). The first argument should be the family of the resulting object. These "NC" methods are mainly used internally by the package.

```
——————————————— Example ———————————————
gap> M := VectorMachine(Rationals,[[[[1]],[[2]]],[[[3]],[[4]]]],[1]);
<Linear machine on alphabet Rationals^2 with 1-dimensional stateset>
gap> Display(M);
 Rationals | 1 | 2 |
----------+---+---+
        1 | 1 | 2 |
----------+---+---+
        2 | 3 | 4 |
----------+---+---+
Output: 1
gap> A := VectorElement(Rationals,[[[[1]],[[2]]],[[[3]],[[4]]]],[1],[1]);
<Linear element on alphabet Rationals^2 with 1-dimensional stateset>
gap> Display(Activity(A,2));
[ [   1,   2,   2,   4 ],
  [   3,   4,   6,   8 ],
  [   3,   6,   4,   8 ],
  [   9,  12,  12,  16 ] ]
gap> DecompositionOfFRElement(A);
[ [ <Linear element on alphabet Rationals^2 with 1-dimensional stateset>,
      <Linear element on alphabet Rationals^2 with 1-dimensional stateset> ],
  [ <Linear element on alphabet Rationals^2 with 1-dimensional stateset>,
      <Linear element on alphabet Rationals^2 with 1-dimensional stateset> ] ]
gap> last=[[A,2*A],[3*A,4*A]];
true
```

6.1.2 AssociativeObject

▷ AssociativeObject(x)   (operation)

   Returns: An associative object related to *x*.

   If *x* belongs to a family that admits a non-associative and an associative product, and the product of *x* is non-associative, this function returns the object corresponding to *x*, but with associative product.

   A typical example is that *x* is a derivation of a vector space. The product of derivations is $a \circ b - b \circ a$, and is not associative; but derivations are endomorphisms of the vector space, and as such can be composed associatively.

```
——————————————— Example ———————————————
gap> A := VectorElement(Rationals,[[[[0]],[[1]]],[[[1]],[[0]]]],[1],[1],IsJacobianElement);
<Linear element on alphabet Rationals^2 with 1-dimensional stateset->
gap> A^2;
<Zero linear element on alphabet Rationals^2->
gap> AssociativeObject(A)^2;
<Identity linear element on alphabet Rationals^2>
```

6.1.3 AlgebraMachine

▷ AlgebraMachine(*[domain,]ring, transitions, output*)   (operation)
▷ AlgebraElement(*[domain,]ring, transitions, output, init*)   (operation)
▷ AlgebraMachineNC(*fam, ring, transitions, output*)   (operation)
▷ AlgebraElementNC(*fam, ring, transitions, output, init*)   (operation)
    Returns: A new algebra machine/element.
    This function constructs a new linear machine or element, of algebra type.
    ring is a free associative algebra, optionally with one. *domain* is the vector space on which the alphabet is defined. If absent, this argument defaults to the LeftActingDomain (**Reference: LeftActingDomain**) of *ring*.
    transitions is a list of matrices; for each generator number *i* of *ring*, the matrix transitions[i], with entries in *ring*, describes the decomposition of generator *i* as a matrix.
    output is a vector over *domain*, and *init* is a vector over *ring*.
    In the "NC" version, no tests are performed to check that the arguments contain values within bounds, or even of the right type (beyond the simple checking performed by GAP's method selection algorithms). The first argument should be the family of the resulting object. These "NC" methods are mainly used internally by the package.

```
———————————————— Example ————————————————
gap> F := FreeAssociativeAlgebraWithOne(Rationals,1);;
gap> A := AlgebraMachine(F,[[[F.1,F.1^2+F.1],[One(F),Zero(F)]]],[1]);
<Linear machine on alphabet Rationals^2 with generators [ (1)*x.1 ]>
gap> Display(A);
 Rationals |     1    |     2    |
-----------+----------+----------+
         1 |      x.1 | x.1+x.1^2 |
-----------+----------+----------+
         2 |        1 |        0 |
-----------+----------+----------+
Output: 1
gap> M := AlgebraElement(F,[[[F.1,F.1^2+F.1],[One(F),Zero(F)]]],[1],F.1);
<Rationals^2|(1)*x.1>
gap> Display(Activity(M,2));
[ [  1,  2,  4,  4 ],
  [  1,  0,  2,  2 ],
  [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ] ]
```

6.1.4 Transition (Linear machine)

▷ Transition(*m, s, a, b*)   (operation)
    Returns: An element of *m*'s stateset.
    This function returns the state reached by *m* when started in state *s* and performing output $a \to b$.

```
———————————————— Example ————————————————
gap> M := AsVectorMachine(Rationals,FRMachine(GuptaSidkiGroup.2));
<Linear machine on alphabet Rationals^3 with 4-dimensional stateset>
gap> Transition(M,[1,0,0,0],[1,0,0],[1,0,0]);
[ 0, 1, 0, 0 ]
gap> Transition(M,[1,0,0,0],[0,1,0],[0,1,0]);
[ 0, 0, 1, 0 ]
```

```
gap> Transition(M,[1,0,0,0],[0,0,1],[0,0,1]);
[ 1, 0, 0, 0 ]
gap> A := AsVectorElement(Rationals,GuptaSidkiGroup.2);
<Linear element on alphabet Rationals^3 with 4-dimensional stateset>
gap> Transition(A,[1,0,0],[1,0,0]);
[ 0, 1, 0, 0 ]
```

6.1.5 Transitions

▷ Transitions(*m*, *s*, a)   (operation)

Returns: An vector of elements of *m*'s stateset.

This function returns the state reached by *m* when started in state *s* and receiving input a. The output is a vector, indexed by the alphabet's basis, of output states.

```
――――――――――――――――――――――― Example ―――――――――――――――――――――――
gap> M := AsVectorMachine(Rationals,FRMachine(GuptaSidkiGroup.2));
<Linear machine on alphabet Rationals^3 with 4-dimensional stateset>
gap> Transitions(M,[1,0,0,0],[1,0,0]);
[ [ 0, 1, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ]
gap> A := AsVectorElement(Rationals,GuptaSidkiGroup.2);
<Linear element on alphabet Rationals^3 with 4-dimensional stateset>
gap> Transitions(A,[1,0,0]);
[ [ 0, 1, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ]
```

6.1.6 NestedMatrixState

▷ NestedMatrixState(*e*, *i*, *j*)   (operation)
▷ NestedMatrixCoefficient(*e*, *i*, *j*)   (operation)

Returns: A coefficent of an iterated decomposition of *e*.

The first form returns the entry at position (i, j) of *e*'s decomposition. Both of *i,j* are lists. The second form returns the output of the state.

In particular, e=NestedMatrixState(e,[],[]), and
Activity(e,1)[i][j]=NestedMatrixCoefficient(e,[i],[j]), and
DecompositionOfFRElement(e,1)[i][j]=NestedMatrixState(e,[i],[j]).

```
――――――――――――――――――――――― Example ―――――――――――――――――――――――
gap> A := AsVectorElement(Rationals,GuptaSidkiGroup.2);;
gap> A=NestedMatrixState(A,[3,3],[3,3]);
true
gap> IsOne(NestedMatrixState(A,[3,3,3,3,1,1],[3,3,3,3,1,2]));
true
gap> List([1..3],i->List([1..3],j->NestedMatrixCoefficient(A,[i],[j])))=Activity(A,1);
true
```

6.1.7 ActivitySparse

▷ ActivitySparse(*m*, *i*)   (operation)

Returns: A sparse matrix.

Activity(m,i) returns an $n^i \times n^i$ matrix describing the action on the *i*-fold tensor power of the alphabet. This matrix can also be returned as a sparse matrix, and this is performed by this command. A

sparse matrix is described as a list of expressions of the form `[[i,j],c]`, representing the elementary matrix with entry c at position (i, j). The activity matrix is then the sum of these elementary matrices.

```
———————————————— Example ————————————————
gap> A := AsVectorElement(Rationals,GuptaSidkiGroup.2);;
gap> Display(Activity(A,2));
[ [  0,  1,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  0,  0,  0,  0,  0 ],
  [  1,  0,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  0,  0,  0,  1,  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,  1,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  0,  1 ] ]
gap> ActivitySparse(A,2);
[ [ [ 1, 2 ], 1 ], [ [ 2, 3 ], 1 ], [ [ 3, 1 ], 1 ], [ [ 4, 6 ], 1 ],
[ [ 5, 4 ], 1 ], [ [ 6, 5 ], 1 ], [ [ 7, 7 ], 1 ], [ [ 8, 8 ], 1 ],
[ [ 9, 9 ], 1 ] ]
```

6.1.8 Activities

▷ Activities(*m*, *i*)   (operation)

Returns: Activities of `m` on the first `i` levels.

`Activity(m,i)` returns an $n^i \times n^i$ matrix describing the action on the i-fold tensor power of the alphabet. This command returns `List([0..i-1],j->Activity(m,j))`.

```
———————————————— Example ————————————————
gap> A := AsVectorElement(Rationals,GrigorchukGroup.2);;
gap> Activities(A,3);
[ [ [ 1 ] ],
  [ [ 1, 0 ], [ 0, 1 ] ],
  [ [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ] ]
```

6.1.9 IsConvergent

▷ IsConvergent(*e*)   (property)

Returns: Whether the linear element `e` is convergent.

A linear element is *convergent* if its state at position $(1, 1)$ is equal to itself.

```
———————————————— Example ————————————————
gap> n := 3;;
gap> shift := VectorElement(CyclotomicField(n), [[[[1,0],[0,0]],
      [[0,0],[0,1]]],[[[0,1],[0,0]],[[1,0],[0,0]]]],[1,E(n)],[1,0]);
<Linear element on alphabet CF(3)^2 with 2-dimensional stateset>
gap> IsConvergent(shift);
true
gap> Display(Activity(shift,2));
[ [    1,    0,    0,    0 ],
  [ E(3),    1,    0,    0 ],
  [    0, E(3),    1,    0 ],
  [    0,    0, E(3),    1 ] ]
gap> Display(Activity(shift,3));
[ [    1,    0,    0,    0,    0,    0,    0,    0 ],
```

```
[  E(3),     1,     0,     0,     0,     0,     0,     0 ],
[     0,  E(3),     1,     0,     0,     0,     0,     0 ],
[     0,     0,  E(3),     1,     0,     0,     0,     0 ],
[     0,     0,     0,  E(3),     1,     0,     0,     0 ],
[     0,     0,     0,     0,  E(3),     1,     0,     0 ],
[     0,     0,     0,     0,     0,  E(3),     1,     0 ],
[     0,     0,     0,     0,     0,     0,  E(3),     1 ] ]
```

6.1.10 TransposedFRElement

▷ TransposedFRElement(*e*)   (operation)
▷ IsSymmetricFRElement(*e*)   (property)
▷ IsAntisymmetricFRElement(*e*)   (property)
▷ IsLowerTriangularFRElement(*e*)   (property)
▷ IsUpperTriangularFRElement(*e*)   (property)
▷ IsDiagonalFRElement(*e*)   (property)

   Returns: The elementary matrix operation/property.

   Since linear FR elements may be interpreted as infinite matrices, it makes sense to transpose them, test whether they're symmetric, antisymmetric, diagonal, or triangular.

———————————————————————— Example ————————————————————————
```
gap> n := 3;;
gap> shift := VectorElement(CyclotomicField(n), [[[[1,0],[0,0]],
    [[0,0],[0,1]]],[[[0,1],[0,0]],[[1,0],[0,0]]]],[1,E(n)],[1,0]);
<Linear element on alphabet CF(3)^2 with 2-dimensional stateset>
gap> Display(Activity(shift,2));
[ [    1,    0,    0,    0 ],
  [ E(3),    1,    0,    0 ],
  [    0, E(3),    1,    0 ],
  [    0,    0, E(3),    1 ] ]
gap> Display(Activity(TransposedFRElement(shift),2));
[ [    1, E(3),    0,    0 ],
  [    0,    1, E(3),    0 ],
  [    0,    0,    1, E(3) ],
  [    0,    0,    0,    1 ] ]
gap> IsSymmetricFRElement(shift);
false
gap> IsSymmetricFRElement(shift+TransposedFRElement(shift));
true
gap> IsLowerTriangularFRElement(shift);
true
gap> IsUpperTriangularFRElement(shift);
false
```

6.1.11 LDUDecompositionFRElement

▷ LDUDecompositionFRElement(*e*)   (operation)

   Returns: A factorization $e = LDU$.

   Given a linear element *e*, this command attempts to find a decomposition of the form $e = LDU$, where L is lower triangular, D is diagonal, and U is upper triangular (see IsLowerTriangularFRElement (6.1.10) etc.).

The result is returned thas a list with entries L, D, U. Note that it is not guaranteed to succeed. For more examples, see Section 9.4.

```
 ────────────────────────── Example ──────────────────────────
  gap> List([0..7],s->List([0..7],t->E(4)^ValuationInt(Binomial(s+t,s),2)));;
  gap> A := GuessVectorElement(last);
  <Linear element on alphabet GaussianRationals^2 with 2-dimensional stateset>
  gap> LDU := LDUDecompositionFRElement(A);
  [ <Linear element on alphabet GaussianRationals^2 with 4-dimensional stateset>,
    <Linear element on alphabet GaussianRationals^2 with 3-dimensional stateset>,
    <Linear element on alphabet GaussianRationals^2 with 4-dimensional stateset> ]
  gap> IsLowerTriangularFRElement(LDU[1]); IsDiagonalFRElement(LDU[2]);
  true
  true
  gap> TransposedFRElement(LDU[1])=LDU[3];
  true
  gap> Product(LDU)=A;
  true
```

6.1.12 GuessVectorElement

▷ GuessVectorElement(*m*)   (function)

Returns: A vector element that acts like *m*.

The arguments to this function include a matrix or list of matrices, and an optional ring. The return value is a vector element, over the ring if it was specified, that acts like the sequence of matrices.

If a single matrix is specified, then it is assumed to represent a convergent element (see IsConvergent (6.1.9)).

This function returns `fail` if it believes that it does not have enough information to make a reasonable guess.

```
 ────────────────────────── Example ──────────────────────────
  gap> n := 3;;
  gap> shift := VectorElement(CyclotomicField(n), [[[[1,0],[0,0]],
        [[0,0],[0,1]]],[[[0,1],[0,0]],,[[1,0],[0,0]]]],[1,E(n)],[1,0]);;
  <Linear element on alphabet CF(3)^2 with 2-dimensional stateset>
  gap> GuessVectorElement(Activity(shift,3)); last=shift;
  <Linear element on alphabet CF(3)^2 with 2-dimensional stateset>
  true
  gap> GuessVectorElement(Inverse(Activity(shift,4)));
  fail
  gap> GuessVectorElement(Inverse(Activity(shift,5)));
  <Linear element on alphabet CF(3)^2 with 4-dimensional stateset>
  gap> IsOne(last*shift);
  true
```

6.1.13 AsLinearMachine

▷ AsLinearMachine(*r, m*)   (operation)
▷ AsLinearElement(*r, m*)   (operation)

Returns: The linear machine/element associated with *m*.

This command accepts a domain and an ordinary machine/element, and constructs the corresponding linear machine/element, defined by extending linearly the action on $[1..d]$ to an action on r^d.

If *m* is a Mealy machine/element, the result is a vector machine/element. If *m* is a group/monoid/semigroup machine/element, the result is an algebra machine/element. To obtain explicitly a vector or algebra machine/element, see `AsVectorMachine` (6.1.14) and `AsAlgebraMachine` (6.1.15).

```
——————————— Example ———————————
gap> Display(I4Machine);
   | 1     2
---+-----+-----+
 a | c,2   c,1
 b | a,1   b,1
 c | c,1   c,2
---+-----+-----+
gap> A := AsLinearMachine(Rationals,I4Machine);
<Linear machine on alphabet Rationals^2 with 3-dimensional stateset>
Correspondence(A);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> Display(A);
 Rationals |   1   |   2   |
-----------+-------+-------+
         1 | 0 0 0 | 0 0 1 |
           | 1 0 0 | 0 0 0 |
           | 0 0 1 | 0 0 0 |
-----------+-------+-------+
         2 | 0 0 1 | 0 0 0 |
           | 0 1 0 | 0 0 0 |
           | 0 0 0 | 0 0 1 |
-----------+-------+-------+
Output: 1 1 1
gap> B := AsLinearMachine(Rationals,AsMonoidFRMachine(I4Machine));
<Linear machine on alphabet Rationals^2 with generators [ (1)*m1, (1)*m2 ]>
gap> Correspondence(B);
MappingByFunction( <free monoid on the generators [ m1, m2 ]>,
<algebra-with-one over Rationals, with 2 generators>, function( w ) ... end )
gap> Display(B);
 Rationals | 1  | 2  |
-----------+----+----+
         1 |  0 |  1 |
           | m1 |  0 |
-----------+----+----+
         2 |  1 |  0 |
           | m2 |  0 |
-----------+----+----+
Output: 1 1
gap> AsLinearElement(Rationals,I4Monoid.1)*AsLinearElement(Rationals,I4Monoid.2);
<Linear element on alphabet Rationals^2 with 4-dimensional stateset>
gap> last=AsLinearElement(Rationals,I4Monoid.1*I4Monoid.2);
true
```

6.1.14   AsVectorMachine

▷ AsVectorMachine(*r, m*)   (operation)
▷ AsVectorElement(*r, m*)   (operation)

Returns: The vector machine/element associated with m.

This command accepts a domain and an ordinary machine/element, and constructs the corresponding linear machine/element, defined by extending linearly the action on $[1..d]$ to an action on r^d. For this command to succeed, the machine/element m must be finite state. For examples see `AsLinearMachine` (6.1.13).

6.1.15 AsAlgebraMachine

▷ AsAlgebraMachine(r, m)   (operation)

▷ AsAlgebraElement(r, m)   (operation)

    Returns: The algebra machine/element associated with m.

This command accepts a domain and an ordinary machine/element, and constructs the corresponding linear machine/element, defined by extending linearly the action on $[1..d]$ to an action on r^d. For examples see `AsLinearMachine` (6.1.13).

6.1.16 AsVectorMachine (Linear machine)

▷ AsVectorMachine(m)   (operation)

▷ AsVectorElement(m)   (operation)

    Returns: The machine/element m in vector form.

This command accepts a linear machine, and converts it to vector form. This command is not guaranteed to terminate.

```
                            Example
  gap> A := AsLinearElement(Rationals,I4Monoid.1);
  <Linear element on alphabet Rationals^2 with 2-dimensional stateset>
  gap> B := AsAlgebraElement(A);
  <Rationals^2|(1)*x.1>
  gap> C := AsVectorElement(B);
  gap> A=B; B=C;
  true
  true
```

6.1.17 AsAlgebraMachine (Linear machine)

▷ AsAlgebraMachine(m)   (operation)

▷ AsAlgebraElement(m)   (operation)

    Returns: The machine/element m in algebra form.

This command accepts a linear machine, and converts it to algebra form.

```
                            Example
  gap> A := AsLinearElement(Rationals,I4Monoid.1);
  <Linear element on alphabet Rationals^2 with 2-dimensional stateset>
  gap> AsAlgebraElement(A)=AsAlgebraElement(Rationals,I4Monoid.1);
  true
  gap> A=AsAlgebraElement(A);
  true
```

Chapter 7

Self-similar groups, monoids and semigroups

Self-similar groups, monoids and semigroups (below *FR semigroups*) are simply groups, monoids and semigroups whose elements are FR machines. They naturally act on the alphabet of their elements, and on sequences over that alphabet.

Most non-trivial calculations in FR groups are performed as follows: GAP searches through words of short length in the generating set of a FR group to find a solution to a group-theoretic question, and at the same time searches through the finite quotients to prove the inexistence of a solution. Often the calculation ends with the answer `fail`, which means that no definite answer, neither positive nor negative, could be found; however, the cases where the calculation actually terminates have been most useful.

The maximal length of words to consider in the search is controlled by the variable `FR_SEARCH.radius` (initially 10), and the maximal depth of the tree in which to search is controlled by the variable `FR_SEARCH.depth` (initially 6). These limits can be modified in any function call using GAP's options mechanism, e.g. in `Index(G,H:FRdepth:=5,FRradius:=5)`.

7.1 Creators for FR semigroups

The most straightforward creation method for FR groups is `Group()`, applied with FR elements as arguments. There are shortcuts to this somewhat tedious method:

7.1.1 FRGroup

▷ FRGroup({*definition*, })   (operation)
▷ FRMonoid({*definition*, })   (operation)
▷ FRSemigroup({*definition*, })   (operation)

Returns: A new self-similar group/monoid/semigroup.

This function constructs a new FR group/monoid/semigroup, generated by group FR elements. It receives as argument any number of strings, each of which represents a generator of the object to be constructed.

Each *definition* is of the form `"name=projtrans"`, where each of `proj` and `trans` is optional. `proj` is of the form `<w1,...,wd>`, where each `wi` is a (possibly empty) word in the names or is

1. `trans` is either a permutation in disjoint cycle notation, or a list, representing the images of a permutation.

The last argument may be one of the filters `IsMealyElement`, `IsFRMealyElement` or `IsFRElement`. By default, if each of the states of generators is a generator or 1, the elements of the created object will be Mealy elements; otherwise, they will be FR elements. Specifying such a filter requires them to be in the appropriate category; e.g., `FRGroup("a=(1,2)",IsFRMealyElement)` asks for the resulting group to be generated by FR-Mealy elements. The generators must of course be finite-state.

```
─────────── Example ───────────
gap> FRGroup("a=(1,2)","b=(1,2,3,4,5)"); Size(last);
<self-similar group over [ 1 .. 5 ] with 2 generators>
120
gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> AssignGeneratorVariables(Dinfinity);
#I  Assigned the global variables [ a, b ]
gap> Order(a); Order(b); Order(a*b);
2
2
infinity
gap> ZZ := FRGroup("t=<,t>[2,1]");
<self-similar group over [ 1 .. 2 ] with 1 generator>
tau := FRElement([[[b,1],[1]]],[()],[1]);
<2|f3>
gap> IsSubgroup(Dinfinity,ZZ);
false
gap> IsSubgroup(Dinfinity^tau,ZZ);
true
gap> Index(Dinfinity^tau,ZZ);
2
```

```
─────────── Example ───────────
gap> i4 := FRMonoid("s=(1,2)","f=<s,f>[1,1]");
<self-similar monoid over [ 1 .. 2 ] with 2 generators>
gap> f := GeneratorsOfMonoid(i4){[1,2]};;
gap> for i in [1..10] do Add(f,f[i]*f[i+1]); od;
gap> f[1]^2=One(m);
true
gap> f[2]^3=f[2];
true
gap> f[11]*f[10]^2=f[1]*Product(f{[5,7..11]})*f[10];
true
gap> f[12]*f[11]^2=f[2]*Product(f{[6,8..12]})*f[11];
true
```

```
─────────── Example ───────────
gap> i2 := FRSemigroup("f0=<f0,f0>(1,2)","f1=<f1,f0>[2,2]");
<self-similar semigroup over [ 1 .. 2 ] with 2 generators>
gap> AssignGeneratorVariables(i2);
#I  Assigned the global variables [ "f0", "f1" ]
gap> f0^2=One(i2);
true
```

```
gap> ForAll([0..10],p->(f0*f1)^p*(f1*f0)^p*f1=f1^2*(f0*f1)^p*(f1*f0)^p*f1);
true
```

7.1.2 NewSemigroupFRMachine

▷ NewSemigroupFRMachine(...)   (attribute)
▷ NewMonoidFRMachine(...)   (attribute)
▷ NewGroupFRMachine(...)   (attribute)

Returns: A new FR machine, based on string descriptions.

This command constructs a new FR machine, in a format similar to `FRGroup` (7.1.1); namely, the arguments are strings of the form "gen=<word-1,...,word-d>perm"; each `word-i` is a word in the generators; and `perm` is a transformation, either written in disjoint cycle or in images notation.

Except in the semigroup case, `word-i` is allowed to be the empty string; and the "<...>" may be skipped altogether. In the group or IMG case, each `word-i` may also contain inverses.

The following example constructs the "universal Grigorchuk machine".

```
 ─────────────────────────── Example ───────────────────────────
  gap> m := NewGroupFRMachine("a=(1,2)(3,4)(5,6)","b=<a,b,a,b,,b>",
        "c=<a,c,,c,a,c>","d=<,d,a,d,a,d>");
  gap> <FR machine with alphabet [ 1, 2, 3, 4, 5, 6 ] on Group( [ a, b, c, d ] )>
```

7.1.3 SCGroup

▷ SCGroup(*m*)   (operation)
▷ SCGroupNC(*m*)   (operation)
▷ SCMonoid(*m*)   (operation)
▷ SCMonoidNC(*m*)   (operation)
▷ SCSemigroup(*m*)   (operation)
▷ SCSemigroupNC(*m*)   (operation)

Returns: The state-closed group/monoid/semigroup generated by the machine *m*.

This function constructs a new FR group/monoid/semigroup g, generated by all the states of the FR machine *m*. There is a bijective correspondence between `GeneratorsOfFRMachine(m)` and the generators of g, which is accessible via `Correspondence(g)` (See Correspondence (7.1.4)); it is a homomorphism from the stateset of *m* to g, or a list indicating for each state of *m* a corresponding generator index in the generators of g (with negatives for inverses, and 0 for identity).

In the non-`NC` forms, redundant (equal, trivial or mutually inverse) states are removed from the generating set of g.

```
 ─────────────────────────── Example ───────────────────────────
  gap> b := MealyMachine([[3,2],[3,1],[3,3]],[(1,2),(),()]);; g := SCGroupNC(b);
  <self-similar group over [ 1 .. 2 ] with 3 generators>
  gap> Size(g);
  infinity
  gap> IsOne(Comm(g.2,g.2^g.1));
  true
```

```
 ─────────────────────────── Example ───────────────────────────
  gap> i4machine := MealyMachine([[3,3],[1,2],[3,3]],[(1,2),[1,1],()]);
  <Mealy machine on alphabet [ 1, 2 ] with 3 states>
  gap> IsInvertible(i4machine);
  false
```

```
gap> i4 := SCMonoidNC(i4machine);
<self-similar monoid over [ 1 .. 2 ] with 3 generators>
gap> f := GeneratorsOfMonoid(i4){[1,2]};;
gap> for i in [1..10] do Add(f,f[i]*f[i+1]); od;
gap> f[1]^2=One(m);
true
gap> f[2]^3=f[2];
true
gap> f[11]*f[10]^2=f[1]*Product(f{[5,7..11]})*f[10];
true
gap> f[12]*f[11]^2=f[2]*Product(f{[6,8..12]})*f[11];
true
```

```
                            ____ Example ____
gap> i2machine := MealyMachine([[1,1],[2,1]],[(1,2),[2,2]]);
<Mealy machine on alphabet [ 1, 2 ] with 2 states>
gap> i2 := SCSemigroupNC(i2machine);
<self-similar semigroup over [ 1 .. 2 ] with 2 generators>
gap> f0 := GeneratorsOfSemigroup(i2)[1];; f1 := GeneratorsOfSemigroup(i2)[2];;
gap> f0^2=One(i2);
true
gap> ForAll([0..10],p->(f0*f1)^p*(f1*f0)^p*f1=f1^2*(f0*f1)^p*(f1*f0)^p*f1);
true
```

7.1.4 Correspondence (FR semigroup)

▷ Correspondence(g)   (attribute)

Returns: A correspondence between the generators of the underlying FR machine of g and g.

If g was created as the state closure of an FR machine m, this attribute records the correspondence between m and g.

If m is a group/monoid/semigroup/algebra FR machine, then Correspondence(g) is a homomorphism from the stateset of m to g.

If m is a Mealy or vector machine, then Correspondence(g) is a list, with in position i the index in the generating set of g of state number i. This index is 0 if there is no corresponding generator because the state is trivial, and is negative if there is no corresponding generator because the inverse of state number i is a generator.

See SCGroupNC (7.1.3), SCGroup (7.1.3), SCMonoidNC (7.1.3), SCMonoid (7.1.3), SCSemigroupNC (7.1.3), SCSemigroup (7.1.3), SCAlgebraNC (8.1.2), SCAlgebra (8.1.2), SCAlgebraWithOneNC (8.1.2), and SCAlgebraWithOne (8.1.2) for examples.

7.1.5 FullSCGroup

▷ FullSCGroup(...)   (function)
▷ FullSCMonoid(...)   (function)
▷ FullSCSemigroup(...)   (function)

Returns: A maximal state-closed group/monoid/semigroup on the alphabet a.

This function constructs a new FR group, monoid or semigroup, which contains all transformations with given properties of the tree with given alphabet.

The arguments can be, in any order: a semigroup, specifying which vertex actions are allowed; a set or domain, specifying the alphabet of the tree; an integer, specifying the maximal depth

of elements; and a filter among `IsFinitaryFRElement` (5.2.10), `IsBoundedFRElement` (5.2.12), `IsPolynomialGrowthFRElement` (5.2.13) and `IsFiniteStateFRElement` (4.2.12).

This object serves as a container for all FR elements with alphabet `a`. Random elements can be drawn from it; they are Mealy elements with a random number of states, and with the required properties.

```
——————————————————— Example ———————————————————
gap> g := FullSCGroup([1..3]);
FullSCGroup([ 1 .. 3 ]);
gap> IsSubgroup(g,GuptaSidkiGroup);
true
gap> g := FullSCGroup([1..3],Group((1,2,3)));
FullSCGroup([ 1 .. 3 ], Group( [ (1,2,3) ] ))
gap> IsSubgroup(g,GuptaSidkiGroup);
true
gap> IsSubgroup(g,GrigorchukGroup);
false
gap> Random(g);
<Mealy element on alphabet [ 1, 2, 3 ] with 2 states, initial state 1>
gap> Size(FullSCGroup([1,2],3));
128
gap> g := FullSCMonoid([1..2]);
FullSCMonoid([ 1 .. 2 ])
gap> IsSubset(g,AsTransformation(FullSCGroup([1..2])));
true
gap> IsSubset(g,AsTransformation(GrigorchukGroup));
true
gap> g := FullSCSemigroup([1..3]);
FullSCSemigroup([ 1 .. 3 ])
gap> h := FullSCSemigroup([1..3],Semigroup(Transformation([1,1,1])));
FullSCSemigroup([ 1 .. 3 ], Semigroup( [ [ 1, 1, 1 ] ] ))
gap> Size(h);
1
gap> IsSubset(g,h);
true
gap> g=FullSCMonoid([1..3]);
true
```

7.1.6 FRMachineFRGroup

▷ FRMachineFRGroup(*g*)   (operation)
▷ FRMachineFRMonoid(*g*)   (operation)
▷ FRMachineFRSemigroup(*g*)   (operation)
▷ MealyMachineFRGroup(*g*)   (operation)
▷ MealyMachineFRMonoid(*g*)   (operation)
▷ MealyMachineFRSemigroup(*g*)   (operation)

Returns: A machine describing all generators of *g*.

This function constructs a new group/monoid/semigroup/Mealy FR machine, with (at least) one generator per generator of *g*. This is done by adding all machines of all generators of *g*, and minimizing.

In particular, if *g* is state-closed, then `SCGroup(FRMachineFRGroup(g))` gives a group isomorphic to *g*, and similarly for monoids and semigroups.

```
                          —— Example ——
gap> FRMachineFRGroup(GuptaSidkiGroup);
<FR machine with alphabet [ 1 .. 3 ] on Group( [ f11, f12 ] )>
gap> Display(last);
 G   |    1          2         3
-----+--------+----------+-------+
 f11 | <id>,2     <id>,3    <id>,1
 f12 |  f11,1    f11^-1,2    f12,3
-----+--------+----------+-------+
```

```
                          —— Example ——
gap> FRMachineFRMonoid(I4Monoid);
<FR machine with alphabet [ 1 .. 2 ] on Monoid( [ m11, m12 ], ... )>
gap> Display(last);
 M   |    1         2
-----+--------+--------+
 m11 | <id>,2    <id>,1
 m12 |  m11,1     m12,1
-----+--------+--------+
```

```
                          —— Example ——
gap> FRMachineFRSemigroup(I2Monoid);
<FR machine with alphabet [ 1 .. 2 ] on Semigroup( [ s11, s12, s1 ] )>
gap> Display(last);
 S   |    1         2
-----+-------+-------+
 s11 | s11,1    s11,2
 s12 | s12,2    s12,1
  s1 |  s1,2    s12,2
-----+-------+-------+
```

7.1.7 IsomorphismFRGroup

▷ IsomorphismFRGroup(*g*)   (operation)
▷ IsomorphismFRMonoid(*g*)   (operation)
▷ IsomorphismFRSemigroup(*g*)   (operation)

Returns: An isomorphism towards a group/monoid/semigroup on a single FR machine.

This function constructs a new FR group/monoid/semigroup, such that all elements of the resulting object have the same underlying group/monoid/semigroup FR machine.

```
                          —— Example ——
gap> phi := IsomorphismFRGroup(GuptaSidkiGroup);
[ <Mealy element on alphabet [ 1, 2, 3 ] with 2 states, initial state 1>,
  <Mealy element on alphabet [ 1, 2, 3 ] with 4 states, initial state 1> ] ->
[ <3|identity ...>, <3|f1>, <3|f1^-1>, <3|f2> ]
gap> Display(GuptaSidkiGroup.2);
   | 1     2     3
---+-----+-----+-----+
 a | a,1   a,2   a,3
 b | a,2   a,3   a,1
 c | a,3   a,1   a,2
```

```
  d | b,1    c,2    d,3
 ---+-----+-----+-----+
 Initial state: d
 gap> Display(GuptaSidkiGroup.2^phi);
     |    1         2         3
 ----+--------+---------+-------+
  f1 | <id>,2    <id>,3    <id>,1
  f2 |   f1,1   f1^-1,2     f2,3
 ----+--------+---------+-------+
 Initial state: f2
```

─────────────────────── Example ───────────────────────
```
 gap> phi := IsomorphismFRSemigroup(I2Monoid);
 MappingByFunction( I2, <self-similar semigroup over [ 1 .. 2 ] with
 3 generators>, <Operation "AsSemigroupFRElement"> )
 gap> Display(GeneratorsOfSemigroup(I2Monoid)[3]);
     | 1     2
 ---+-----+-----+
  a | a,2    b,2
  b | b,2    b,1
 ---+-----+-----+
 Initial state: a
 gap> Display(GeneratorsOfSemigroup(I2Monoid)[3]^phi);
  S  | 1      2
 ----+------+------+
  s1 | s1,2    s2,2
  s2 | s2,2    s2,1
 ----+------+------+
 Initial state: s1
```

─────────────────────── Example ───────────────────────
```
 gap> phi := IsomorphismFRMonoid(I4Monoid);
 MappingByFunction( I4, <self-similar monoid over [ 1 .. 2 ] with
 2 generators>, <Operation "AsMonoidFRElement"> )
 gap> Display(GeneratorsOfMonoid(I4Monoid)[1]);
     | 1     2
 ---+-----+-----+
  a | b,2    b,1
  b | b,1    b,2
 ---+-----+-----+
 Initial state: a
 gap> Display(GeneratorsOfMonoid(I4Monoid)[1]^phi);
  M  |    1         2
 ----+--------+--------+
  m1 | <id>,2    <id>,1
 ----+--------+--------+
 Initial state: m1
```

7.1.8 IsomorphismMealyGroup

▷ IsomorphismMealyGroup(g)   (operation)
▷ IsomorphismMealyMonoid(g)   (operation)
▷ IsomorphismMealySemigroup(g)   (operation)

Returns: An isomorphism towards a group/monoid/semigroup all of whose elements are Mealy machines.

This function constructs a new FR group/monoid/semigroup, such that all elements of the resulting object are Mealy machines.

```
─────────────────────────── Example ───────────────────────────
gap> G := FRGroup("a=(1,2)","b=<a,b>","c=<c,b>");
<self-similar group over [ 1 .. 2 ] with 3 generators>
gap> phi := IsomorphismMealyGroup(G);
[ <2|a>, <2|b>, <2|c> ] ->
[ <Mealy element on alphabet [ 1, 2 ] with 2 states, initial state 1>,
  <Mealy element on alphabet [ 1, 2 ] with 3 states, initial state 1>,
  <Mealy element on alphabet [ 1, 2 ] with 4 states, initial state 1> ]
gap> Display(G.3);
   |    1        2
---+--------+--------+
 a | <id>,2   <id>,1
 b |    a,1       b,2
 c |    c,1       b,2
---+--------+--------+
Initial state: c
gap> Display(G.3^phi);
   | 1      2
---+-----+-----+
 a | a,1    b,2
 b | c,1    b,2
 c | d,2    d,1
 d | d,1    d,2
---+-----+-----+
Initial state: a
```

7.1.9 FRGroupByVirtualEndomorphism

▷ FRGroupByVirtualEndomorphism(*hom[, transversal]*) (operation)

Returns: A new self-similar group.

This function constructs a new FR group P, generated by group FR elements. Its first argument is a virtual endomorphism of a group G, i.e. a homomorphism from a subgroup H to G. The constructed FR group acts on a tree with alphabet a transversal of H in G (represented as [1..d]), and is a homomorphic image of G. The stabilizer of the first-level vertex corresponding to the trivial coset is the image of H. This function is loosely speaking an inverse of VirtualEndomorphism (7.2.29).

The optional second argument is a transversal of H in G, either of type IsRightTransversal or a list.

Furthermore, an option "MealyElement" can be passed to the function, as FRGroupByVirtualEndomorphism(f:MealyElement), to require the resulting group to be generated by Mealy elements and not FR elements. The call will succeed, of course, only if the representation of G is finite-state.

The resulting FR group has an attribute Correspondence(P) that records a homomorphism from G to P.

The example below constructs the binary adding machine, and a non-standard representation of it.

```
─────────────────────────── Example ───────────────────────────
gap> G := FreeGroup(1);
<free group on the generators [ f1 ]>
gap> f := GroupHomomorphismByImages(Group(G.1^2),G,[G.1^2],[G.1]);
[ f1^2 ] -> [ f1 ]
gap> H := FRGroupByVirtualEndomorphism(f);
<self-similar group over [ 1 .. 2 ] with 1 generator>
gap> Display(H.1);
    |     1       2
----+--------+------+
 x1 | <id>,2   x1,1
----+--------+------+
Initial state: x1
gap> Correspondence(H);
[ f1 ] -> [ <2|x1> ]
gap> H := FRGroupByVirtualEndomorphism(f,[G.1^0,G.1^3]);;
gap> Display(H.1);
    |      1        2
----+---------+--------+
 x1 | x1^-1,2   x1^2,1
----+---------+--------+
Initial state: x1
gap> H := FRGroupByVirtualEndomorphism(f:MealyElement);
<self-similar group over [ 1 .. 2 ] with 1 generator>
gap> Display(H.1);
   | 1     2
---+-----+-----+
 a | b,2   a,1
 b | b,1   b,2
---+-----+-----+
Initial state: a
```

7.1.10 TreeWreathProduct (FR group)

▷ TreeWreathProduct(*g*, *h*, *x0*, *y0*) (operation)

Returns: The tree-wreath product of groups *g*,*h*.

The tree-wreath product of two FR groups is a group generated by a copy of *g* and of *h*, in such a way that many conjugates of *g* commute.

More formally, assume without loss of generality that all generators of *g* are states of a machine m, and that all generators of *h* are states of a machine n. Then the tree-wreath product is generated by the images of generators of *g*,*h* in TreeWreathProduct(m,n,x0,y0).

For the operation on FR machines see TreeWreathProduct (3.5.8)). It is described (with small variations, and in lesser generality) in [Sid05]. For example, in

```
─────────────────────────── Example ───────────────────────────
gap> w := TreeWreathProduct(AddingGroup(2),AddingGroup(2),1,1);
<recursive group over [ 1 .. 4 ] with 2 generators>
gap> a := w.1; b := w.2;
<Mealy element on alphabet [ 1 .. 4 ] with 3 states>
<Mealy element on alphabet [ 1 .. 4 ] with 2 states>
gap> Order(a); Order(b);
infinity
```

```
      infinity
   gap> ForAll([-100..100],i->IsOne(Comm(a,a^(b^i))));
   true
```

the group w is the wreath product $Z \wr Z$.

7.1.11 WeaklyBranchedEmbedding

▷ WeaklyBranchedEmbedding(*g*)   (operation)

   Returns: A embedding of *g* in a weakly branched group.

   This function constructs a new FR group, on alphabet the square of the alphabet of *g*. It is generated by the canonical copy of *g* and by the tree-wreath product of *g* with an adding machine on the same alphabet as *g* (see TreeWreathProduct (7.1.10)). The function returns a group homomorphism into this new FR group.

   The main result of [SW03] is that the resulting group *h* is weakly branched. More precisely, h' contains $|X|^2$ copies of itself.   gap> f := WeaklyBranchedEmbedding(BabyAleshinGroup);; gap> Range(f); <recursive group over [1 .. 4] with 8 generators>   constructs a finitely generated branched group containing a free subgroup.

7.2 Operations for FR semigroups

7.2.1 PermGroup

▷ PermGroup(*g*, *l*)   (operation)
▷ EpimorphismPermGroup(*g*, *l*)   (operation)

   Returns: [An epimorphism to] the permutation group of *g*'s action on level *l*.

   The first function returns a permutation group on d^l points, where *d* is the size of *g*'s alphabet. It has as many generators as *g*, and represents the action of *g* on the *l*th layer of the tree.

   The second function returns a homomorphism from *g* to this permutation group.

```
 ──────────────────────────── Example ────────────────────────────
  gap> g := FRGroup("a=(1,2)","b=<a,>"); Size(g);
  <self-similar group over [ 1 .. 2 ] with 2 generators>
  8
  gap> PermGroup(g,2);
  Group([ (1,3)(2,4), (1,2) ])
  gap> PermGroup(g,3);
  Group([ (1,5)(2,6)(3,7)(4,8), (1,3)(2,4) ])
  gap> List([1..6],i->LogInt(Size(PermGroup(GrigorchukGroup,i)),2));
  [ 1, 3, 7, 12, 22, 42 ]
  gap> g := FRGroup("t=<,t>(1,2)"); Size(g);
  <self-similar group over [ 1 .. 2 ] with 1 generator>
  infinity
  gap> pi := EpimorphismPermGroup(g,5);
  MappingByFunction( <self-similar group over [ 1 .. 2 ] with 1 generator,
  of size infinity>, Group([ (1,17,9,25,5,21,13,29,3,19,11,27,7,23,15,31,
  2,18,10,26,6,22,14,30,4,20,12,28,8,24,16,32) ]), function( w ) ... end )
  gap> Order(g.1);
  infinity
  gap> Order(g.1^pi);
  32
```

7.2.2 PcGroup

▷ PcGroup(*g*, *l*) (operation)
▷ EpimorphismPcGroup(*g*, *l*) (operation)

Returns: [An epimorphism to] the pc group of *g*'s action on level *l*.

The first function returns a polycyclic group representing the action of *g* on the *l*th layer of the tree. It converts the permutation group PermGroup(g,l) to a Pc group, in which computations are often faster.

The second function returns a homomorphism from *g* to this pc group.

```
————————————————————— Example —————————————————————
gap> g := PcGroup(GrigorchukGroup,7); time;
<pc group with 5 generators>
3370
gap> NormalClosure(g,Group(g.3)); time;
<pc group with 79 generators>
240
gap> g := PermGroup(GrigorchukGroup,7); time;
<permutation group with 5 generators>
3
gap> NormalClosure(g,Group(g.3)); time;
<permutation group with 5 generators>
5344
gap> g := FRGroup("t=<,t>(1,2)"); Size(g);
<self-similar group over [ 1 .. 2 ] with 1 generator>
infinity
gap> pi := EpimorphismPcGroup(g,5);
MappingByFunction( <self-similar group over [ 1 .. 2 ] with
1 generator, of size infinity>, Group([ f1, f2, f3, f4, f5 ]), function( w ) ... end )
gap> Order(g.1);
infinity
gap> Order(g.1^pi);
32
```

7.2.3 TransformationMonoid

▷ TransformationMonoid(*g*, *l*) (operation)
▷ EpimorphismTransformationMonoid(*g*, *l*) (operation)

Returns: [An epimorphism to] the transformation monoid of *g*'s action on level *l*.

The first function returns a transformation monoid on d^l points, where *d* is the size of *g*'s alphabet. It has as many generators as *g*, and represents the action of *g* on the *l*th layer of the tree.

The second function returns a homomorphism from *g* to this transformation monoid.

```
————————————————————— Example —————————————————————
gap> i4 := SCMonoid(MealyMachine([[3,3],[1,2],[3,3]],[(1,2),[1,1],()]));
<self-similar monoid over [ 1 .. 2 ] with 3 generators>
gap> g := TransformationMonoid(i4,6);
<monoid with 3 generators>
gap> List([1..6],i->Size(TransformationMonoid(i4,i)));
[ 4, 14, 50, 170, 570, 1882 ]
gap> Collected(List(g,RankOfTransformation));
[ [ 1, 64 ], [ 2, 1280 ], [ 4, 384 ], [ 8, 112 ], [ 16, 32 ], [ 32, 8 ], [ 64, 2 ] ]
gap> pi := EpimorphismTransformationMonoid(i4,9);
```

```
  MappingByFunction( <self-similar monoid over [ 1 .. 2 ] with 3 generators>,
  <monoid with 3 generators>, function( w ) ... end )
gap> f := GeneratorsOfMonoid(i4){[1,2]};;
gap> for i in [1..10] do Add(f,f[i]*f[i+1]); od;
gap> Product(f{[3,5,7,9,11]})=f[11]*f[10];
false
gap> Product(f{[3,5,7,9,11]})^pi=(f[11]*f[10])^pi;
true
```

7.2.4 TransformationSemigroup

▷ TransformationSemigroup(g, l)   (operation)

▷ EpimorphismTransformationSemigroup(g, l)   (operation)

Returns: [An epimorphism to] the transformation semigroup of g's action on level l.

The first function returns a transformation semigroup on d^l points, where d is the size of g's alphabet. It has as many generators as g, and represents the action of g on the lth layer of the tree.

The second function returns a homomorphism from g to this transformation semigroup.

```
                              Example
gap> i2 := SCSemigroup(MealyMachine([[1,1],[2,1]],[(1,2),[2,2]]]));
<self-similar semigroup over [ 1 .. 2 ] with 2 generators>
gap> g := TransformationSemigroup(i2,6);
<semigroup with 2 generators>
gap> List([1..6],i->Size(TransformationSemigroup(i2,i)));
[ 4, 14, 42, 114, 290, 706 ]
gap> Collected(List(g,RankOfTransformation));
[ [ 1, 64 ], [ 2, 384 ], [ 4, 160 ], [ 8, 64 ], [ 16, 24 ], [ 32, 8 ], [ 64, 2 ] ]
gap> f0 := GeneratorsOfSemigroup(i2)[1];; f1 := GeneratorsOfSemigroup(i2)[2];;
gap> pi := EpimorphismTransformationSemigroup(i2,10);
MappingByFunction( <self-similar semigroup over [ 1 .. 2 ] with
2 generators>, <semigroup with 2 generators>, function( w ) ... end )
gap> (f1*(f1*f0)^10)=((f1*f0)^10);
false
gap> (f1*(f1*f0)^10)^pi=((f1*f0)^10)^pi;
true
```

7.2.5 EpimorphismGermGroup

▷ EpimorphismGermGroup(g, l)   (operation)

▷ EpimorphismGermGroup(g)   (operation)

Returns: A homomorphism to a polycyclic group.

This function returns an epimorphism to a polycyclic group, encoding the action on the first l levels of the tree and on the germs below. If l is omitted, it is assumed to be 0.

Since the elements of g are finite automata, they map periodic sequences to periodic sequences. The action on the periods, and in the immediate vicinity of them, is called the *germ action* of g. This function returns the natural homomorphism from g to the wreath product of this germ group with the quotient of g acting on the lth layer of the tree.

The germ group, by default, is abelian. If it is finite, this function returns a homomorphism to a Pc group; otherwise, a homomorphism to a polycyclic group.

The `GrigorchukTwistedTwin` (9.1.12) is, for now, the only example with a hand-coded, non-abelian germ group.

```
────────────────────── Example ──────────────────────
gap> EpimorphismGermGroup(GrigorchukGroup,0);
MappingByFunction( GrigorchukGroup, <pc group of size 4 with 2 generators>,
  function( g ) ... end )
gap> List(GeneratorsOfGroup(GrigorchukGroup),x->x^last);
[ <identity> of ..., f1, f1*f2, f2 ]
gap> StructureDescription(Image(last2));
"C2 x C2"
gap> g := FRGroup("t=<,t>(1,2)","m=<,m^-1>(1,2)");;
gap> EpimorphismGermGroup(g,0);
MappingByFunction( <state-closed, bounded group over [ 1, 2 ] with 2
  generators>, Pcp-group with orders [ 0, 0 ], function( x ) ... end )
gap> EpimorphismGermGroup(g,1);; Range(last); Image(last2);
Pcp-group with orders [ 2, 0, 0, 0, 0 ]
Pcp-group with orders [ 2, 0, 0, 0 ]
```

7.2.6 GermData

▷ GermData(*group*)   (attribute)
▷ GermValue(*element, data*)   (operation)

The first command computes some data useful to determine the germ value of a group element; the second command computes these germ values. For more information on germs, see `Germs` (5.2.24).

```
────────────────────── Example ──────────────────────
gap> data := GermData(GrigorchukGroup);
rec( endo := [ f1, f2 ] -> [ f1*f2, f1 ], group := <pc group of size 4 with 2 generators>,
  machines := [  ], map := [ <identity> of ..., f2, f1, f1*f2, <identity> of ... ],
  nucleus := [ <Trivial Mealy element on alphabet [ 1 .. 2 ]>, d, c, b, a ],
  nucleusmachine := <Mealy machine on alphabet [ 1 .. 2 ] with 5 states> )
gap> List(GeneratorsOfGroup(GrigorchukGroup),x->GermValue(x,data));
[ <identity> of ..., f1*f2, f1, f2 ]
```

7.2.7 StabilizerImage

▷ StabilizerImage(*g, v*)   (operation)
    Returns: The group of all states at *v* of elements of *g* fixing *v*.

This function constructs a new FR group, consisting of all states at vertex *v* (which can be an integer or a list) of the stabilizer of *v* in *g*.

The result is *g* itself precisely if *g* is recurrent (see `IsRecurrentFRSemigroup` (7.2.11)).

```
────────────────────── Example ──────────────────────
gap> G := FRGroup("t=<,t>(1,2)","u=<,u^-1>(1,2)","b=<u,t>");
<self-similar group over [ 1 .. 2 ] with 3 generators>
gap> Stabilizer(G,1);
<self-similar group over [ 1 .. 2 ] with 5 generators>
gap> GeneratorsOfGroup(last);
[ <2|u*t^-1>, <2|b>, <2|t^2>, <2|t*u>, <2|t*b*t^-1> ]
gap> StabilizerImage(G,1);
<self-similar group over [ 1 .. 2 ] with 5 generators>
```

```
gap> GeneratorsOfGroup(last);
[ <2|identity ...>, <2|u>, <2|t>, <2|u^-1>, <2|t> ]
```

7.2.8 LevelStabilizer

▷ LevelStabilizer(*g, n*) (operation)

Returns: The fixator of the *n*th level of the tree.

This function constructs the normal subgroup of *g* that fixes the *n*th level of the tree.

```
────────────────── Example ──────────────────
gap> G := FRGroup("t=<,t>(1,2)","a=(1,2)");
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> LevelStabilizer(G,2);
<self-similar group over [ 1 .. 2 ] with 9 generators>
gap> Index(G,last);
8
gap> IsNormal(G,last2);
true
```

7.2.9 IsStateClosed

▷ IsStateClosed(*g*) (property)

Returns: true if all states of elements of *g* belong to *g*.

This function tests whether *g* is a *state-closed* group, i.e. a group such that all states of all elements of *g* belong to *g*.

The smallest state-closed group containing *g* is computed with StateClosure (7.2.10).

```
────────────────── Example ──────────────────
gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> AssignGeneratorVariables(Dinfinity);
#I  Assigned the global variables [ a, b ]
gap> IsStateClosed(Group(a));
     IsStateClosed(Group(b));
     IsStateClosed(Dinfinity);
true
false
true
```

7.2.10 StateClosure

▷ StateClosure(*g*) (operation)

Returns: The smallest state-closed group containing *g*.

This function computes the smallest group containing all states of all elements of *g*, i.e. the smallest group containing *g* and for which IsStateClosed (7.2.9) returns true.

```
────────────────── Example ──────────────────
gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> AssignGeneratorVariables(Dinfinity);
#I  Assigned the global variables [ a, b ]
gap> StateStateClosure(Group(a))=Dinfinity; StateClosure(Group(b))=Dinfinity;
```

```
  false
  true
```

7.2.11   IsRecurrentFRSemigroup

▷ IsRecurrentFRSemigroup(*g*)   (property)

Returns: true if *g* is a recurrent group.

This function returns true if *g* is a *recurrent* group, i.e. if, for every vertex v, all elements of *g* appear as states at v of elements fixing v.

```
 ──────────────────── Example ────────────────────
  gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
  <self-similar group over [ 1 .. 2 ] with 2 generators>
  gap> AssignGeneratorVariables(Dinfinity);
  #I  Assigned the global variables [ a, b ]
  gap> IsRecurrentFRSemigroup(Group(a)); IsRecurrentFRSemigroup(Group(b));
  false
  false
  gap> IsRecurrentFRSemigroup(Dinfinity);
  true
```

7.2.12   IsLevelTransitiveFRGroup

▷ IsLevelTransitiveFRGroup(*g*)   (property)

Returns: true if *g* is a level-transitive group.

This function returns true if *g* is a *level-transitive* group, i.e. if the action of *g* is transitive at every level of the tree on which it acts.

This function can be abbreviated as IsLevelTransitive.

```
 ──────────────────── Example ────────────────────
  gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
  <self-similar group over [ 1 .. 2 ] with 2 generators>
  gap> AssignGeneratorVariables(Dinfinity);
  #I  Assigned the global variables [ a, b ]
  gap> IsLevelTransitiveFRGroup(Group(a)); IsLevelTransitiveFRGroup(Group(b));
       IsLevelTransitiveFRGroup(Dinfinity);
  false
  false
  true
```

7.2.13   IsInfinitelyTransitive

▷ IsInfinitelyTransitive(*g*)   (property)
▷ IsLevelTransitiveOnPatterns(*g*)   (property)

Returns: true if *g* is infinitely transitive.

This function returns true if *g* is an *infinitely transitive* group. This means that *g* is the state-closed group of a bireversible Mealy machine (see IsBireversible (5.2.7)), and that the action of the set of reduced words of any given length over the alphabet (where "reduced" means no successive letters related by the involution) is transitive.

Reduced words are defined as follows: if the underlying Mealy machine of *g* has an involution on its alphabet (see `AlphabetInvolution` (5.2.6)), then reduced words are words in which two consecutive letters are not images of each other under the involution. If no involution is defined, then all words are considered reduced; the command then becomes synonymous to `IsLevelTransitiveFRGroup` (7.2.12).

This notion is of fundamental importance for the study of lattices in a product of trees; it implies under appropriate circumstances that the dual group is free.

```
─────────────── Example ───────────────
gap> IsInfinitelyTransitive(BabyAleshinGroup);
true
gap> IsLevelTransitiveFRGroup(BabyAleshinGroup);
true
gap> s := DualMachine(BabyAleshinMachine);
<Mealy machine on alphabet [ 1 .. 3 ] with 2 states>
gap> AlphabetInvolution(s); # set attribute
[ 1, 2, 3 ]
gap> g := SCGroup(s);
<state-closed group over [ 1 .. 3 ] with 2 generators>
gap> IsInfinitelyTransitive(g);
true
gap> IsLevelTransitiveFRGroup(g);
false
```

7.2.14 IsFinitaryFRSemigroup

▷ IsFinitaryFRSemigroup(*g*)   (property)
▷ IsWeaklyFinitaryFRSemigroup(*g*)   (property)
▷ IsBoundedFRSemigroup(*g*)   (property)
▷ IsPolynomialGrowthFRSemigroup(*g*)   (property)
▷ IsFiniteStateFRSemigroup(*g*)   (property)

Returns: true if all elements of *g* have the required property.

This function returns true if all elements of *g* have the required property, as FR elements; see `IsFinitaryFRElement` (5.2.10), `IsWeaklyFinitaryFRElement` (5.2.23), `IsBoundedFRElement` (5.2.12), `IsPolynomialGrowthFRElement` (5.2.13) and `IsFiniteStateFRElement` (4.2.12).

```
─────────────── Example ───────────────
gap> G := FRGroup("a=(1,2)","b=<a,b>","c=<c,b>","d=<d,d>(1,2)");
<self-similar group over [ 1 .. 2 ] with 4 generators>
gap> L := [Group(G.1),Group(G.1,G.2),Group(G.1,G.2,G.3),G];;
gap> List(L,IsFinitaryFRSemigroup);
[ true, false, false, false ]
gap> List(L,IsBoundedFRSemigroup);
[ true, true, false, false ]
gap> List(L,IsPolynomialGrowthFRSemigroup);
[ true, true, true, false ]
gap> List(L,IsFiniteStateFRSemigroup);
[ true, true, true, true ]
```

7.2.15 Degree (FR semigroup)

▷ Degree(g)   (attribute)
▷ DegreeOfFRSemigroup(g)   (attribute)
▷ Depth(g)   (attribute)
▷ DepthOfFRSemigroup(g)   (attribute)

Returns: The maximal degree/depth of elements of g.

This function returns the maximal degree/depth of elements of g; see Degree (5.2.9) and Depth (5.2.11).

```
───────────────── Example ─────────────────
gap> G := FRGroup("a=(1,2)","b=<a,b>","c=<c,b>");
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> Degree(Group(G.1)); Degree(Group(G.1,G.2)); Degree(G);
0
1
2
gap> Depth(Group(G.1)); Depth(Group(G.1,G.2)); Depth(G);
1
infinity
infinity
```

7.2.16 HasOpenSetConditionFRSemigroup

▷ HasOpenSetConditionFRSemigroup(g)   (property)

Returns: true if g has the open set condition.

This function returns true if all elements of g have the *open set condition*, see HasOpenSetConditionFRElement (5.2.25).

```
───────────────── Example ─────────────────
gap> HasOpenSetConditionFRSemigroup(GrigorchukGroup);
false
gap> HasOpenSetConditionFRSemigroup(BinaryAddingGroup);
true
```

7.2.17 HasCongruenceProperty

▷ HasCongruenceProperty(G)   (property)

Returns: true if G has the congruence property.

This function returns true if the transformation (semi)group G has the *congruence property*, namely if every homomorphism $G \to Q$ to a finite quotient factors as $G \to H \to Q$ via an action of G on a finite set.

This command is not guaranteed to terminate.

```
───────────────── Example ─────────────────
gap> HasCongruenceProperty(GrigorchukGroup);
true
gap> HasCongruenceProperty(GrigorchukTwistedTwin);
...runs forever...
```

7.2.18 IsContracting

▷ IsContracting(*g*) (property)

    Returns: true if *g* is a contracting semigroup.

    This function returns true if *g* is a *contracting* semigroup, i.e. if there exists a finite subset N of *g* such that the LimitStates (4.2.11) of every element of *g* belong to N.

    The minimal such N can be computed with NucleusOfFRSemigroup (7.2.19).

```
                                 Example
  gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
  <self-similar group over [ 1 .. 2 ] with 2 generators>
  gap> IsContracting(Dinfinity);
  true
```

7.2.19 NucleusOfFRSemigroup

▷ NucleusOfFRSemigroup(*g*) (attribute)
▷ Nucleus(*g*) (operation)

    Returns: The nucleus of the contracting semigroup *g*.

    This function returns the *nucleus* of the contracting semigroup *g*, i.e. the smallest subset N of *g* such that the LimitStates (4.2.11) of every element of *g* belong to N.

    This function returns fail if no such N exists. Usually, it loops forever without being able to decide whether N is finite or infinite. It succeeds precisely when IsContracting(g) succeeds.

```
                                 Example
  gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
  <self-similar group over [ 1 .. 2 ] with 2 generators>
  gap> NucleusOfFRSemigroup(Dinfinity);
  [ <2|identity ...>, <2|b>, <2|a> ]
```

7.2.20 NucleusMachine (FR semigroup)

▷ NucleusMachine(*g*) (attribute)

    Returns: The nucleus machine of the contracting semigroup *g*.

    This function returns the *nucleus* of the contracting semigroup *g*, see NucleusOfFRSemigroup (7.2.19), in the form of a Mealy machine.

    Since all states of the nucleus are elements of the nucleus, the transition and output function may be restricted to the nucleus, defining a Mealy machine. Finitely generated recurrent groups are generated by their nucleus machine.

    This function returns fail if no such n exists. Usually, it loops forever without being able to decide whether n is finite or infinite. It succeeds precisely when IsContracting(g) succeeds.

```
                                 Example
  gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>");
  <self-similar group over [ 1 .. 2 ] with 2 generators>
  gap> M := NucleusMachine(Dinfinity);
  <Mealy machine on alphabet [ 1, 2 ] with 3 states>
  gap> Display(M);
     |  1      2
  ---+-----+-----+
   a | a,1    a,2
   b | c,1    b,2
```

```
  c | a,2   a,1
---+-----+-----+
gap> Dinfinity=SCGroup(M);
true
```

7.2.21 AdjacencyBasesWithOne

▷ AdjacencyBasesWithOne(*g*)   (attribute)

▷ AdjacencyPoset(*g*)   (attribute)

Returns: The bases, or the poset, of the simplicial model of *g*.

For these arguments, *g* can be either the nucleus of an FR semigroup, or that semigroup itself, in which case its nucleus is first computed.

The first function computes those maximal (for inclusion) subsets of the nucleus that are recurrent, namely subsets B such that Set(B,x->States(x,v))=B for a string v.

The second function then computes the poset of intersections of these bases, and returns it as a binary relation.

For more details on these concepts, see [Nek08a].

```
─────────────── Example ───────────────
gap> n := NucleusOfFRSemigroup(BasilicaGroup);
[ <Trivial Mealy element on alphabet [ 1 .. 2 ]>, b,
  <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
  <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
  <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
  <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
  <Mealy element on alphabet [ 1 .. 2 ] with 3 states> ]
gap> AdjacencyBasesWithOne(n);
[ [ <Trivial Mealy element on alphabet [ 1 .. 2 ]>,
      <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
      <Mealy element on alphabet [ 1 .. 2 ] with 3 states> ],
  [ <Trivial Mealy element on alphabet [ 1 .. 2 ]>,
      <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
      <Mealy element on alphabet [ 1 .. 2 ] with 3 states> ],
  [ <Trivial Mealy element on alphabet [ 1 .. 2 ]>,
      <Mealy element on alphabet [ 1 .. 2 ] with 3 states>,
      <Mealy element on alphabet [ 1 .. 2 ] with 3 states> ] ]
gap> AdjacencyPoset(n);
<general mapping: <object> -> <object> >
gap> Draw(HasseDiagramBinaryRelation(last));
```
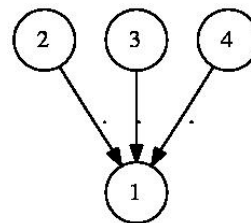

This produces (in a new window) the following picture:

7.2.22 BranchingSubgroup

▷ BranchingSubgroup(*g*)   (operation)

Returns: A branching subgroup of *g*.

This function searches for a subgroup k of g, such that k contains $k \times \cdots \times k$.

It searches for elements in larger and larger balls in g, calling FindBranchingSubgroup (7.2.23).

```
————————————— Example —————————————
gap> K := BranchingSubgroup(GrigorchukGroup);
<self-similar group over [ 1 .. 2 ] with 9 generators>
gap> IsBranchingSubgroup(K);
true
gap> IsBranched(GrigorchukGroup);
true
gap> Index(GrigorchukGroup,K);
16
```

7.2.23 FindBranchingSubgroup

▷ FindBranchingSubgroup(*g*, *l*, *r*)   (operation)

Returns: A branching subgroup of g.

This function searches for a subgroup k of g, such that k contains $k \times \cdots \times k$.

The second argument `l` specifies the level at which branching must occur; i.e. asks to search for a subgroup k such that g contains k^{d^l} where d is the size of the alphabet. If `l=infinity`, the resulting k will be a regularly branched subgroup.

The third argument `r` specifies the radius to explore in g and all branching subgroups at levels smaller than `l` for elements with all level-1 states trivial except one.

```
————————————— Example —————————————
gap> FindBranchingSubgroup(GrigorchukGroup,1,4);
<self-similar group over [ 1 .. 2 ] with 8 generators>
gap> Index(GrigorchukGroup,last);
8
gap> FindBranchingSubgroup(GrigorchukGroup,2,4);
<self-similar group over [ 1 .. 2 ] with 6 generators>
gap> Index(GrigorchukGroup,last);
16
gap> FindBranchingSubgroup(GrigorchukGroup,3,4);
<self-similar group over [ 1 .. 2 ] with 9 generators>
gap> Index(GrigorchukGroup,last);
16
```

7.2.24 IsBranched (FR group)

▷ IsBranched(*g*)   (property)

Returns: true if g has a finite-index branching subgroup.

This function returns true if g has a finite-index subgroup k, such that k contains $k \times \cdots \times k$.

```
————————————— Example —————————————
<Example><![CDATA[
gap> K := BranchingSubgroup(GrigorchukGroup);
<self-similar group over [ 1 .. 2 ] with 9 generators>
gap> IsBranchingSubgroup(K);
true
gap> IsBranched(GrigorchukGroup);
true
```

```
gap> Index(GrigorchukGroup,K);
16
```

7.2.25 IsBranchingSubgroup (FR semigroup)

▷ IsBranchingSubgroup(k)   (property)

    Returns: true if k is a branching subgroup.

    This function returns true if k contains $k \times \cdots \times k$.

─────────────────────── Example ───────────────────────
```
gap> K := BranchingSubgroup(GrigorchukGroup);
<self-similar group over [ 1 .. 2 ] with 9 generators>
gap> IsBranchingSubgroup(K);
true
gap> IsBranched(GrigorchukGroup);
true
gap> Index(GrigorchukGroup,K);
16
```

7.2.26 BranchStructure

▷ BranchStructure(G)   (attribute)

    Returns: A record describing the branching of G.

    This function constructs a record with fields group,quo,set,top,wreath,epi giving respectively a group isomorphic to G/K, the quotient map from G to it, the alphabet of G, the group of permutations of the alphabet, the wreath product of group with its top permutations, and an epimorphism from a subgroup of wreath to group.

    This information is used as essential data on the branch group, and are used to construct e.g. its Zeta function.

7.2.27 TopVertexTransformations

▷ TopVertexTransformations(g)   (attribute)

    Returns: The transformations at the root under the action of g.

    This function returns the permutation group, or the transformation group/semigroup/monoid, of all activities of all elements under the root vertex of the tree on which g acts.

    It is a synonym for PermGroup(g,1) or TransformationMonoid(g,1) or TransformationSemigroup(g,1).

─────────────────────── Example ───────────────────────
```
gap> TopVertexTransformations(GrigorchukGroup);
Group([ (), (1,2) ])
gap> IsTransitive(last,AlphabetOfFRSemigroup(GrigorchukGroup));
true
gap> TopVertexTransformations(FullSCMonoid([1..3]));
<monoid with 3 generators>
gap> Size(last);
27
```

7.2.28 VertexTransformations (FR semigroup)

▷ VertexTransformations(*g*)   (attribute)

    Returns: The transformations at all vertices under the action of *g*.

    This function returns the permutation group, or the transformation group/monoid/semigroup, of all activities of all elements under all vertices of the tree on which *g* acts.

    This is the smallest group/monoid/semigroup P such that *g* is a subset of `FullSCGroup(AlphabetOfFRSemigroup(g),P)` or `FullSCMonoid(AlphabetOfFRSemigroup(g),P)` or `FullSCSemigroup(AlphabetOfFRSemigroup(g),P)`.

```
───────────────────── Example ─────────────────────
gap> VertexTransformations(GuptaSidkiGroup);
Group([ (), (1,2,3), (1,3,2) ])
gap> TopVertexTransformations(Group(GuptaSidkiGroup.2));
Group(())
gap> VertexTransformations(Group(GuptaSidkiGroup.2));
Group([ (), (1,2,3), (1,3,2) ])
```

7.2.29 VirtualEndomorphism

▷ VirtualEndomorphism(*g/m, v*)   (operation)

    Returns: The virtual endomorphism at vertex *v*.

    This function returns the homomorphism from `Stabilizer(g,v)` to *g*, defined by computing the state at *v*. It is loosely speaking an inverse of `FRGroupByVirtualEndomorphism` (7.1.9).

    The first argument *m* may also be an FR machine.

```
───────────────────── Example ─────────────────────
gap> A := SCGroup(MealyMachine([[2,1],[2,2]],[(1,2),()]));
<self-similar group over [ 1 .. 2 ] with 1 generator>
gap> f := VirtualEndomorphism(A,1);
MappingByFunction( <self-similar group over [ 1 .. 2 ] with
1 generator>, <self-similar group over [ 1 .. 2 ] with
1 generator>, function( g ) ... end )
gap> ((A.1)^2)^f=A.1;
true
gap> B := FRGroupByVirtualEndomorphism(f);
<self-similar group over [ 1 .. 2 ] with 1 generator>
gap> A=B;
true
```

7.2.30 EpimorphismFromFpGroup

▷ EpimorphismFromFpGroup(*g, l*)   (operation)

    Returns: An epimorphism from a finitely presented group to *g*.

    For some examples of self-similar groups, a recursive presentation of the group is coded into FR, and an approximate presentation is returned by this command, together with a map onto the group *g*. The argument *l* roughly means the number of iterates of an endomorphism were applied to a finite set of relators. An isomorphic group would be obtained by setting `l=infinity`; for that purpose, see `IsomorphismSubgroupFpGroup` (7.2.31).

    Preimages can be computed, with `PreImagesRepresentative`. They are usually reasonably short words, though by no means guaranteed to be of minimal length.

Currently this command is implemented through an ad hoc method for `BinaryKneadingGroup` (9.1.2), `GrigorchukGroup` (9.1.10) and `GrigorchukOverGroup` (9.1.11).

```
――――――――――――――――――――――― Example ―――――――――――――――――――――――
gap> f := EpimorphismFromFpGroup(GrigorchukGroup,1);
MappingByFunction( <fp group on the generators
[ a, b, c, d ]>, GrigorchukGroup, function( w ) ... end )
4 gap> RelatorsOfFpGroup(Source(f));
[ a^2, b^2, c^2, d^2, b*c*d, a*d*a*d*a*d*a*d, a^-1*c*a*c*a^-1*c*a*c*a^-1*c*a*c*a^
    -1*c*a*c, a^-1*c^-1*a*b*a^-1*c*a*b*a^-1*c^-1*a*b*a^-1*c*a*b*a^-1*c^-1*a*b*a^-1*
    c*a*b*a^-1*c^-1*a*b*a^-1*c*a*b, a*d*a*c*a*c*a*d*a*c*a*c*a*d*a*c*a*c*a*d*a*c*a*c,
  a^-1*c*a*c*a^-1*c*a*b*a^-1*c*a*b*a^-1*c*a*c*a^-1*c*a*b*a^-1*c*a*b*a^-1*c*a*c*a^
    -1*c*a*b*a^-1*c*a*b*a^-1*c*a*c*a^-1*c*a*b*a^-1*c*a*b ]
gap> PreImagesRepresentative(f,Comm(GrigorchukGroup.1,GrigorchukGroup.2));
a*c*a*d*a*d*a*c
gap> Source(f).4^f=GrigorchukGroup.4;
true
```

7.2.31 IsomorphismSubgroupFpGroup

▷ IsomorphismSubgroupFpGroup(*g*)   (operation)
▷ AsSubgroupFpGroup(*g*)   (operation)
▷ IsomorphismLpGroup(*g*)   (operation)
▷ AsLpGroup(*g*)   (operation)

Returns: An isomorphism to a subgroup of a finitely presented group, or an L-presented group.

For some examples of self-similar groups, a recursive presentation of the group is coded into FR, and is returned by this command. The group *g* itself sits as a subgroup of a finitely presented group. To obtain a finitely presented group approximating *g*, see `EpimorphismFromFpGroup` (7.2.30). PreImages can also be computed; it is usually better to use `PreImageElm`, since the word problem may not be solvable by GAP in the f.p. group.

Currently this command is implemented through an ad hoc method for `BinaryKneadingGroup` (9.1.2), `GrigorchukGroup` (9.1.10), `GrigorchukOverGroup` (9.1.11), generalized `GuptaSidkiGroups` (9.1.18) and generalized `FabrykowskiGuptaGroups` (9.1.21).

The second form returns an isomorphism to an L-presented group (see [Bar03a] and [BEH08]). It requires the package NQL.

```
――――――――――――――――――――――― Example ―――――――――――――――――――――――
gap> f := IsomorphismSubgroupFpGroup(BasilicaGroup);
MappingByFunction( BasilicaGroup, Group([ a^-1, a*t^-1*a^-1*t*a^-1
 ]), function( g ) ... end, function( w ) ... end )
gap> Range(f);
Group([ a^-1, a*t^-1*a^-1*t*a^-1 ])
gap> c := Comm(BasilicaGroup.1,BasilicaGroup.2);
<Mealy element on alphabet [ 1, 2 ] with 9 states, initial state 1>
gap> c^f;
t^-2*a*t^-1*a*t*a^-2*t*a*t^-2*a*t^-1*a*t*a^-1*t*a*t^-1*a*t^-2*
a^-1*t*a*t^-1*a*t^-1*a^-1*t*a^-1*t^5*a*t^-1*a^-1*t*a^-1
gap> PreImageElm(f,last);
<Mealy element on alphabet [ 1, 2 ] with 9 states, initial state 1>
gap> last=c;
true
```

7.3 Properties for infinite groups

7.3.1 IsTorsionGroup

▷ IsTorsionGroup(g)   (property)

    Returns: `true` if g is a torsion group.

    This function returns `true` if g is a torsion group, i.e. if every element in g has finite order; and `false` if g contains an element of infinite order.

    This method is quite rudimentary, and is not guaranteed to terminate. At the minimum, g should be a group in which `Order()` succeeds in computing element orders; e.g. a bounded group in Mealy machine format.

```
——————————————————— Example ———————————————————
gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>":IsMealyElement);
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> IsTorsionGroup(Dinfinity);
false
gap> IsTorsionGroup(GrigorchukGroup); IsTorsionGroup(GuptaSidkiGroup);
true
true
gap> IsTorsionGroup(FabrykowskiGuptaGroup);
false
```

7.3.2 IsTorsionFreeGroup

▷ IsTorsionFreeGroup(g)   (property)

    Returns: `true` if g is a torsion-free group.

    This function returns `true` if g is a torsion-free group, i.e. if no element in g has finite order; and `false` if g contains an element of finite order.

    This method is quite rudimentary, and is not guaranteed to terminate. At the minimum, g should be a group in which `Order()` succeeds in computing element orders; e.g. a bounded group in Mealy machine format.

```
——————————————————— Example ———————————————————
gap> Dinfinity := FRGroup("a=(1,2)","b=<a,b>":IsMealyElement);
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> IsTorsionFreeGroup(Dinfinity);
false
gap> IsTorsionFreeGroup(BasilicaGroup);
true
```

7.3.3 IsAmenableGroup

▷ IsAmenableGroup(g)   (property)

    Returns: `true` if g is an amenable group.

    Amenable groups, introduced by von Neumann [vN29], are those groups that admit a finitely additive, translation-invariant measure.

```
——————————————————— Example ———————————————————
gap> IsAmenableGroup(FreeGroup(2));
false
gap> IsAmenableGroup(BasilicaGroup);
true
```

7.3.4 IsVirtuallySimpleGroup

▷ IsVirtuallySimpleGroup(*g*)   (property)

▷ LambdaElementVHGroup(*g*)   (attribute)

    Returns: true if *g* admits a finite-index simple subgroup.

    This function attempts to prove that the VH group *g* admits a finite-index simple subgroup.

    It is based on the following test: let D be a direction (vertical or horizontal) such that the corresponding action is infinitely transitive (see IsInfinitelyTransitive (7.2.13)). If the corresponding subgroup of *g* contains a non-trivial element λ that acts trivially in the corresponding action, then every normal subgroup contains λ. It then remains to check that the normal closure of λ has finite index. This element λ is then stored as the attribute LambdaElementVHGroup(g).

    The current implementation is based on results in [BM00a] and [BM00b], and does not work for the Rattaggi examples (see RattaggiGroup (9.1.24)).

7.3.5 IsResiduallyFinite

▷ IsResiduallyFinite(*obj*)   (property)

    Returns: true if *obj* is residually finite.

    An object is *residually finite* if it can be approximated arbitrarily well by finite quotients; i.e. if for every $g \neq h \in X$ there exists a finite quotient $\pi : X \to Q$ such that $g^\pi \neq h^\pi$.

```
——————————— Example ———————————
  gap> IsResiduallyFinite(FreeGroup(2));
  true
  gap> IsResiduallyFinite(BasilicaGroup);
  true
```

7.3.6 IsSQUniversal

▷ IsSQUniversal(*obj*)   (property)

    Returns: true if *obj* is SQ-universal.

    An object *obj* is *SQ-universal* if every countable object of the same category as *obj* is a subobject of a quotient of *obj*.

```
——————————— Example ———————————
  gap> IsSQUniversal(FreeGroup(2));
  true
```

7.3.7 IsJustInfinite

▷ IsJustInfinite(*obj*)   (property)

    Returns: true if *obj* is just-infinite.

    An object *obj* is *just-infinite* if *obj* is infinite, but every quotient of *obj* is finite.

```
——————————— Example ———————————
  gap> IsJustInfinite(FreeGroup(2));
  false
  gap> IsJustInfinite(FreeGroup(1));
  true
  gap> IsJustInfinite(GrigorchukGroup); time
  true
  8284
```

Chapter 8

Algebras

Self-similar algebras and algebras with one (below *FR algebras*) are simply algebras [with one] whose elements are linear FR machines. They naturally act on the alphabet of their elements, which is a vector space.

Elements may be added, subtracted and multiplied. They can be vector or algebra linear elements; the vector elements are in general preferable, for efficiency reasons.

Finite-dimensional approximations of self-similar algebras can be computed; they are given as matrix algebras.

8.1 Creators for FR algebras

The most straightforward creation method for FR algebras is `Algebra()`, applied with linear FR elements as arguments. There are shortcuts to this somewhat tedious method:

8.1.1 FRAlgebra

▷ FRAlgebra(*ring, {definition, }*)   (operation)

▷ FRAlgebraWithOne(*ring, {definition, }*)   (operation)

Returns: A new self-similar algebra [with one].

This function constructs a new FR algebra [with one], generated by linear FR elements. It receives as argument any number of strings, each of which represents a generator of the object to be constructed.

ring is the acting domain of the vector space on which the algebra will act.

Each *definition* is of the form `"name=[[...],...,[...]]"` or of the form `"name=[[...],...,[...]]:out"`, namely a matrix whose entries are algebraic expressions in the `names`, possibly using `0,1`, optionally followed by a scalar. The matrix entries specify the decomposition of the element being defined, and the optional scalar specifies the output of that element, by default assumed to be one.

The option `IsVectorElement` asks for the resulting algebra to be generated by vector elements, see example below. The generators must of course be finite-state.

```
───────────────────────── Example ─────────────────────────
  gap> m := FRAlgebra(Rationals,"a=[[1,a],[a,0]]");
  <self-similar algebra on alphabet Rationals^2 with 1 generator>
  gap> Display(Activity(m.1,2));
  [ [  1,  0,  1,  1 ],
    [  0,  1,  1,  0 ],
```

```
   [  1,  1,  0,  0 ],
   [  1,  0,  0,  0 ] ]
gap> m2 := FRAlgebra(Rationals,"a=[[1,a],[a,0]]":IsVectorElement);;
gap> m.1=m2.1;
true
```

8.1.2 SCAlgebra

▷ SCAlgebra(*m*)   (operation)
▷ SCLieAlgebra(*m*)   (operation)
▷ SCAlgebraWithOne(*m*)   (operation)
▷ SCAlgebraNC(*m*)   (operation)
▷ SCAlgebraWithOneNC(*m*)   (operation)
     Returns: The state-closed algebra [with one] generated by the machine *m*.

     This function constructs a new FR algebra [with one] a, generated by all the states of the FR machine *m*. There is a bijective correspondence between GeneratorsOfFRMachine(m) and the generators of a, which is accessible via Correspondence(a) (See Correspondence (7.1.4)); it is a homomorphism from the stateset of *m* to a, or a list indicating for each state of *m* a corresponding generator index in the generators of a (with 0 for identity).

     In the non-NC forms, redundant (equal, zero or one) states are removed from the generating set of a.

```
                                  Example
  gap> a := SCAlgebra(AsLinearMachine(Rationals,I4Machine));
  <self-similar algebra on alphabet Rationals^2 with 3 generators>
  gap> a.1 = AsLinearElement(Rationals,I4Monoid.1);
  true
```

8.1.3 NucleusOfFRAlgebra

▷ NucleusOfFRAlgebra(*a*)   (attribute)
▷ Nucleus(*a*)   (operation)
     Returns: The nucleus of the contracting algebra a.

     This function returns the *nucleus* of the contracting algebra a, i.e. the smallest subspace N of a such that the LimitStates (4.2.11) of every element of a belong to N.

     This function returns fail if no such N exists. Usually, it loops forever without being able to decide whether N is finite or infinite. It succeeds precisely when IsContracting(g) succeeds.

```
                                  Example
  gap> > a := GrigorchukThinnedAlgebra(2);
  <self-similar algebra-with-one on alphabet GF(2)^2 with 4 generators, of dimension infinity>
  gap> NucleusOfFRAlgebra(a);
  <vector space over GF(2), with 4 generators>
```

8.1.4 BranchingIdeal

▷ BranchingIdeal(*A*)   (operation)
     Returns: An ideal *I* that contains matrices over itself.

```
───────── Example ─────────
gap> R := GrigorchukThinnedAlgebra(2);;
gap> I := BranchingIdeal(R);
<two-sided ideal in <self-similar algebra-with-one on alphabet GF(2)^2
 with 4 generators, of dimension infinity>, (3 generators)>
gap> e := EpimorphismMatrixQuotient(R,3);;
gap> eI := Ideal(Range(e),List(GeneratorsOfIdeal(I),x->x^e));
<two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (3 generators)>
gap> Dimension(Range(e)/eI);
6
```

8.2 Operations for FR algebras

8.2.1 MatrixQuotient

▷ MatrixQuotient(*a*, *l*)   (operation)
▷ EpimorphismMatrixQuotient(*a*, *l*)   (operation)
   Returns: The matrix algebra of *a*'s action on level *l*.
   The first function returns the matrix algebra generated by the activities of *a* on level *l* (see the examples in 6.1.7). The second functon returns an algebra homomorphism from *a* to the matrix algebra.

```
───────── Example ─────────
gap> a := ThinnedAlgebraWithOne(GF(2),GrigorchukGroup);
<self-similar algebra-with-one on alphabet GF(2)^2 with 4 generators>
gap> List([0..4],i->Dimension(MatrixQuotient(a,i)));
[ 1, 2, 6, 22, 78 ]
```

8.2.2 ThinnedAlgebra

▷ ThinnedAlgebra(*r*, *g*)   (operation)
▷ ThinnedAlgebraWithOne(*r*, *g*)   (operation)
   Returns: The thinned algebra [with one] associated with *g*.
   The first function returns the thinned algebra of a FR group/monoid/semigroup *g*, over the domain *r*. This is the linear envelope of *g* in its natural action on sequences.
   The embedding of *g* in its thinned algebra is returned by Embedding(g,a).

```
───────── Example ─────────
gap> a := ThinnedAlgebraWithOne(GF(2),GrigorchukGroup);
<self-similar algebra on alphabet GF(2)^2 with 5 generators>
gap> a.1 = GrigorchukGroup.1^Embedding(GrigorchukGroup,a);
true
gap> Dimension(VectorSpace(GF(2),[One(a),a.2,a.3,a.4]));
3
```

8.2.3 Nillity

▷ Nillity(*x*)   (attribute)
▷ IsNilElement(*x*)   (property)
   Returns: The smallest n such that $x^n = 0$.

The first command computes the nillity of x, i.e. the smallest n such that $x^n = 0$. The command is not guaranteed to terminate.

The second command returns whether x is nil, that is, whether its nillity is finite.

8.2.4 DegreeOfHomogeneousElement

▷ DegreeOfHomogeneousElement(x) (attribute)

▷ IsHomogeneousElement(x) (property)

Returns: The degree of x in its parent.

If x belongs to a graded algebra A, then the second command returns whether x belongs to a homogeneous component of Grading(A), and the first command returns the degree of that component (or fail if no such component exists).

Chapter 9

Examples

FR predefines a large collection of machines and groups. The groups are, whenever possible, defined as state closures of corresponding Mealy machines.

9.1 Examples of groups

9.1.1 FullBinaryGroup

▷ FullBinaryGroup (global variable)
▷ FiniteDepthBinaryGroup(*l*) (function)
▷ FinitaryBinaryGroup (global variable)
▷ BoundedBinaryGroup (global variable)
▷ PolynomialGrowthBinaryGroup (global variable)
▷ FiniteStateBinaryGroup (global variable)

These are the finitary, bounded, polynomial-growth, finite-state, or unrestricted groups acting on the binary tree. They are respectively shortcuts for `FullSCGroup([1..2])`, `FullSCGroup([1..2],1)`, `FullSCGroup([1..2],IsFinitaryFRSemigroup)`, `FullSCGroup([1..2],IsBoundedFRSemigroup)`, `FullSCGroup([1..2],IsPolynomialGrowthFRSemigroup)`, and `FullSCGroup([1..2],IsFiniteStateFRSemigroup)`.

They may be used to draw random elements, or to test membership.

9.1.2 BinaryKneadingGroup

▷ BinaryKneadingGroup(*angle/ks*) (function)
▷ BinaryKneadingMachine(*angle/ks*) (function)
  Returns: The [machine generating the] iterated monodromy group of a quadratic polynomial.
  The first function constructs a Mealy machine whose state closure is the binary kneading group.

The second function constructs a new FR group `G`, which is the iterated monodromy group of a quadratic polynomial, either specified by its angle or by its kneading sequence(s).

If the argument is a (rational) angle, the attribute `Correspondence(G)` is a function returning, for any rational, the corresponding generator of `G`.

If there is one argument, which is a list or a string, it is treated as the kneading sequence of a periodic (superattracting) polynomial. The sequence is implicity assumed to end by '*'. The attribute `Correspondence(G)` is a list of the generators of `G`.

If there are two arguments, which are lists or strings, they are treated as the preperiod and period of the kneading sequence of a preperiodic polynomial. The last symbol of the arguments must differ. The attribute `Correspondence(G)` is a pair of lists of generators; `Correspondence(G)[1]` is the preperiod, and `Correspondence(G)[2]` is the period. The attribute `KneadingSequence(G)` returns the kneading sequence, as a pair of strings representing preperiod and period respectively.

As particular examples, `BinaryKneadingMachine()` is the adding machine; `BinaryKneadingGroup()` is the adding machine; and `BinaryKneadingGroup("1")` is `BasilicaGroup` (9.1.3).

```
———————————————————— Example ————————————————————
gap> BinaryKneadingGroup()=AddingGroup(2);
true
gap> BinaryKneadingGroup(1/3)=BasilicaGroup;
true
gap> g := BinaryKneadingGroup([0,1],[0]);
BinaryKneadingGroup("01","0")
gap> ForAll(Correspondence(g)[1],IsFinitaryFRElement);
true
gap> ForAll(Correspondence(g)[2],IsFinitaryFRElement);
false
gap> ForAll(Correspondence(g)[2],IsBoundedFRElement);
true
```

9.1.3 BasilicaGroup

▷ BasilicaGroup (global variable)

The *Basilica group*. This is a shortcut for `BinaryKneadingGroup("1")`. It is the first-discovered amenable group that is not subexponentially amenable, see [BV05] and [GŻ02].

```
———————————————————— Example ————————————————————
gap> IsBoundedFRSemigroup(BasilicaGroup);
true
gap> pi := EpimorphismFromFreeGroup(BasilicaGroup); F := Source(pi);;
[ x1, x2 ] -> [ a, b ]
gap> sigma := GroupHomomorphismByImages(F,F,[F.1,F.2],[F.2,F.1^2]);
[ x1, x2 ] -> [ x2, x1^2 ]
gap> ForAll([0..10],i->IsOne(Comm(F.2,F.2^F.1)^(sigma^i*pi)));
true
```

9.1.4 FornaessSibonyGroup

▷ FornaessSibonyGroup (global variable)

The *Fornaess-Sibony group*. This group was studied by Nekrashevych in [Nek08b]. It is the iterated monodromy group of the endomorphism of $\mathbb{C}P^2$ defined by $(z,p) \mapsto ((1-2z/p)^2, (1-2/p)^2)$.

```
———————————————————— Example ————————————————————
gap> Size(NucleusOfFRSemigroup(FornaessSibonyGroup));
288
gap> List(AdjacencyBasesWithOne(FornaessSibonyGroup),Length);
[ 128, 128, 36, 36, 16, 16, 8 ]
gap> p := AdjacencyPoset(FornaessSibonyGroup);
```

```
    <general mapping: <object> -> <object> >
gap> Draw(HasseDiagramBinaryRelation(p));
```
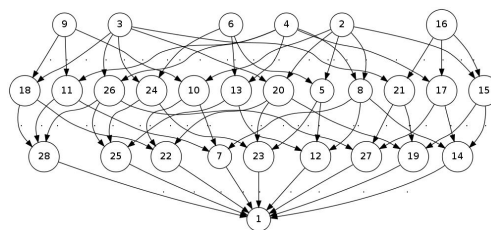

This produces (in a new window) the following picture:

9.1.5 AddingGroup

▷ AddingGroup(*n*)   (function)
▷ AddingMachine(*n*)   (function)
▷ AddingElement(*n*)   (function)

The second function constructs the adding machine on the alphabet `[1..n]`. This machine has a trivial state 1, and a non-trivial state 2. It implements the operation "add 1 with carry" on sequences.

The third function constructs the Mealy element on the adding machine, with initial state 2.

The first function constructs the state-closed group generated by the adding machine on `[1..n]`. This group is isomorphic to the `Integers`.

───── Example ─────
```
gap> Display(AddingElement(3));
   |  1     2     3
---+-----+-----+-----+
 a | a,1   a,2   a,3
 b | a,2   a,3   b,1
---+-----+-----+-----+
Initial state: b
gap> ActivityPerm(FRElement(AddingMachine(3),2),2);
(1,4,7,2,5,8,3,6,9)
gap> G := AddingGroup(3);
<self-similar group over [ 1 .. 3 ] with 1 generator>
gap> Size(G);
infinity
gap> IsRecurrentFRSemigroup(G);
true
gap> IsLevelTransitiveFRGroup(G);
true
```

9.1.6 BinaryAddingGroup

▷ BinaryAddingGroup   (global variable)
▷ BinaryAddingMachine   (global variable)
▷ BinaryAddingElement   (global variable)

These are respectively the same as `AddingGroup(2)`, `AddingMachine(2)` and `AddingElement(2)`.

9.1.7 MixerGroup

▷ MixerGroup(*A, B, f[, g]*) (function)
▷ MixerMachine(*A, B, f[, g]*) (function)

Returns: A Mealy "mixer" machine/group.

The second function constructs a Mealy "mixer" machine `m`. This is a machine determined by a permutation group `A`, a finitely generated group `B`, and a matrix of homomorphisms from `B` to `A`. If `A` acts on [1..d], then each row of `f` contains at most $d-1$ homomorphisms. The optional last argument is an endomorphism of `B`. If absent, it is treated as the identity map on `B`.

The states of the machine are 1, followed by some elements a of `A`, followed by as many copies of some elements b of `B` as there are rows in `f`. The elements in `B` that are taken is the smallest sublist of `B` containing its generators and closed under `g`. The elements in `A` that are taken are the generators of `A` and all images of all taken elements of `B` under maps in `f`.

The transitions from a are towards 1 and the outputs are the permutations in `A`. The transitions from b are periodically given by `f`, completed by trivial elements, and finally by b^g; the output of b is trivial.

This construction is described in more detail in [BŠ01] and [BGŠ03].

`Correspondence(m)` is a list, with in first position the subset of the states that correspond to `A`, in second position the states that correspond to the first copy of `B`, etc.

The first function constructs the group generated by the mixer machine. For examples see `GrigorchukGroups` (9.1.9), `NeumannGroup` (9.1.20), `GuptaSidkiGroups` (9.1.18), and `ZugadiSpinalGroup` (9.1.22).

```
──────────── Example ────────────
gap> grigorchukgroup := MixerGroup(Group((1,2)),Group((1,2)),
    [[IdentityMapping(Group((1,2)))],[IdentityMapping(Group((1,2)))],[]]));
<self-similar group over [ 1 .. 2 ] with 4 generators>
gap> IdGroup(Group(grigorchukgroup.1,grigorchukgroup.2));
[ 32, 18 ]
```

9.1.8 SunicGroup

▷ SunicGroup(*phi*) (function)
▷ SunicMachine(*phi*) (function)

Returns: The Sunic machine associated with the polynomial *phi*.

A "Sunic machine" is a special kind of `MixerMachine` (9.1.7), in which the group A is a finite field $GF(q)$, the group B is an extension $A[x]/(\phi)$, where ϕ is a monic polynomial; there is a map $f : B \to A$, given say by evaluation; and there is an endomorphism of $g : B \to B$ given by multiplication by ϕ.

These groups were described in [Šun07]. In particular, the case $q = 2$ and $\phi = x^2 + x + 1$ gives the original `GrigorchukGroup` (9.1.10).

```
──────────── Example ────────────
gap> x := Indeterminate(GF(2));;
gap> g := SunicGroup(x^2+x+1);
SunicGroup(x^2+x+Z(2)^0)
gap> g = GrigorchukGroup;
true
```

9.1.9 GrigorchukMachines

▷ GrigorchukMachines(*omega*)   (function)
▷ GrigorchukGroups(*omega*)   (function)

    Returns: One of the Grigorchuk groups.

    This function constructs the Grigorchuk machine or group associated with the infinite sequence *omega*, which is assumed (pre)periodic; *omega* is either a periodic list (see PeriodicList (11.2.2)) or a plain list describing the period. Entries in the list are integers in [1..3].

    These groups are MixerGroup (9.1.7)s. The most famous example is GrigorchukGroups([1,2,3]), which is also called GrigorchukGroup (9.1.10).

    These groups are all 4-generated and infinite. They are described in [Gri84]. GrigorchukGroups([1]) is infinite dihedral. If *omega* contains at least 2 different digits, GrigorchukGroups([1]) has intermediate word growth. If *omega* contains all 3 digits, GrigorchukGroups([1]) is torsion.

    The growth of GrigorchukGroups([1,2]) has been studied in [Ers04].

```
——————————————— Example ———————————————
gap> G := GrigorchukGroups([1]);
GrigorchukGroups([ 1 ])
gap> Index(G,DerivedSubgroup(G)); IsAbelian(DerivedSubgroup(G));
4
true
gap> H := GrigorchukGroups([1,2]);
GrigorchukGroups([ 1, 2 ])
gap> Order(H.1*H.2);
8
gap> Order(H.1*H.4);
infinity
gap> IsSubgroup(H,G);
true
```

9.1.10 GrigorchukMachine

▷ GrigorchukMachine   (global variable)
▷ GrigorchukGroup   (global variable)

    This is Grigorchuk's first group, introduced in [Gri80]. It is a 4-generated infinite torsion group, and has intermediate word growth. It could have been defined as FRGroup("a=(1,2)","b=<a,c>","c=<a,d>","d=<,b>"), but is rather defined using Mealy elements.

    The command EpimorphismFromFpGroup(GrigorchukGroup,n) constructs an approximating presentation for the Grigorchuk group, as proven in [Lys85]. Adding the relations Image(sigma^(n-2),(a*d)^2), Image(sigma^(n-1),(a*b)^2) and Image(sigma^(n-2),(a*c)^4) yields the quotient acting on 2^n points, as a finitely presented group.

9.1.11 GrigorchukOverGroup

▷ GrigorchukOverGroup   (global variable)

This is a group strictly containing the Grigorchuk group (see `GrigorchukGroup` (9.1.10)). It also has intermediate growth (see [BG02], but it contains elements of infinite order. It could have been defined as `FRGroup("a=(1,2)","b=<a,c>","c=<,d>","d=<,b>")`, but is rather defined using Mealy elements.

```
─────────── Example ───────────
gap> IsSubgroup(GrigorchukOverGroup,GrigorchukGroup);
true
gap> Order(Product(GeneratorsOfGroup(GrigorchukOverGroup)));
infinity
gap> GrigorchukGroup.2=GrigorchukSuperGroup.2*GrigorchukSuperGroup.3;
true
```

The command `EpimorphismFromFpGroup(GrigorchukOverGroup,n)` will will construct an approximating presentation for the Grigorchuk overgroup, as proven in [Bar03a].

9.1.12 GrigorchukTwistedTwin

▷ GrigorchukTwistedTwin   (global variable)

This is a group with same closure as the Grigorchuk group (see `GrigorchukGroup` (9.1.10)), but not isomorphic to it. It could have been defined as `FRGroup("a=(1,2)","x=<y,a>","y=<a,z>","z=<,x>")`, but is rather defined using Mealy elements.

```
─────────── Example ───────────
gap> AbelianInvariants(GrigorchukTwistedTwin);
[ 2, 2, 2, 2 ]
gap> AbelianInvariants(GrigorchukGroup);
[ 2, 2, 2 ]
gap> PermGroup(GrigorchukGroup,8)=PermGroup(GrigorchukTwistedTwin,8);
true
```

9.1.13 BrunnerSidkiVieiraGroup

▷ BrunnerSidkiVieiraGroup   (global variable)
▷ BrunnerSidkiVieiraMachine   (global variable)

This machine is the sum of two adding machines, one in standard form and one conjugated by the element d described below. The group that it generates is described in [BSV99]. It could have been defined as `FRGroup("tau=<,tau>(1,2)","mu=<,mu^-1>(1,2)")`, but is rather defined using Mealy elements.

```
─────────── Example ───────────
gap> V := FRGroup("d=<e,e>(1,2)","e=<d,d>");
<self-similar group over [ 1 .. 2 ] with 2 generators>
gap> Elements(V);
[ <2|identity ...>, <2|e>, <2|d>, <2|e*d> ]
gap> AssignGeneratorVariables(BrunnerSidkiVieiraGroup);
#I  Assigned the global variables [ "tau", "mu", "" ]
gap> List(V,x->tau^x)=[tau,mu,mu^-1,tau^-1];
true
```

9.1.14 AleshinGroups

▷ AleshinGroups(*l*)   (function)

▷ AleshinMachines(*l*)   (function)

    Returns: The Aleshin machine with `Length(l)` states.

    This function constructs the bireversible machines introduced by Aleshin in [Ale83]. The argument *l* is a list of permutations, either `()` or `(1,2)`. The groups that they generate are contructed as `AleshinGroups`.

    If `l=[(1,2),(1,2),()]`, this is `AleshinGroup` (9.1.15). More generally, if `l=[(1,2,(1,2),(),...,()]` has odd length, this is a free group of rank `Length(l)`, see [VV07] and [VV06].

    If `l=[(1,2),(1,2),...]` has even length and contains an even number of `()`'s, then this is also a free group of rank `Length(l)`, see [SVV06].

    If `l=[(),(),(1,2)]`, this is `BabyAleshinGroup` (9.1.16). more generally, if `l=[(),(),...]` has even length and contains an even number of `(1,2)`'s, then this is the free product of `Length(l)` copies of the order-2 group.

9.1.15 AleshinGroup

▷ AleshinGroup   (global variable)

▷ AleshinMachine   (global variable)

    This is the first example of non-abelian free group. It is the group generated by `AleshinMachine([(1,2),(1,2),()])`. It could have been defined as `FRGroup("a=<b,c>(1,2)","b=<c,b>(1,2)","c=<a,a>")`, but is rather defined using Mealy elements.

9.1.16 BabyAleshinGroup

▷ BabyAleshinGroup   (global variable)

▷ BabyAleshinMachine   (global variable)

    There are only two connected, transitive bireversible machines on a 2-letter alphabet, with 3 generators: `AleshinMachine(3)` and the baby Aleshin machine.

    The group generated by this machine is abstractly the free product of three C_2's, see [Nek05, 1.10.3]. It could have been defined as `FRGroup("a=<b,c>","b=<c,b>","c=<a,a>(1,2)")`, but is rather defined here using Mealy elements.

```
———————————————— Example ————————————————
gap> K := Kernel(GroupHomomorphismByImagesNC(BabyAleshinGroup,Group((1,2)),
               GeneratorsOfGroup(BabyAleshinGroup),[(1,2),(1,2),(1,2)]));
<self-similar group over [ 1 .. 2 ] with 4 generators>
gap> Index(BabyAleshinGroup,K);
2
gap> IsSubgroup(AleshinGroup,K);
true
```

9.1.17   SidkiFreeGroup

▷ SidkiFreeGroup   (global variable)

This is a group suggested by Sidki as an example of a non-abelian state-closed free group. Nothing is known about that group: whether it is free as conjectured; whether generator a is state-closed, etc. It is defined as FRGroup("a=<a^2,a^t>","t=<,t>(1,2)")]]>.

9.1.18   GuptaSidkiGroups

▷ GuptaSidkiGroups(*n*)   (function)
▷ GeneralizedGuptaSidkiGroups(*n*)   (function)
▷ GuptaSidkiMachines(*n*)   (function)
   Returns: The Gupta-Sidki group/machine on an *n*-letter alphabet.
   This function constructs the machines introduced by Gupta and Sidki in [GS83]. They generate finitely generated infinite torsion groups: the exponent of every element divides some power of *n*. The special case *n* = 3 is defined as GuptaSidkiGroup (9.1.19) and GuptaSidkiMachine (9.1.19).
   For *n* > 3, there are (at least) two natural ways to generalize the Gupta-Sidki construction: the original one involves the recursion "t=<a,a^-1,1,...,1,t>", while the second (called here 'generalized') involves the recursion "t=<a,a^2,...,a^-1,t>". A finite L-presentation for the latter is implemented. It is not as computationally efficient as the L-presentation of the normal closure of t (a subgroup of index *p*), which is an ascending L-presented group. The inclusion of that subgroup may be recoverd via EmbeddingOfAscendingSubgroup(GuptaSidkiGroup).

9.1.19   GuptaSidkiGroup

▷ GuptaSidkiGroup   (global variable)
▷ GuptaSidkiMachine   (global variable)

This is an infinite, 2-generated, torsion 3-group. It could have been defined as FRGroup("a=(1,2,3)","t=<a,a^-1,t>"), but is rather defined using Mealy elements.

9.1.20   NeumannGroup

▷ NeumannGroup(*P*)   (function)
▷ NeumannMachine(*P*)   (function)
   Returns: The Neumann group/machine on the permutation group *P*.
   The first function constructs the Neumann group associated with the permutation group *P*. These groups were first considered in [Neu86]. In particular, NeumannGroup(PSL(3,2)) is a group of non-uniform exponential growth (see [Bar03b]), and NeumannGroup(Group((1,2,3))) is FabrykowskiGuptaGroup (9.1.21).
   The second function constructs the Neumann machine associated to the permutation group *P*. It is a shortcut for MixerMachine(P,P,[[IdentityMapping(P)]]).
   The attribute Correspondence(G) is set to a list of homomorphisms from *P* to the A and B copies of P.

9.1.21   FabrykowskiGuptaGroup

▷ FabrykowskiGuptaGroup   (global variable)
▷ FabrykowskiGuptaGroups(*p*)   (function)

   This is an infinite, 2-generated group of intermediate word growth. It was studied in [FG85] and [FG91]. It could have been defined as `FRGroup("a=(1,2,3)","t=<a,t>")`, but is rather defined using Mealy elements. It has a torsion-free subgroup of index 3 and is branched.

   The natural generalization, replacing 3 by *p*, is constructed by the second form. It is a specific case of Neumann group (see `NeumannGroup` (9.1.20)), for which an ascending L-presentation is known.

9.1.22   ZugadiSpinalGroup

▷ ZugadiSpinalGroup   (global variable)

   This is an infinite, 2-generated group, which was studied in [BG02]. It has a torsion-free subgroup of index 3, and is virtually branched but not branched. It could have been defined as `FRGroup("a=(1,2,3)","t=<a,a,t>")`, but is rather defined using Mealy elements.

   Amaia Zugadi computed its Hausdorff dimension to be $1/2$.

9.1.23   GammaPQMachine

▷ GammaPQMachine(*p, q*)   (function)
▷ GammaPQGroup(*p, q*)   (function)

   Returns: The quaternion-based machine / SC group.

   The first function constructs a bireversible (see `IsBireversible` (5.2.7)) Mealy machine based on quaternions, see [BM00a] and [BM00b]. This machine has $p+1$ states indexed by integer quaternions of norm *p*, and an alphabet of size $q+1$ indexed by quaternions of norm *q*. These quaternions are congruent to 1   (mod 2) or *i*   (mod 2) depending on whether the odd prime *p* or *q* is 1 or 3 (mod 4).

   The structure of the machine is such that there is a transition from (q,a) to (q',a') precisely when $qa' = \pm q'a$. In particular, the relations of the `StructuralGroup` (3.5.1) hold up to a sign, when the alphabet/state letters are substituted for the appropriate quaternions.

   The quaternions themselves can be recovered through `Correspondence` (3.5.12), which is a list with in first position the quaternions of norm *p* and in second those of norm *q*.

   The second function constructs the quaternion lattice that is the `StructuralGroup` (3.5.1) of that machine. It has actions on two trees, given by `VerticalAction` (9.5.2) and `HorizontalAction` (9.5.2); the ranges of these actions are groups generated by automata, which are infinitely-transitive (see `IsInfinitelyTransitive` (7.2.13)) and free by [GM05, Proposition 3.3]; see also [Ale83].

9.1.24   RattaggiGroup

▷ RattaggiGroup   (global variable)

   This record contains interesting examples of VH groups, that were studied by Rattaggi in his PhD thesis [Rat04]. His Example 2.x appears as `RattaggiGroup.2_x`.

   The following summary of the examples' properties are copied from Rattaggi's thesis. RF means "residually finite"; JI means "just infinite"; VS means "virtually simple".

Example	P_h	P_v	Irred?	Linear?	RF?	JI?	VS?
2.2	2tr	2tr	Y	N	N?	Y	Y?
2.15							
2.18							
2.21							
2.26	2tr	q-prim	Y	N	N	N	N
2.30	2tr	2tr	Y	N	N	Y	Y?
2.36							
2.39							
2.43	2tr	2tr	Y	N	N	Y	Y
2.46							
2.48							
2.50							
2.52	tr	2tr	Y	N	N	N	N
2.56							
2.58	2tr	prim	Y	N	N?	N	N
2.70							
3.26	2tr	2tr	Y	Y	Y	Y	N
3.28							
3.31							
3.33							
3.36							
3.38							
3.40							
3.44							
3.46							
3.72							

9.1.25 HanoiGroup

▷ `HanoiGroup(n)` (function)

Returns: The Hanoi group on an `n` pegs.

This function constructs the Hanoi group on `n` pegs. Generators of the group correspond to moving a peg, and tree vertices correspond to peg configurations. This group is studied in [GŠ06].

9.1.26 DahmaniGroup

▷ `DahmaniGroup` (global variable)

This is an example of a non-contracting weakly branched group. It was first studied in [Dah05]. It could have been defined as `FRGroup("a=<c,a>(1,2)","b=<b,a>(1,2)","c=<b,c>")`, but is rather defined using Mealy elements.

It has relators abc, $[a^2c, [a,c]]$, $[cab, a^{-1}c^{-1}ab]$ and $[ac^2, c^{-1}b^{-1}c^2]$ among others.

It admits an endomorphism on its derived subgroup. Indeed `FRElement(1,Comm(a,b))=Comm(c^-1,b/a)`, `FRElement(1,Comm(a,c))=Comm(a/b,c)`, `FRElement(1,Comm(b,c))=Comm(c,(a/b)^c)`.

9.1.27 MamaghaniGroup

▷ MamaghaniGroup (global variable)

This group was studied in [Mam03]. It is fractal, but not contracting. It could have been defined as `FRGroup("a=<,b>(1,2)","b=<a,c>","c=<a,a^-1>(1,2)")]]>`, but is rather defined using Mealy elements. It partially admits branching on its subgroup `Subgroup(G,[a^2,(a^2)^b,(a^2)^c])`, and, setting `x=Comm(a^2,b)`, on `Subgroup(G,[x,x^a,x^b,x^(b*a),x^(b/a)])`. One has `FRElement(1,x)=(x^-1)^b/x`.

9.1.28 WeierstrassGroup

▷ WeierstrassGroup (global variable)

This is the iterated monodromy group associated with the Weierstrass $\wp$-function.

Some relators in the group: $(atbt)^4$, $((atbt)(bt)^4n)^4$, $((atbt)^2(bt)^4n)^2$.

Set $x = [a,t]$, $y = [b,t]$, $z = [c,t]$, and $w = [x,y]$. Then $G' = \langle x,y,z \rangle$ of index 8, and $\gamma_3 = \langle [\{x,y,z\},\{a,b,c\}] \rangle$ of index 32, and $\gamma_4 = G'' = \langle w \rangle^G$, of index 256, and $G'' > (G'')^4$ since $[[t^a,t],[t^b,t]] = (w,1,1,1)$.

The Schreier graph is obtained in the complex plane as the image of a $2^n \times 2^n$ lattice in the torus, via Weierstrass's $\wp$-function.

The element *at* has infinite order.

$[c,t,b][b,t,c][a,t,c][c,t,a]$ has order 2, and belongs to G''; so there exist elements of arbitrary large finite order in the group.

9.1.29 StrichartzGroup

▷ StrichartzGroup (global variable)

This group generates the graph of the Strichartz hexacarpet.

The Strichartz hexacarpet is the dual graph to the infinitely iterated barycentric subdivision of the triangle. The Strichartz group acts on $\{1,\ldots,6\}^n$ for all n, and the Schreier graph with 6^n vertices is the nth Strichartz graph.

Conjecturally, that graph's radius is $1/18(2^(n+1)(13+3n)+(-1)^n-9)$ and its diameter is $1/9(2^(n-1)(31+12n)+2(-1)^(n-1)-18)$.

See [BKN$^+$12] for details.

9.1.30 FRAffineGroup

▷ FRAffineGroup(d, R, u[, transversal]) (operation)

Returns: The `d`-dimensional affine group over `R`.

This function constructs a new FR group G, which is finite-index subgroup of the `d`-dimensional affine group over R_u, the local ring over `R` in which all non-multiples of `u` are invertible. Since no generators of G are known, G is in fact returned as a full SC group; only its attribute `Correspondence(G)`, which is a homomorphism from $GL_{d+1}(R_u)$ to G, is relevant.

The affine group can also be described as a subgroup of $GL_{d+1}(R_u)$, consisting of those matrices M with $M_{i,d+1} = 0$ and $M_{d+1,d+1} = 1$. The finite-index subgroup is defined by the conditions $u|M_{i,j}$ for all $j < i$.

The only valid arguments are R=Integers and R=PolynomialRing(S) for a finite ring S. The alphabet of the affine group is R/RuR; an explicit transversal of RuR be specified as last argument.

The following examples construct the "Baumslag-Solitar group" $\mathbb{Z}[\frac{1}{2}] \rtimes_2 \mathbb{Z}$ first introduced in [BS62], the "lamplighter group" $(\mathbb{Z}/2) \wr \mathbb{Z}$, and a 2-dimensional affine group. Note that the lamplighter group may also be defined via CayleyGroup (9.1.31).

```
 ─────────────── Example ───────────────
 gap> A := FRAffineGroup(1,Integers,3);
 <self-similar group over [ 1 .. 3 ]>
 gap> f := Correspondence(A);
 MappingByFunction( ( Integers^
 [ 2, 2 ] ), <self-similar group over [ 1 .. 3 ]>, function( mat ) ... end )
 gap> BaumslagSolitar := Group([[2,0],[0,1]]^f,[[1,0],[1,1]]^f);
 <self-similar group over [ 1 .. 3 ] with 2 generators>
 gap> BaumslagSolitar.2^BaumslagSolitar.1=BaumslagSolitar.2^2;
 true
 gap> R := PolynomialRing(GF(2));;
 gap> A := FRAffineGroup(1,R,R.1);;
 gap> f := Correspondence(A);;
 gap> Lamplighter := Group(([[1+R.1,0],[0,1]]*One(R))^f,([[1,0],[1,1]]*One(R))^f);
 <self-similar group over [ 1 .. 2 ] with 2 generators>
 gap> Lamplighter = CayleyGroup(Group((1,2)));
 true
 gap> StructureDescription(Group(Lamplighter.2,Lamplighter.2^Lamplighter.1));
 "C2 x C2"
 gap> ForAll([1..10],i->IsOne(Comm(Lamplighter.2,Lamplighter.2^(Lamplighter.1^i))));
 true
 gap> A := FRAffineGroup(2,Integers,2);;
 gap> f := Correspondence(A);;
 gap> a := [[1,4,0],[2,3,0],[1,0,1]];
 [ [ 1, 4, 0 ], [ 2, 3, 0 ], [ 1, 0, 1 ] ]
 gap> b := [[1,2,0],[4,3,0],[0,1,1]];
 [ [ 1, 2, 0 ], [ 4, 3, 0 ], [ 0, 1, 1 ] ]
 gap> Display(b^f);
     |   1       2
 ----+------+------+
   a |  b,1     c,2
   b |  d,2     e,1
   c |  a,2     f,1
 ...
  bh |  cb,1    be,2
  ca |  bd,1    bf,2
  cb |  ae,2    bh,1
 ----+------+------+
 Initial state:  a
 gap> a^f*b^f=(a*b)^f;
 true
```

9.1.31 CayleyGroup

▷ CayleyGroup(*G*)   (function)
▷ CayleyMachine(*G*)   (function)

▷ LamplighterGroup(*IsFRGroup, G*)   (method)

Returns: The Cayley machine/group of the group G.

The *Cayley machine* of a group G is a machine with alphabet and stateset equal to G, and with output and transition functions given by multiplication in the group, in the order state*letter.

The second function constructs a new FR group CG, which acts on [1..Size(G)]. Its generators are in bijection with the elements of G, and have as output the corresponding permutation of G induced by right multiplication, and as transitions all elements of G; see CayleyMachine. This construction was introduced in [SS05].

If G is an abelian group, then CG is the wreath product $G \wr \mathbb{Z}$; it is created by the constructor LamplighterGroup(IsFRGroup,G).

The attribute Correspondence(CG) is a list. Its first entry is a homomorphism from G into CG. Its second entry is the generator of CG that has trivial output. CG is generated Correspondence(CG)[2] and the image of Correspondence(CG)[1].

In the example below, recall the definition of Lamplighter in the example of FRAffineGroup (9.1.30).

```
——————————————————————————— Example ———————————————————————————
gap> L := CayleyGroup(Group((1,2)));
CayleyGroup(Group( [ (1,2) ] ))
gap> L=LamplighterGroup(IsFRGroup,CyclicGroup(2));
true
gap> (1,2)^Correspondence(L)[1];
<Mealy element on alphabet [ 1, 2 ] with 2 states, initial state 1>
gap> IsFinitaryFRElement(last); Display(last2);
true
    |  1     2
 ---+-----+-----+
  a |  b,2   b,1
  b |  b,1   b,2
 ---+-----+-----+
Initial state: a
```

9.2   Examples of semigroups

9.2.1   I2Machine

▷ I2Machine   (global variable)

▷ I2Monoid   (global variable)

The Mealy machine I_2, and the monoid that it generates. This is the smallest Mealy machine generating a monoid of intermediate word growth; see [BRS06].

For sample calculations in this monoid see SCSemigroup (7.1.3).

9.2.2   I4Machine

▷ I4Machine   (global variable)

▷ I4Monoid   (global variable)

The Mealy machine generating I_4, and the monoid that it generates. This is a very small Mealy machine generating a monoid of intermediate word growth; see [BR08].

For sample calculations in this monoid see `SCMonoid` (7.1.3).

9.3 Examples of algebras

9.3.1 PSZAlgebra

▷ PSZAlgebra(*k[, m]*)   (function)

This function creates an associative algebra `A`, over the field *k* of positive characteristic, generated by *m* derivations `d0,...,d(m-1),v`. If the argument *m* is absent, it is taken to be 2.

This algebra has polynomial growth, and is not nilpotent. Petrogradsky showed in [Pet06] that the Lie subalgebra of `PSZAlgebra(GF(2))` generated by $v, [u, v]$ is nil; this result was generalized by Shestakov and Zelmanov in [SZ08] to arbitrary *k* and *m* = 2.

This ring is $\mathbb{Z}^m$-graded; the attribute `Grading` is set. It is graded nil [PSZ].

```
———————————— Example ————————————
gap> a := PSZAlgebra(2);
PSZAlgebra(GF(2))
gap> Nillity(a.1); Nillity(a.2);
2
4
gap> IsNilElement(LieBracket(a.1,a.2));
true
gap> DecompositionOfFRElement(LieBracket(a.1,a.2))=DiagonalMat([a.2,a.2]);
true
```

9.3.2 GrigorchukThinnedAlgebra

▷ GrigorchukThinnedAlgebra(*k*)   (function)

This function creates the associative envelope `A`, over the field *k*, of Grigorchuk's group `GrigorchukGroup` (9.1.10).

k may be a field or an prime representing the prime field `GF(k)`. In characteristic 2, this ring is graded, and the attribute `Grading` is set.

For more information on the structure of this thinned algebra, see [Bar06].

```
———————————— Example ————————————
gap> R := GrigorchukThinnedAlgebra(2);
<self-similar algebra-with-one on alphabet GF(2)^2 with 4 generators, of dimension infinity>
gap> GrigorchukGroup.1^Embedding(GrigorchukGroup,R)=R.1;
true
gap> Nillity(R.2+R.1);
16
gap> x := 1+R.1+R.2+(R.1-1)*(R.4-1); # x has infinite order
<Linear element on alphabet GF(2)^2 with 5-dimensional stateset>
gap> Inverse(x);
<Linear element on alphabet GF(2)^2 with 9-dimensional stateset>
gap> Grading(R).hom_components(4);
<vector space of dimension 6 over GF(2)>
```

```
gap> Random(last);
<Linear element on alphabet GF(2)^2 with 6-dimensional stateset>
gap> Nillity(last);
4
```

9.3.3 GuptaSidkiThinnedAlgebra

▷ GuptaSidkiThinnedAlgebra(*k*)   (function)

This function creates the associative envelope A, over the field *k*, of Gupta-Sidki's group GuptaSidkiGroup (9.1.19).

k may be a field or an prime representing the prime field GF(k).

For more information on the structure of this thinned algebra, see [Sid97].

```
—————————— Example ——————————
gap> R := GuptaSidkiThinnedAlgebra(3);
<self-similar algebra-with-one on alphabet GF(3)^3 with 4 generators>
gap> Order(R.1);
3
gap> R.1*R.3;
<Identity linear element on alphabet GF(3)^3>
gap> IsOne(R.2*R.4);
true
gap> x := 1+R.2*(1+R.1+R.3); # a non-invertible element
<Linear element on alphabet GF(3)^3 with 5-dimensional stateset>
gap> Inverse(x);
#I  InverseOp: extending to depth 3
#I  InverseOp: extending to depth 4
#I  InverseOp: extending to depth 5
#I  InverseOp: extending to depth 6
Error, user interrupt in
```

9.3.4 GrigorchukLieAlgebra

▷ GrigorchukLieAlgebra(*k*)   (function)
▷ GuptaSidkiLieAlgebra(*k*)   (function)

Two more examples of self-similar Lie algebras; see [Bar10].

9.3.5 SidkiFreeAlgebra

▷ SidkiFreeAlgebra(*k*)   (function)

This is an example of a free 2-generated associative ring over the $\mathbb{Z}$, defined by self-similar matrices. It is due to S. Sidki. The argument is a field on which to construct the algebra. The recursion is s=[[1,0],[0,2*s]] and t=[[0,2*s],[0,2*t]].

```
—————————— Example ——————————
gap> R := SidkiFreeAlgebra(Rationals);
<self-similar algebra-with-one on alphabet Rationals^2 with 2 generators>
gap> V := VectorSpace(Rationals,[R.1,R.2]);
```

```
<vector space over Rationals, with 2 generators>
gap> P := [V];; for i in [1..3] do Add(P,ProductSpace(P[i],V)); od;
gap> List(P,Dimension);
[ 2, 4, 8, 16 ]
gap> R := SidkiFreeAlgebra(GF(3));
<self-similar algebra-with-one on alphabet GF(3)^2 with 2 generators>
gap> V := VectorSpace(GF(3),[R.1,R.2]);;
gap> P := [V];; for i in [1..3] do Add(P,ProductSpace(P[i],V)); od;
gap> List(P,Dimension);
[ 2, 4, 7, 12 ]
```

9.3.6 SidkiMonomialAlgebra

▷ SidkiMonomialAlgebra(*k*)   (function)

This is an example of a self-similar algebra that does not come from a semigroup; it is due to S. Sidki. The argument is a field on which to construct the algebra. The recursion is `s=[[0,0],[1,0]]` and `t=[[0,t],[0,s]]`. Sidki shows that this algebra, like the Grigorchuk thinned algebra in characteristic 2 (see `GrigorchukThinnedAlgebra` (9.3.2)), admits a monomial presentation, and in particular is a graded ring.

```
——————————————————————————— Example ———————————————————————————
gap> R := SidkiMonomialAlgebra(Rationals);
<self-similar algebra-with-one on alphabet Rationals^2 with 2 generators>
gap> m := FreeSemigroup("s","t");;
gap> lambda := MagmaEndomorphismByImagesNC(m,[m.2,m.1*m.2]);;
gap> u := [m.1^2];; for i in [1..3] do u[2*i] := m.2*u[2*i-1]^lambda; u[2*i+1] := u[2*i]^lambda;
gap> u; # first relations
[ s^2, t^3, s*t*s*t*s*t, t^2*s*t^2*s*t^2*s*t,
  s*t*s*t^2*s*t*s*t^2*s*t*s*t^2*s*t,
  t^2*s*t^2*s*t*s*t^2*s*t^2*s*t*s*t^2*s*t^2*s*t*s*t^2*s*t,
  s*t*s*t^2*s*t*s*t^2*s*t^2*s*t*s*t*s*t^2*s*t^2*s*t*s*t^2*s*t^2*s*t*s*t^2*s*t ]
gap> pi := MagmaHomomorphismByImagesNC(m,R,[R.1,R.2]);;
gap> Image(pi,u);
[ <Zero linear element on alphabet Rationals^2> ]
gap> # growth given by fibonacci numbers
gap> List([0..6],Grading(R).hom_components);
[ <vector space over Rationals, with 1 generators>, <vector space over Rationals, with 2 generato
  <vector space of dimension 3 over Rationals>, <vector space of dimension 4 over Rationals>,
  <vector space of dimension 5 over Rationals>, <vector space of dimension 7 over Rationals>,
  <vector space of dimension 8 over Rationals> ]
```

9.4   Bacher's determinant identities

In his paper [Bac08], Roland Bacher exhibits striking formulas for determinants of matrices obtained from binomial coefficients.

The general construction is as follows: let P be an infinite matrix, and let $P(n)$ be its upper $n \times n$ corner. To evaluate $\det P(n)$, decompose $P = LDR$ where L, D, R are respectively lower triangular, diagonal, and upper triangular, with 1's on the diagonals of L and R. Then that determinant is the product of the first n entries of D.

Bacher considers some natural examples of matrices arising from binomial coefficients, and notes that the matrix P is the limit of a convergent vector element (see `IsConvergent` (6.1.9)). He also notes that the decomposition $P = LDR$ may be achieved within vector elements.

As a first example, consider the $n \times n$ matrix $P(n)$ with coefficients $P_{s,t} = \binom{s+t}{s}$ (mod 2). Here and below, indices start at 0. Let also $ds(n)$ denote the digit-sum of the integer n. Then

$$\det(P(n)) = \begin{cases} (-1)^{n/2} & \text{if } n \text{ is even,} \\ (-1)^{(n-1)/2 + ds((n-1)/2)} & \text{if } n \text{ is odd.} \end{cases}$$

For the proof, note that P is a convergent infinite matrix; it may be presented as a self-similar linear element by `FRAlgebra("P=[[P,P],[P,0]]")`. It then suffices to construct an LR decomposition of P within FR vector elements, following Bacher:

```
──────────────── Example ────────────────
gap> AssignGeneratorVariables(FRAlgebra(Rationals,
    "P=[[P,P],[P,0]]","L=[[L,0],[L,L]]","D=[[D,0],[0,-D]]"));
gap> L*D*TransposedFRElement(L)=P;
true
```

and to analyse the elements of the diagonal matrix D.

For a more complicated example, let v_2 denote 2-valuation of a rational, and construct the $n \times n$ matrix $V(n)$ with coefficients $V_{s,t} = i^{v_2(\binom{s+t}{s})}$. Then

$$\det(V(n)) = \det(P(n)) \prod_{k=1}^{n-1} (1 - f(k)i),$$

where $f(k)$ is the regular paper-folding sequence defined by $f(2^n) = 1$ and $f(2^n + a) = -f(2^n - a)$ for $1 \le a < 2^n$.

This is again proved by noticing that V is a corner in a self-similar element, namely

```
──────────────── Example ────────────────
gap> AssignGeneratorVariables(FRAlgebra(GaussianRationals,
    "V1=[[V1,V2],[V2,E(4)*V1]]",
    "V2=[[V1,-E(4)*V1+(1+E(4))*V2],[-E(4)*V1+(1+E(4))*V2,-V1]]"));
gap> Activity(V1,3)=
    List([0..7],s->List([0..7],t->E(4)^ValuationInt(Binomial(s+t,s),2)));
true
```

The LR decomposition of $V = V1$ can be checked as follows:

```
──────────────── Example ────────────────
gap> AssignGeneratorVariables(FRAlgebra(GaussianRationals,
    "L1=[[L1,0],[L3,L4]]",
    "L2=[[0,-E(4)*L2],[-L1+L3,-E(4)*L2-E(4)*L4]]:0",
    "L3=[[L1,L2],[-E(4)*L1+(1+E(4))*L3,L2+(1+E(4))*L4]]",
    "L4=[[L1,0],[(1-E(4))*L1+E(4)*L3,L4]]",
    "D1=[[D1,0],[0,D2]]",
    "D2=[[D3,0],[0,2*D1-D2+2*D3]]:-1+E(4)",
    "D3=[[D3,0],[0,-D2]]:-1+E(4)"));
gap> L1*D1*TransposedFRElement(L1)=V1;
true
```

The LR decomposition can also, in favourable situations, be discovered by FR through the command `LDUDecompositionFRElement` (6.1.11). This approach will be followed below.

For the next example, consider "Beeblebrox reduction" $\beta(4k \pm 1) = \pm 1, \beta(2k) = 0$, and construct the $n \times n$ matrix $Z(n)$ (named after Zaphod Beeblebrox) with coefficients $Z_{s,t} = \beta\left(\binom{s+t}{s}\right)$. Then

$$\det(Z(n)) = \prod_{k=1}^{n-1} g(k),$$

where $g(\sum a_i 2^i) = (-1)^{a_0} 3^{\#\{i:a_i=a_{i+1}=1\} - \#\{i:a_i \neq a_{i+1}=1\}}$ with all $a_i \in \{0,1\}$.

This is again proved by noticing that Z is a corner in a self-similar element, namely

```
──────────────────── Example ────────────────────
gap> beta := n->Jacobi(-1,n)*(n mod 2);;
gap> Zaphod := GuessVectorElement(List([0..7],i->List([0..7],j->beta(Binomial(i+j,j)))));
<Linear element on alphabet Rationals^2 with 3-dimensional stateset>
gap> Display(Zaphod);
 Rationals |    1     |    2     |
-----------+----------+----------+
         1 |  1  0  0 |  0  1  0 |
           |  1  0  0 |  0  1  0 |
           |  1  0  0 |  0 -1  0 |
-----------+----------+----------+
         2 |  0  0  1 |  0  0  0 |
           |  0  0 -1 |  0  0  0 |
           |  0  0  1 |  0  0  0 |
-----------+----------+----------+
Output:  1  1  1
Initial state:  1  0  0
gap> LDUDecompositionFRElement(guessZ);
[ <Linear element on alphabet Rationals^2 with 4-dimensional stateset>,
  <Linear element on alphabet Rationals^2 with 2-dimensional stateset>,
  <Linear element on alphabet Rationals^2 with 4-dimensional stateset> ]
gap> Display(last[2]);
 Rationals |    1    |    2    |
-----------+---------+---------+
         1 |  1   0  |  0   0  |
           |  3   0  |  0   0  |
-----------+---------+---------+
         2 |  0   0  |  0   1  |
           |  0   0  |  0 1/3  |
-----------+---------+---------+
Output:  1  -1
Initial state:   1   0
```

and now the recursion read on this diagonal self-similar matrix gives immediately Bacher's recursion for $\det(Z(n))$.

Bacher notes that the group generated by $a = L_1, b = L_2/2, c = L_3, d = L_4$ in the last example may be of interest. A quick check produces the following relations (slightly rewritten):

```
──────────────────── Example ────────────────────
gap> AssignGeneratorVariables(FRAlgebra(Rationals,
      "a=[[a,0],[c,d]]","b=[[-1/3*a,2*b],[1/3*c,d]]",
      "c=[[a,2*b],[c,d]]","d=[[a,0],[1/3*c,d]]"));
```

```
gap> g := Group(List([a,b,c,d], x->Activity(x,3)));
<matrix group with 4 generators>
gap> FindShortGroupRelations(g,10);
[ b*d^-1*c*a^-1,
  c*a^-1*c*a^-1,
  c*a*d^-1*a^-1*d^2*a^-1*b^-1,
  c*a*d^-1*c^-1*b*d*a^-1*b^-1,
  c*d*a^-2*d*a*d^-1*b^-1,
  c*a^2*d^-1*a^-2*d*a*d*a^-2*b^-1,
  d^2*a*d^-2*b^-1*c*a*d*a^-3,
  c*d*a*d^-2*a^-1*d*a*d*a^-2*b^-1 ]
```

Consider next the "triangular Beeblebrox matrix" with entries $L_{s,t} = \beta(\binom{s}{t})$. The recurrence is now given by

```
────────────────────────── Example ──────────────────────────
gap> A := FRAlgebra(Rationals,
      "L1=[[L1,0],[L2,L3]]",
      "L2=[[L1,0],[L2,-L3]]",
      "L3=[[L1,0],[-L2,L3]]");
<self-similar algebra on alphabet Rationals^2 with 3 generators>
```

and it is striking that A is a graded algebra, with L_1, L_2, L_3 homogeneous of degree 1, and each homogeneous component is 3-dimensional; all of L_1, L_2, L_3 are invertible (with inverses have degree -1), and generate a group that admits a faithful 3×3 linear representation. As a final example, Bacher considers the "Jacobi character" $\chi(8\mathbb{Z} \pm 1) = 1, \chi(8\mathbb{Z} \pm 3) = -1, \chi(2\mathbb{Z}) = 0$, and the associated matrix $J_{s,t} = \chi(\binom{s+t}{s})$. He gives an easily-computed, but complicated formula for $\det(J(n))$. We can recover this formula, as before, by "guessing" an LR decomposition for J, which is self-similar and convergent:

```
────────────────────────── Example ──────────────────────────
gap> chi := function(x)
        if x mod 8 in [1,7] then return 1;
        elif x mod 8 in [3,5] then return -1;
        else return 0; fi;
      end;;
gap> m := List([0..63],i->List([0..63],j->chi(Binomial(i+j,j))));;
gap> J := GuessVectorElement(m,2);
<Linear element on alphabet Rationals^2 with 9-dimensional stateset>
gap> LDUDecompositionFRElement(J);
[ <Linear element on alphabet Rationals^2 with 20-dimensional stateset>,
  <Linear element on alphabet Rationals^2 with 4-dimensional stateset>,
  <Linear element on alphabet Rationals^2 with 20-dimensional stateset> ]
gap> time;
26869
gap> Display(last2[2]);
 Rationals |       1        |       2        |
 ----------+----------------+----------------+
        1 |  1    0    0    0 |  0    0    0    0 |
          |  0    0    1    0 |  0    0    0    0 |
          |  3    0    0    0 |  0    0    0    0 |
          |  0    0    3    0 |  0    0    0    0 |
 ----------+----------------+----------------+
```

```
      2 |   0   0   0   0 |   0   1   0   0 |
        |   0   0   0   0 |   0   0   0   1 |
        |   0   0   0   0 |   0 1/3  0   0 |
        |   0   0   0   0 |   0   0   0 1/3 |
----------+----------------+----------------+
Output:   1  -1   3 -1/3
Initial state:   1   0   0   0
```

9.5 VH groups

FR understands a special kind of finitely presented groups, called *VH groups*. These are groups with two distinguished sets of generators, V and H, and such that for every choice of $v \in V, h \in H$ there are unique $v' \in V, h' \in H$ such that $vh = h'v'$ and conversely. In other words, these are finitely presented groups whose Cayley complex is a product of two trees.

These groups are of particular interest thanks to the work of Burger and Mozes, see [BM00a] and [BM00b], who constructed the first examples of finitely presented simple groups in this manner.

VH groups are connected to groups generated by automata as follows. Given a VH group, consider the automaton with stateset V, acting on alphabet H; its output and transition are determined by $\Phi(v, h) = (h', v')$ where v', h' are determined by the equation $vh = h'v'$.

Conversely, any bireversible automaton gives rise to a VH group by the inverse construction.

FR contains commands that automatize the verification that a VH group is non-residually finite, or virtually simple. Inspiration came from Diego Rattaggi's PhD thesis [Rat04].

9.5.1 VHStructure

▷ VHStructure(g) (operation)
▷ IsVHGroup(g) (filter)

Returns: A VH-structure for the group g.

A *VH-structure* on a group g is a partition of the generators in two sets V, H such that every relator of g is of the form $vhv'h'$, and such that for all $v \in V, h \in H$ there exist unique $v' \in V, h' \in H$ such that $vhv'h' = 1$.

The VH structure is stored as a record with fields v,h containing lists of generators, and integer matrices transitions,output such that transitions[v][h']=v' and output[v][h']=h.

The filter recognizes groups with a VH structure.

9.5.2 VerticalAction

▷ VerticalAction(g) (attribute)
▷ HorizontalAction(g) (attribute)

Returns: A homomorphism to an FR group.

A group with VH structure admits a *vertical action* of its subgroup $\langle V \rangle$; this is the group generated by the automaton MealyMachine(trans,out). The function returns the group homomorphism from the subgroup $\langle V \rangle$ to that FR group.

The horizontal action is that of the dual automaton (see DualMachine (5.2.3)).

```
—————————————— Example ——————————————
gap> v := VerticalAction(RattaggiGroup.2_21);
[ a1, a2, a3 ] -> [ <Mealy element on alphabet [ 1 .. 8 ] with 6 states>,
```

```
                        <Mealy element on alphabet [ 1 .. 8 ] with 6 states>,
                        <Mealy element on alphabet [ 1 .. 8 ] with 6 states> ]
gap> RattaggiGroup.2_21.1^v;
<Mealy element on alphabet [ 1 .. 8 ] with 6 states>
gap> Range(v);
<state-closed group over [ 1, 2, 3, 4, 5, 6, 7, 8 ] with 3 generators>
gap> PermGroup(last,1);
Group([ (3,4)(5,6), (1,7,8,2)(3,4,6,5), (1,7,5,3)(2,8,6,4) ])
gap> DisplayCompositionSeries(last);
G (3 gens, size 1344)
 | A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~ 2A(1,7) = U(2,7) ~ A(2,2) = L(3,2)
S (3 gens, size 8)
 | Z(2)
S (2 gens, size 4)
 | Z(2)
S (1 gens, size 2)
 | Z(2)
1 (0 gens, size 1)
```

9.5.3   VHGroup

▷ VHGroup(*l1, l2, ...*)   (function)

Returns: A new VH group.

This function constructs the VH group specified by the squares *l1, l2, ...* Each *li* is a list of length 4, of the form [v,h,v',h']. Here the entries are indices of vertical, respectively horizontal generators, if positive; and their inverses if negative.

─────────────────────────────── Example ───────────────────────────────
```
gap> # the Baby-Aleshin group
gap> g := VHGroup([[1,1,-2,-1],[1,2,-3,-2],[2,1,-3,-1],
                   [2,2,-2,-2],[3,1,-1,-2],[3,2,-1,-1]]);
<VH group on the generators [ a1, a2, a3 | b1, b2 ]>
gap> Display(g);
generators = [ a1, a2, a3, b1, b2 ]                            ## relators = [
 a1*b1*a2^-1*b1^-1,
 a1*b2*a3^-1*b2^-1,
 a2*b1*a3^-1*b1^-1,
 a2*b2*a2^-1*b2^-1,
 a3*b1*a1^-1*b2^-1,
 a3*b2*a1^-1*b1^-1 ]
```

9.5.4   IsIrreducibleVHGroup

▷ IsIrreducibleVHGroup(*g*)   (property)

Returns: Whether *g* is an irreducible lattice.

A VH group is *irreducible* if its projections on both trees is dense.

─────────────────────────────── Example ───────────────────────────────
```
gap> Display(RattaggiGroup.2_21);
generators = [ a1, a2, a3, b1, b2, b3, b4 ]
relators = [
 a1*b1*a1^-1*b1^-1,
```

```
   a1*b2*a1^-1*b2^-1,
   a1*b3*a1^-1*b4^-1,
   a1*b4*a2^-1*b3^-1,
   a1*b4^-1*a2^-1*b3,
   a2*b1*a2^-1*b2^-1,
   a2*b2*a3^-1*b1,
   a2*b3*a2^-1*b4,
   a2*b2^-1*a3*b1^-1,
   a3*b1*a3*b3^-1,
   a3*b2*a3*b4^-1,
   a3*b3*a3*b4 ]
gap> IsIrreducibleVHGroup(RattaggiGroup.2_21);
true
```

9.5.5 MaximalSimpleSubgroup

▷ MaximalSimpleSubgroup(*g*)   (attribute)

Returns: A maximal simple subgroup of *g*, if possible.

A VH group is never simple, but in favourable cases it admits a finite-index simple subgroup, see [BM97]. This function attempts to construct such a subgroup. It returns `fail` if no such subgroup can be found.

The current implementation is not smart enough to work with the Rattaggi examples (see `IsVirtuallySimpleGroup` (7.3.4)).

Chapter 10

FR implementation details

FR creates new categories for the various objects considered in the package. The first category is FRObject; all objects are in this category, and have an `Alphabet` method.

There are two categories below: `FRMachine` and `FRElement`. An `FRMachine` must have a `StateSet`, and methods for `Output` and a `Transition`. An `FRElement` must have an underlying `FRMachine` and `InitialState`, and `Output` and a `Transition` that use the initial state.

A self-similar group is simply a collections category of FR elements which is also a group.

10.1 The family of FR objects

All FR objects have an associated `AlphabetOfFRObject` (10.1.3).

10.1.1 FRMFamily

▷ FRMFamily(*obj*)   (operation)
    Returns: the family of FR machines on alphabet *obj*.

The family of an FR object is the arity of the tree on which elements cat act; in other words, there is one family for each alphabet.

10.1.2 FREFamily

▷ FREFamily(*obj*)   (operation)
    Returns: the family of FR elements on alphabet *obj*.

The family of an FR object is the arity of the tree on which elements cat act; in other words, there is one family for each alphabet.

The argument may be an FR machine, an alphabet, or a family of FR machines.

10.1.3 AlphabetOfFRObject

▷ AlphabetOfFRObject(*obj*)   (operation)
▷ AlphabetOfFRAlgebra(*obj*)   (operation)
▷ AlphabetOfFRSemigroup(*obj*)   (operation)
▷ Alphabet(*obj*)   (operation)
    Returns: the alphabet associated with *obj*.

This command applies to the family of any FR object, or to the object themselves. Alphabets are returned as lists, and in pratice are generally of the form `[1..n]`.

10.1.4 AsPermutation (FR object)

▷ AsPermutation(*o*) (method)

This method takes as argument an FR object *o*: machine, element, or group, and produces an equivalent object whose outputs are permutations. In particular, it converts Mealy machines from domain representation to int representation.

If this is not possible, the method returns `fail`.

10.1.5 AsTransformation (FR object)

▷ AsTransformation(*o*) (method)

This method takes as argument an FR object *o*: machine, element, or group, and produces an equivalent object whose outputs are transformations. In particular, it converts Mealy machines from domain representation to int representation.

Since transformations can never be inverted by GAP, even when they are invertible, this function returns a monoid when applied to a full SC group.

10.2 Filters for `FRObject`s

10.2.1 IsGroupFRMachine

▷ IsGroupFRMachine(*obj*) (property)
▷ IsMonoidFRMachine(*obj*) (property)
▷ IsSemigroupFRMachine(*obj*) (property)
   Returns: `true` if *obj* is an FR machine whose stateset is a free group/monoid/semigroup.

This function is the acceptor for those functionally recursive machines whose stateset (accessible via `StateSet` (3.4.1)) is a free group, monoid or semigroup. The generating set of its stateset is accessible via `GeneratorsOfFRMachine` (3.4.2).

10.2.2 IsFRMachineStrRep

▷ IsFRMachineStrRep(*obj*) (filter)
   Returns: `true` if *obj* is a standard (group,monoid,semigroup) FR machine.

There is a free object `free`, of rank *N*, a list `transitions` of length *N*, each entry a list, indexed by the alphabet, of elements of `free`, and a list `output` of length `N` of transformations or permutations of the alphabet.

10.2.3 IsMealyMachine

▷ IsMealyMachine(*obj*) (filter)
   Returns: `true` if *obj* is a Mealy machine.

This function is the acceptor for the *Mealy machine* subcategory of *FR machine*s.

10.2.4 IsMealyElement

▷ IsMealyElement(*obj*) (filter)

Returns: `true` if *obj* is a Mealy element.

This function is the acceptor for the *Mealy element* subcategory of *FR element*s.

10.2.5 IsMealyMachineIntRep

▷ IsMealyMachineIntRep(*obj*) (filter)

Returns: `true` if *obj* is a Mealy machine in integer representation.

A Mealy machine in *integer* representation has components `nrstates`, `transitions`, `output` and optionally `initial`.

Its stateset is `[1..nrstates]`, its transitions is a matrix with `transitions[s][x]` the transition from state `s` with input `x`, its output is a list of transformations or permutations, and its initial state is an integer.

10.2.6 IsMealyMachineDomainRep

▷ IsMealyMachineDomainRep(*obj*) (filter)

Returns: `true` if *obj* is a Mealy machine in domain representation.

A Mealy machine in *domain* representation has components `states`, `transitions`, `output` and optionally `initial`.

Its states is a domain, its transitions is a function with `transitions(s,x)` the transition from state `s` with input `x`, its output is a function with `output(s,x)` the output from input `x` in state `s`, and its initial state is an elemnent of `states`.

10.2.7 IsVectorFRMachineRep

▷ IsVectorFRMachineRep(*obj*) (filter)

Returns: `true` if *obj* is a vector machine

A *vector machine* is a representation of a linear machine by a finite-dimensional vector space (implicit in the structure), a transition tensor (represented as a matrix of matrices), and an output vector (represented as a list).

10.2.8 IsAlgebraFRMachineRep

▷ IsAlgebraFRMachineRep(*obj*) (filter)

Returns: `true` if *obj* is an algebra machine

An *algebra machine* is a representation of a linear machine by a finitely generated free algebra, a tensor of transitions, indexed by generator index and two alphabet indices, and an output vector, indexed by a generator index.

The transition tensor's last two entries are the 0 and 1 matrix over the free algebra, and the output tensor's last two entries are the 0 and 1 elements of the left acting domain.

10.2.9 IsLinearFRMachine

▷ IsLinearFRMachine(*obj*) (filter)

Returns: `true` if *obj* is a linear machine.

This function is the acceptor for the *linear machine* subcategory of *FR machine*s.

10.2.10   IsLinearFRElement

▷ IsLinearFRElement(*obj*)   (filter)
   Returns: `true` if *obj* is a linear element.
   This function is the acceptor for the *linear element* subcategory of *FR element*s.

10.2.11   IsFRElement

▷ IsFRElement(*obj*)   (filter)
▷ IsSemigroupFRElement(*obj*)   (filter)
▷ IsMonoidFRElement(*obj*)   (filter)
▷ IsGroupFRElement(*obj*)   (filter)
   Returns: `true` if *obj* is an FR element.
   This filter is the acceptor for the *functionally recursive element* category.
   It implies that *obj* has an underlying FR machine, may act on sequences, and has a recursive `DecompositionOfFRElement` (4.2.6).
   The next filters specify the type of free object the stateset of *obj* is modelled on.

10.2.12   IsFRMealyElement

▷ IsFRMealyElement(*obj*)   (filter)
▷ IsSemigroupFRMealyElement(*obj*)   (filter)
▷ IsMonoidFRMealyElement(*obj*)   (filter)
▷ IsGroupFRMealyElement(*obj*)   (filter)
▷ UnderlyingMealyElement(*obj*)   (attribute)
   Returns: `true` if *obj* is an FR element.
   This filter is the acceptor for the *functionally recursive element* category, with an additional Mealy element stored as attribute for faster calculations. It defines a subcategory of `IsFRElement` (10.2.11). This additional Mealy element may be obtained as `UnderlyingMealyElement(obj)`.
   The next filters specify the type of free object the stateset of *obj* is modelled on.

10.2.13   IsFRObject

▷ IsFRObject(*obj*)   (filter)
   Returns: `true` if *obj* is an FR machine or element.
   This function is the acceptor for the most general FR category (which splits up as `IsFRMachine` (10.2.14) and `IsFRElement` (10.2.11)).
   It implies that *obj* has an attribute `AlphabetOfFRObject` (10.1.3).

10.2.14   IsFRMachine

▷ IsFRMachine(*obj*)   (filter)
   Returns: `true` if *obj* is an FR machine.
   This function is the acceptor for the *functionally recursive machine* category. It splits up as `IsGroupFRMachine` (10.2.1), `IsSemigroupFRMachine` (10.2.1), `IsMonoidFRMachine` (10.2.1) and `IsMealyMachine` (10.2.3)).

It implies that *obj* has attributes `StateSet` (3.4.1), `GeneratorsOfFRMachine` (3.4.2), and `WreathRecursion` (3.4.6); the last two are usually not used for Mealy machines.

10.2.15 IsInvertible

▷ `IsInvertible(m)` (property)

Returns: `true` if `m` is an invertible FR machine.

This function accepts invertible FR machines, i.e. machines `m` such that (m, q) is an invertible transformation of the alphabet for all q in the stateset of `m`.

```
                                 Example
  gap> m := FRMachine([[[],[]]],[(1,2)]);
  <FR machine with alphabet [ 1, 2 ] on Group( [ f1 ] )>
  gap> IsInvertible(m);
  true
  gap> m := FRMachine([[[],[]]],[[1,1]]);
  <FR machine with alphabet [ 1, 2 ] on Monoid( [ m1 ], ... )>
  gap> IsInvertible(m);
  false
```

10.2.16 IsFRGroup

▷ `IsFRGroup(obj)` (filter)
▷ `IsFRMonoid(obj)` (filter)
▷ `IsFRSemigroup(obj)` (filter)

Returns: `true` if *obj* is a FR group/monoid/semigroup.

These functions accept *self-similar groups/monoids/semigroups*, i.e. groups/monoids/semigroups whose elements are FR elements.

10.2.17 IsFRAlgebra

▷ `IsFRAlgebra(obj)` (filter)
▷ `IsFRAlgebraWithOne(obj)` (filter)

Returns: `true` if *obj* is a FR algebra [with one].

These functions accept *self-similar algebras [with one]*, i.e. algebras whose elements are linear FR elements.

10.3 Some of the algorithms implemented

Few calculations with infinite groups can be guaranteed to terminate — and especially to terminate within reasonable time. This section describes some of the algorithms implemented in FR.

10.3.1 FRMachineRWS

▷ `FRMachineRWS(m)` (attribute)

Returns: A record containing a rewriting system for `m`.

Elements of an FR machine are compared using a rewriting system, which records all known relations among states of the machine.

One may specify via an optional argument `:fr_maxlen:=n`, the maximal length of rules to be added. By default, this maximum length is 5.

```
———————————————— Example ————————————————
gap> n := FRMachine(["a","b"],[[[],[2]],[[],[1]]],[(1,2),()]);
<FR machine with alphabet [ 1, 2 ] on Group( [ a, b ] )>
gap> FRMachineRWS(n);
rec( rws := Knuth Bendix Rewriting System for Monoid( [ a^-1, a, b^-1, b
    ], ... ) with rules
  [ [ a^-1*a, <identity ...> ], [ a*a^-1, <identity ...> ],
    [ b^-1*b, <identity ...> ], [ b*b^-1, <identity ...> ] ],
  tzrules := [ [ [ 1, 2 ], [ ] ], [ [ 2, 1 ], [ ] ], [ [ 3, 4 ], [ ] ],
      [ [ 4, 3 ], [ ] ] ], letterrep := function( w ) ... end,
  pi := function( w ) ... end, reduce := function( w ) ... end,
  addgprule := function( w ) ... end, commit := function(  ) ... end,
  restart := function(  ) ... end )
```

10.3.2 Order of FR elements

The order of an FR element `e` is computed as follows: the tree is traversed recursively, filling it as follows. For each cycle of `e` on the first level, the product of the states on that cycle are computed. The method continues recursively with that product, remembering the order of the cycle. Once a state reappears in the traversal, FR determines if one instance of the state is in the subtree of the other, and if so whether the top one was raised to a non-trivial power to yield the second one as a state. If this happens, then `e` has infinite order. Otherwise, the least common multiple of the powers that appeared in the traversal is returned.

This method is guaranteed to succeed if `e` is a bounded element. To improve chances of success, FR first computes whether `e` acts by vertex transformations belonging to an abelian group; and if so, if `e` is conjugate to an adding machine. In that case too, `e` has infinite order.

10.3.3 Membership in semigroups

The following algorithm is used to determine whether a Mealy element belongs to a self-similar group. The corresponding problem of membership of an FR element in a state-closed self-similar group can be much simpler, because an FR element has an associated FR machine, all of whose states belong to the group.

Assume the group is given by generators. FR attempts to express the given Mealy element as a product of generators. At the same time, it constructs epimorphisms to finite groups. It is hoped that one of these two processes will stop.

This amounts, in fact, to the following. Consider a group G acting on a tree. It has a natural, profinite closure $\overline{G}$. The algorithm then attempts either to write an element x as a product of generators of G, or to show that x does not belong to $\overline{G}$.

There are groups G such that $\overline{G} \setminus G$ contains Mealy machines. For these, the above algorithm will not terminate.

An additional refinement is implemented for bounded groups (see `IsBoundedFRSemigroup` (7.2.14)). The `Germs` (5.2.24) of an element are computed, and compared to the germs of elements in the group.

Finally, for a group that possesses self-similar data (see Section 10.3.9), very fast methods are implemented to recognize and express an FR element as a product of generators.

10.3.4 The conjugacy problem

The conjugacy problem for self-similar branch groups has been implemented by Thorsten Groth, as part of his Diploma Thesis. His code is integrated in FR.

Specialized algorithms are implemented for the Grigorchuk and Gupta-Sidki groups, and a generic algorithm is implemented, which is however not guaranteed to succeed. The implementation follows [BBSZ13].

The following extra attibutes are part of his implementation:

10.3.5 OrbitSignalizer

▷ OrbitSignalizer(*g*) (attribute)

Returns: The Orbit Signalizer of the group element *g*

This attribute computes the orbit signalizer of an element. This is the set $OS(g) := \{g^{|Orb_g(v)|}@v \mid v \in X^*\}$ where X is the alphabet of the element g and $Orb_g(v)$ is the orbit of v under $\langle g \rangle$.

```
──────────────────────── Example ────────────────────────
  gap> a := MealyElement([[2,2],[2,2]],[(1,2),()],1);
  <Mealy element on alphabet [ 1 .. 2 ] with 2 states>
  gap> OrbitSignalizer(a);
  [ <Mealy element on alphabet [ 1 .. 2 ] with 2 states>, <Trivial Mealy element on alphabet [ 1 ..
```

DeclareAttribute("OrbitSignalizer", IsFRElement);

10.3.6 FRConjugacyAlgorithm

▷ FRConjugacyAlgorithm(*G*) (attribute)

Returns: A function which solves the conjugacy problem for *G*

This attribute stores a function in three arguments which computes a representative conjugator if exists or fail otherwise.

This attribute is not meant to have a standard setter but to be set if a specialized conjugacy algorithm for a certain group is discovered.

```
──────────────────────── Example ────────────────────────
  gap> f := FRConjugacyAlgorithm(GrigorchukGroup);
  function( G, g, h ) ... end
  gap> AssignGeneratorVariables(GrigorchukGroup);
  #I  Assigned the global variables [ "a", "b", "c", "d" ]
  gap> f(GrigorchukGroup,a,a^b);
  <Mealy element on alphabet [ 1 .. 2 ] with 5 states>
```

DeclareAttribute("FRConjugacyAlgorithm", IsFRGroup,2);

10.3.7 FRBranchGroupConjugacyData

▷ FRBranchGroupConjugacyData(*G*) (attribute)

Returns: The initial data for the branch algorithm for *G*

This attribute records the data for the branch algorithm. The record has the following components:

initial_conj_dic:

   Dictionary of already known conjugacy pairs with corresponding conjugator tuples. This has to cover at least the TorsionNucleus of *G*

Branchstructure

Usally calculated by the function BranchStructure

RepSystem

List of representatives of G/K where K is the branching subgroup of G

DeclareAttribute("FRBranchGroupConjugacyData", IsFRGroup,"mutable");

10.3.8 Order of groups

The order of an FR group is computed as follows: if all generators are finitary, then enumeration will succeed in computing the order. If the action of the group is primitive, and it comes from a bireversible automaton, then the Thompson-Wielandt theorem is tested against. This theorem states that, in our context (a group acting on a rooted tree, coming from a larger group acting transitively), if the group is finite then the stabilizer of a sphere of radius 2 is a p-group; see [BM00a, Proposition 2.1.1]. Then, FR attempts to find whether the group is level-transitive (in which case it would be infinite). Finally, it attempts to enumerate the group's elements, testing at the same time whether these elements have infinite order.

Needless to say, none except the first few steps are guaranteed to succeed.

10.3.9 Images and preimages of some groups in f.p. and l.p. groups

Contracting, branched groups admit finite L-presentations (see [Bar03a]), that is, presentations by finitely many generators, relators and endomorphisms; the (usual) relators are the images of the given relators under iteration by all endomorphisms.

Using the package NQL, it is possible to construct infinite nilpotent quotients of self-similar groups, and perform fast computations in them.

It is possible to construct, algorithmically, such an L-presentation from a self-similar groups; however, this algorithm has not been implemented yet, mainly because efficiency issues would make it usable only in very few cases.

For groups with an isomorphism to an L-presented group (constructed by IsomorphismLpGroup (7.2.31)), a fast method expresses group elements as words in the L-presented group's generators. It proceeds recursively on the decomposition of the element, mapping elements that are expressible by short words over the nucleus (usually length 1; length 3 is needed for the BrunnerSidkiVieiraGroup (9.1.13)) to their value in the L-presented group, and using the presentation's endomorphism to construct words with appropriate decompositions.

In particular, the algorithm will stop, returning fail, if during the recursion it reaches an element x such that x is a state of x but x does not belong to the nucleus.

10.3.10 Comparison of FR, Mealy, vector, and algebra elements

FR and Mealy elements can be compared quite efficiently, as long as they are distinct. The algorithm runs as follows: let the two elements be x and y. Considering both in turn, FR constructs the first entries of minimal Mealy elements expressing x and y; as soon as an output entry is distinct for x and for y, the status of $x < y$ is determined; and similarly for transition entries. Finally, if either of x or y is finite-state and the entries were identical up to that step, then the element with smallest stateset is considered smaller.

In this way, FR and Mealy elements can efficiently be compared. For Mealy elements, it suffices to follow their internal data; while for FR elements, this amounts to constructing Mealy elements approximating them to a sufficient precision so that they can be compared as such.

The algorithm first tries to test its arguments for equality; this test is not guaranteed to succeed.

A similar algorithm applies for linear elements. Here, one constructs vector element approximations; and compares, for ever-increasing values of i, first the output vectors of basis state i; then the transitions from state i to state j, for all $j \in \{1,\ldots,i\}$; then the transitions from state j to state i for all $j \in \{1,\ldots,i-1\}$.

10.3.11 Inverses of linear elements

It is probably difficult to compute the inverse of a vector element. The following approach is used: to compute the inverse of x, large (scalar) matrix approximations of x are computed; they are inverted using linear algebra; a vector element representing this inverse is guessed; and the guess is checked. As long as that check fails, larger approximations are computed.

Needless to say, this method need not succeed; for there are vector elements that are invertible, but whose inverse is not a vector element. A good test example appears in [Bac08]: consider the infinite matrix with 1's on the diagonal, and ω below the diagonal. This element admits an inverse if and only if ω is a root of unity. The complexity of the inverse grows as the degree of ω grows. Here is an illustation:

```
————————————————————————— Example —————————————————————————
gap> bacher := function(n)
  local f;
  f := CyclotomicField(n);
  return VectorElement(f,One(f)*[[[[1,0],[0,0]],
      [[0,0],[0,1]]],[[[0,1],[0,0]],[[1,0],[0,0]]]],[One(f),E(n)],[One(f),Zero(f)]]);
end;;
gap> Inverse(bacher(3));
<Linear element on alphabet CF(3)^2 with 4-dimensional stateset>
6 gap> Inverse(bacher(5));
<Linear element on alphabet CF(5)^2 with 6-dimensional stateset>
```

n	1	2	3	4	5	6	7	8	9	10
dimension		2	4	4	6	3	5	5	8	5
n	11	12	13	14	15	16	17	18	19	20
dimension	?	5	?	4	6	6	?	7	?	7
n	22	24	26	28	30	32	34	36	38	40
dimension	?	6	?	6	?	7	?	?	?	?

Table: Dimension of states of inverse

Chapter 11

Miscellanea

11.1 Generic operations

11.1.1 TensorSum

▷ TensorSum(*objects, ...*)   (function)

This function is similar in syntax to `DirectProduct` (**Reference: DirectProduct**), and delegates to `TensorSumOp`; its meaning depends on context, see e.g. `TensorSumOp` (3.5.4).

11.1.2 TensorProduct

▷ TensorProduct(*objects, ...*)   (function)

This function is similar in syntax to `DirectProduct` (**Reference: DirectProduct**), and delegates to `TensorProductOp`; its meaning depends on context, see e.g. `TensorProductOp` (3.5.5).

11.1.3 DirectSum

▷ DirectSum(*objects, ...*)   (function)

This function is similar in syntax to `DirectProduct` (**Reference: DirectProduct**), and delegates to `DirectSumOp`; its meaning depends on context, see e.g. `DirectSumOp` (3.5.6).

11.2 Periodic lists

11.2.1 PeriodicListsFamily

▷ PeriodicListsFamily   (family)
▷ IsPeriodicList   (filter)

The family, respectively filter, of `PeriodicList` (11.2.2)s.

11.2.2 PeriodicList

▷ PeriodicList(*preperiod[, period]*) (operation)
▷ PeriodicList(*list, i*) (operation)
▷ PeriodicList(*list, f*) (operation)
▷ CompressedPeriodicList(*preperiod[, period]*) (operation)
▷ CompressedPeriodicList(*list, i*) (operation)
▷ PrePeriod(*list*) (operation)
▷ Period(*list*) (operation)

These functions manipulate *periodic lists*, i.e. lists of infinite length such that elements follow a periodic order after some point.

The first command creates a periodic list, specified by its preperiod and period, which must both be lists. If the period is absent, this is actually a finite list.

The second command creates a periodic list by decreeing that the entries after the end of the list start again at position `i`.

The third command creates a list by applying function `f` to all elements of `l`.

The fourth and fifth command compress the newly created periodic list, see CompressPeriodicList (11.2.3).

The sixth and seventh commands return respectively the preperiod and period of a periodic list.

Most of the methods applied for lists have an obvious equivalent for periodic lists: List (**Reference: list and non-list difference**), Filtered (**Reference: Filtered**), First (**Reference: First**), ForAll (**Reference: ForAll**), ForAny (**Reference: ForAny**), Number (**Reference: Number**).

```
───────────────────── Example ─────────────────────
gap> l := PeriodicList([1],[2,3,4]);
[ 1, / 2, 3, 4 ]
gap> l[5];
2
gap> Add(l,100,3); l;
[ 1, 2, 100, / 3, 4, 2 ]
gap> Remove(l,5);
4
gap> l;
[ 1, 2, 100, 3, / 2, 3, 4 ]
gap> PrePeriod(l);
[ 1, 2, 100, 3 ]
gap> Period(l);
[ 2, 3, 4 ]
```

11.2.3 CompressPeriodicList

▷ CompressPeriodicList(*l*) (operation)

This function compresses a periodic list, in replacing the period by a minimal period, and shortening the preperiod. No value is returned, but the list `l` is modified. It remains equal (under =) to the original list.

```
───────────────────── Example ─────────────────────
gap> l := PeriodicList([1],[2,3,4,2,3,4]);
[ 1, / 2, 3, 4, 2, 3, 4 ]
```

```
gap> Add(l,4,5); l;
[ 1, 2, 3, 4, 4, / 2, 3, 4, 2, 3, 4 ]
gap> CompressPeriodicList(l);
gap> l;
[ 1, 2, 3, 4, / 4, 2, 3 ]
```

11.2.4 IsConfinal

▷ IsConfinal(`l, m`)   (operation)

    Returns: `true` if `l` and `m` are eventually equal.

    This function tests whether two lists are *confinal*, i.e. whether, after removal of the same suitable number of elements from both lists, they become equal.

```
───────────────── Example ─────────────────
gap> l := PeriodicList([1],[2,3,2,3]);
[ 1, / 2, 3, 2, 3 ]
gap> m := PeriodicList([0,1],[3,2]);
[ 0, 1, / 3, 2 ]
gap> IsConfinal(l,m);
true
```

11.2.5 ConfinalityClass

▷ ConfinalityClass(`l`)   (operation)

    Returns: The strictly periodic list with same tail as `l`.

    There exists a unique periodic list, with no preperiod, which is confinal (see `IsConfinal` (11.2.4)) to `l`. This strictly periodic list is returned by this command.

```
───────────────── Example ─────────────────
gap> l := PeriodicList([1],[2,3,2,3]);
[ 1, / 2, 3, 2, 3 ]
gap> ConfinalityClass(l);
[/ 3, 2 ]
```

11.2.6 LargestCommonPrefix

▷ LargestCommonPrefix(`c`)   (operation)

    Returns: The longest list that is a prefix of all elements of `c`.

    This command computes the longest (finite or periodic) list which is a prefix of all elements of `c`. The argument `c` is a collection of finite and periodic lists.

```
───────────────── Example ─────────────────
gap> LargestCommonPrefix([PeriodicList([1],[2,3,2,3]),[1,2,3,4]]);
[ 1, 2, 3 ]
```

11.3 Word growth

11.3.1 WordGrowth

▷ WordGrowth(`g, rec(options...)`)   (function)
▷ WordGrowth(`g: options...`)   (function)

▷ `OrbitGrowth(g, point[, limit])` (function)

▷ `Ball(g, radius)` (function)

▷ `Sphere(g, radius)` (function)

Returns: The word growth of the semigroup g.

This function computes the first terms of growth series associated with the semigroup g. The argument g can actually be a group/monoid/semigroup, or a list representing that semigroup's generating set.

The behaviour of `WordGrowth` is controlled via options passed in the second argument, which is a record. They can be combined when reasonable, and are:

`limit:=n`
to specify a limit radius;

`sphere:=radius`
to return the sphere of the specified radius, unless a radius was specified in `limit`, in which case the value is ignored;

`spheres:=maxradius`
to return the list of spheres of radius between 0 and the specified limit;

`spheresizes:=maxradius`
to return the list sizes of spheres of radius between 0 and the specified limit;

`ball:=radius`
to return the ball of the specified radius;

`balls:=maxradius`
to return the list of balls of radius between 0 and the specified limit;

`ballsizes:=maxradius`
to return the list sizes of balls of radius between 0 and the specified limit;

`indet:=z`
to return the `spheresizes`, as a polynomial in `z` (or the first indeterminate if `z` is not a polynomial;

`draw:=filename`
to create a rendering of the Cayley graph of g. Edges are given colours according to the cyclic ordering "red", "blue", "green", "gray", "yellow", "cyan", "orange", "purple". If `filename` is a string, the graph is appended, in `dot` format, to that file. Otherwise, the output is converted to Postscript using the program `neato` from the graphviz package, and displayed in a separate X window using the program display or rsvg-view. This works on UNIX systems.

It is assumed, but not checked, that graphviz and display/rsvg-view are properly installed on the system. The option `usesvg` requests the use of rsvg-view; by default, display is used.

`point:=p`
to compute the growth of the orbit of `p` under g, rather than the growth of g.

`act:=function(point,g) ... end`
to compute the action with a specific function, rather than the default `OnPoints`.

```
track:=true
```
> to keep track of a word in the generators that gives the element. This affects the "ball", "balls", "sphere" and "spheres" commands, where the result returned is a 3-element list: the first entry is the original results; the second entry is a homomorphism from a free group/monoid/semigroup; and the third entry contains the words corresponding to the first entry via the homomorphism.

If the first argument is an integer `n` and not a record, the command is interpreted as `WordGrowth(...,rec(spheresizes:=n))`.

`WordGrowth(...,rec(draw:=true))` may be abbreviated as `Draw(...)`; `WordGrowth(...,rec(ball:=n))` may be abbreviated as `Ball(...,n)`; `WordGrowth(...,rec(sphere:=n))` may be abbreviated as `Sphere(...,n)`;

```
─────────────────────────────── Example ───────────────────────────────
gap> WordGrowth(GrigorchukGroup,4);
[ 1, 4, 6, 12, 17 ]
gap> WordGrowth(GrigorchukGroup,rec(limit:=4,indet:=true));
17*x_1^4+12*x_1^3+6*x_1^2+4*x_1+1
gap> WordGrowth(GrigorchukGroup,rec(limit:=1,spheres:=true));
[ [ <Mealy element on alphabet [ 1, 2 ] with 1 state, initial state 1> ],
  [ d, b, c, a ] ]
gap> WordGrowth(GrigorchukGroup,rec(point:=[2,2,2]));
[ 1, 1, 1, 1, 1, 1, 1, 1 ]
gap> OrbitGrowth(GrigorchukGroup,[1,1,1]);
[ 1, 2, 2, 1, 1, 1 ]
gap> WordGrowth(GrigorchukGroup,rec(spheres:=4,point:=PeriodicList([],[2])));
[ [ [/ 2 ] ], [ [ 1, / 2 ] ], [ [ 1, 1, / 2 ] ], [ [ 2, 1, / 2 ] ],
  [ [ 2, 1, 1, / 2 ] ] ]
gap> WordGrowth([(1,2),(2,3)],rec(spheres:=infinity,track:=true));
[ [ [  ], [ (2,3), (1,2) ], [ (), (1,2,3), (1,3,2) ], [ (1,3) ] ],
  MappingByFunction( <free semigroup on the generators [ s1, s2 ]>, <group>, function( w ) ... en
  [ [  ], [ s2, s1 ], [ s2^2, s2*s1, s1*s2 ], [ s2*s1*s2 ] ] ]
```

Note that the orbit growth of `[/2]` is constant 1, while that of `[/1]` is constant 2. The following code would find the point with maximal orbit growth of a semigroup acting on the integers (for example, constructed with `PermGroup` (7.2.1)):

```
MaximalOrbitGrowth := function(g)
    local maxpt, growth, max;
    maxpt := LargestMovedPoint(g);
    growth := List([1..maxpt],n->WordGrowth(g:point:=n));
    max := Maximum(growth);
    return [max,Filtered([1..maxpt],n->growth[n]=max)];
end;
```

For example, the command `Draw(BasilicaGroup,rec(point:=PeriodicList([],[2,1]),limit:=3));`
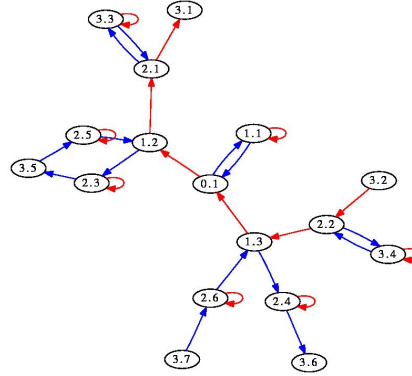
produces (in a new window) the following picture:

11.4 Finding short relations

11.4.1 ShortGroupRelations

▷ ShortGroupRelations(*g, n*)   (operation)
▷ ShortMonoidRelations(*g, n*)   (operation)
    Returns: A list of relations between words over *g*, of length at most *n*.

    This function assumes that *g* is a list of monoid elements. it searches for products of at most *n* elements over *g* that are equal.

    In its first form, it returns a list of words in a free group f of rank the length of *g*, that are trivial in *g*. The first argument may be a group, in which case its symmetric generating set is considered.

    In its second form, it returns a list of pairs [l,r], where l and r are words in a free monoid f of rank the length of *g*, that are equal in *g*. The first argument may be a monoid, in which case its monoid generating set is considered.

    This command does not construct all such pairs; rather, it returns a small set, in the hope that it may serve as a presentation for the monoid generated by *g*.

    The first element of the list returned is actually not a relation: it is a homomorphism from f to [the group/monoid generated by] *g*.

```
——————————————————— Example ———————————————————
gap> ShortGroupRelations(GrigorchukGroup,10);
[ [ x1, x2, x3, x4 ] -> [ a, b, c, d ],
  x1^2, x2^2, x3^2, x4^2, x2*x3*x4, x4*x1*x4*x1*x4*x1*x4*x1,
  x3*x1*x3*x1*x3*x1*x3*x1*x3*x1*x3*x1*x3*x1*x3*x1 ]
gap> ShortGroupRelations(GuptaSidkiGroup,9);
[ [ x1, x2 ] -> [ x, gamma ],
  x1^3, x2^3, x2*x1^-1*x2*x1^-1*x2*x1^-1*x2*x1^-1*x2*x1^-1*x2*x1^-1*
      x2*x1^-1*x2*x1^-1*x2*x1^-1,    x1^-1*x2^-1*x1^-1*x2^-1*x1^-1*x2^-1*
x1^-1*x2^-1*x1^-1*x2^-1*x1^-1*x2^-1*x1^-1*x2^-1*x1^-1*x2^-1*x1^-1*x2^-1 ]
```

11.4.2 ShortGroupWordInSet

▷ ShortGroupWordInSet(*g, s, n*)   (operation)
▷ ShortMonoidWordInSet(*g, s, n*)   (operation)
▷ ShortSemigroupWordInSet(*g, s, n*)   (operation)
    Returns: Words over *g* that express elements of *s*.

This command produces words in the free group/monoid/semigroup generated by g's generators that express elements of the set `s`. Elements of length at most `AbsoluteValue(n)` are searched; if `n` is non-negative then at most one element is returned. The value `n=infinity` is allowed.

The second argument may be either a list, a predicate (i.e. a function returning `true` or `false`) or an element.

The function returns a list of words in the free group/monoid/semigroup; the first entry of the list is a homomorphism from the free group/monoid/semigroup to g.

```
———————————————————————— Example ————————————————————————
gap> l := ShortMonoidWordInSet(Group((1,2),(2,3),(3,4)),
            [(1,2,3,4),(4,3,2,1)],-3);
[ MappingByFunction( <free monoid on the generators [ m1, m2, m3 ]>, Group(
    [ (1,2), (2,3), (3,4) ]), function( w ) ... end ), m3*m2*m1, m1*m2*m3 ]
gap> f := Remove(l,1);;
gap> List(l,x->x^f);
[ (1,2,3,4), (1,4,3,2) ]
gap> ShortMonoidWordInSet(GrigorchukGroup,
        [Comm(GrigorchukGroup.1,GrigorchukGroup.2)],4);
[ MappingByFunction( <free monoid on the generators [ m1, m2, m3, m4
    ]>, <self-similar monoid over [ 1 .. 2 ] with
    4 generators>, function( w ) ... end ), m1*m2*m1*m2 ]
```

11.5   Braid groups

11.5.1   SurfaceBraidFpGroup

▷ `SurfaceBraidFpGroup(n, g, p)`   (function)
▷ `PureSurfaceBraidFpGroup(n, g, p)`   (function)
   Returns: The [pure] surface braid group on `n` strands.

This function creates a finitely presented group, isomorphic to the [pure] braid group on `n` strands of the surface of genus g, with `p` punctures. In particular, `SurfaceBraidFpGroup(n,0,1)` is the usual braid group (on the disc).

The presentation comes from [Bel04]. The first $2g$ generators are the standard a_i, b_i surface generators; the next $n-1$ are the standard s_i braid generators; and the last are the extra z generators.

The pure surface braid group is the kernel of the natural map from the surface braid group to the symmetric group on `n` points, defined by sending a_i, b_i, z to the identity and s_i to the transposition `(i,i+1)`.

11.5.2   CharneyBraidFpGroup

▷ `CharneyBraidFpGroup(n)`   (function)
   Returns: The braid group on `n` strands.

This function creates a finitely presented group, isomorphic to the braid group on `n` strands (on the disc). It is isomorphic to `SurfaceBraidFpGroup(n,0,1)`, but has a different presentation, due to Charney ([Cha95]), with one generator per non-trivial permutation of `n` points.

11.5.3   ArtinRepresentation

▷ `ArtinRepresentation(n)`   (function)
   Returns: The braid group's representation on `FreeGroup(n)`.

This function creates a Artin's representatin, a homomorphism from the braid group on `n` strands (on the disc) into the automorphism group of a free group of rank `n`.

11.6 Dirichlet series

11.6.1 DirichletSeries (0)

▷ DirichletSeries()   (operation)
▷ DirichletSeries(*maxdeg*)   (operation)
▷ DirichletSeries(*indices, coeffs[, maxdeg]*)   (operation)
▷ DirichletSeries(*series, maxdeg*)   (operation)

Creates a new Dirichlet series, namely, a formal power series of the form $f(s) = \sum_{n \geq 1} a(n) n^{-s}$. Such series have a maximal degree, which may be `infinity`, and may be added or multiplied as polynomials.

11.6.2 DegreeDirichletSeries

▷ DegreeDirichletSeries(*f*)   (attribute)
   Returns: The maximal degree of a non-zero coefficient of `f`.

11.6.3 SpreadDirichletSeries

▷ SpreadDirichletSeries(*f, n*)   (attribute)
   Returns: The series $f(ns)$.

11.6.4 ShiftDirichletSeries

▷ ShiftDirichletSeries(*s, n*)   (attribute)
   Returns: The series $n^{-s} f(s)$.

11.6.5 ShrunkDirichletSeries

▷ ShrunkDirichletSeries(*f*)   (attribute)
   Returns: The series `f`, with maximal precision set to its maximal degree.

11.6.6 ZetaSeriesOfGroup

▷ ZetaSeriesOfGroup(*G*)   (attribute)
   Returns: The series \sum_{\chi\in\widehat G}(\dim G)^{-s}.

11.6.7 ValueOfDirichletSeries

▷ ValueOfDirichletSeries(*f, s*)   (attribute)
   Returns: The evaluation of `f` at `s`. Synonym for `Value`.

11.7 Projective representations

11.7.1 IsProjectiveRepresentation

▷ IsProjectiveRepresentation(*rep*) (property)
▷ IsLinearRepresentation(*rep*) (property)

A projective representation is a mapping to matrices, that is multiplicative up to scalars. This property is set by the following functions that create projective representations.

The second property describes those projective representations that are in fact homomorphisms.

11.7.2 ProjectiveRepresentationByFunction

▷ ProjectiveRepresentationByFunction(*group, matrixgroup, function*) (operation)
   Returns: A projective representation of *group*.

11.7.3 LinearRepresentationByImages

▷ LinearRepresentationByImages(*group, matrixgroup, src, dst*) (operation)
   Returns: A linear representation of *group*.

11.7.4 DegreeOfProjectiveRepresentation

▷ DegreeOfProjectiveRepresentation(*rep*) (operation)
   Returns: The dimension of the matrices in the image of *rep*.

11.7.5 ProjectiveExtension

▷ ProjectiveExtension(*rep, group*) (operation)
   Returns: A projective representation of *group* whose restriction to `Source(rep)` (which is a subgroup of *group*) is *rep*.

11.7.6 ProjectiveQuotient

▷ ProjectiveQuotient(*rep, hom*) (operation)
   Returns: A projective representation of `Image(hom)` that comes from *rep*.

11.8 Miscellanea

11.8.1 ForwardOrbit

▷ ForwardOrbit(*g, x*) (operation)
   Returns: The forward orbit of *x* under *g*.
   This operation computes the smallest list containing *x* and closed under application of *g*.

```
 ──────────────── Example ────────────────
 gap> ForwardOrbit((1,2,4,3),2);
 [ 2, 4, 3, 1 ]
 gap> ForwardOrbit(Transformation([4,4,5,2,1]),1);
 [ 1, 4, 2 ]
```

11.8.2 StringByInt

▷ StringByInt(*n[, b]*) (function)

Returns: A string representing *n* in base *b*.

This function converts a positive integer to string. It accepts an optional second argument, which is a base in which to print *n*. By default, *b* is 2.

11.8.3 PositionInTower

▷ PositionInTower(*t, x*) (function)

Returns: The largest index such that t[i] contains *x*.

This function assumes *t* is a descending tower of domains, such as that constructed by LowerCentralSeries. It returns the largest integer i such that t[i] contains *x*; in case the tower ends precisely with *x*, the value infinity is returned.

x can be an element or a subdomain of t[1].

11.8.4 RenameSubobjects

▷ RenameSubobjects(*obj, refobj*) (function)

This function traverses *obj* if it is a list or a record, and, when it finds an element which has no name, but is equal (in the sense of =) to an element of *refobj*, assigns it the name of that element.

```
———————————————— Example ————————————————
gap> trivial := Group(());; SetName(trivial,"trivial");
gap> a := List([1..10],i->Group(Random(SymmetricGroup(3))));
[ Group([ (2,3) ]), Group([ (2,3) ]), Group([ (1,3) ]), Group([ (1,3) ]),
  Group([ (1,3,2) ]), Group([ (1,3,2) ]), Group([ (1,2) ]), Group(()),
  Group([ (2,3) ]), Group([ (1,3,2) ]) ]
gap> RenameSubobjects(a,[trivial]); a;
[ Group([ (2,3) ]), Group([ (2,3) ]), Group([ (1,3) ]), Group([ (1,3) ]),
  Group([ (1,3,2) ]), Group([ (1,3,2) ]), Group([ (1,2) ]), trivial,
  Group([ (2,3) ]), Group([ (1,3,2) ]) ]
```

11.8.5 CoefficientsInAbelianExtension

▷ CoefficientsInAbelianExtension(*x, b, G*) (function)

Returns: The coefficients in *b* of the element *x*, modulo *G*.

If *b* is a list of group elements $b_1, \ldots, b_k$, and $H = \langle G, b_1, \ldots, b_k \rangle$ contains *G* as a normal subgroup, and H/G is abelian and $x \in H$, then this function computes exponents $e_1, \ldots, e_k$ such that $\prod b_i^{e_i} G = xG$.

11.8.6 MagmaEndomorphismByImagesNC

▷ MagmaEndomorphismByImagesNC(*f, im*) (function)

Returns: An endomorphism of *f*.

This function constructs an endomorphism of the group, monoid or semigroup *f* specified by sending generator number *i* to the *i*th entry in *im*. It is a shortcut for a call to GroupHomomorphismByImagesNC or MagmaHomomorphismByFunctionNC(...,MappedWord(...)).

11.8.7   MagmaHomomorphismByImagesNC

▷ MagmaHomomorphismByImagesNC(`f, g, im`)   (function)

Returns: An homomorphism from `f` to `g`.

This function constructs a homomorphism of the group,monoid or semi-group `f` specified by sending generator number `i` to the `i`th entry in `im`.   It is a shortcut for a call to `GroupHomomorphismByImagesNC` or `MagmaHomomorphismByFunctionNC(...,MappedWord(...))`.

11.8.8   Draw (poset)

▷ Draw(`p`)   (function)
▷ HeightOfPoset(`p`)   (function)

Returns: The length of a maximal chain in the poset.

11.8.9   IsFIFO

▷ IsFIFO   (filter)
▷ NewFIFO(`[l]`)   (operation)
▷ Add(`f, i`)   (operation)
▷ Append(`f, l`)   (operation)

These functions create and extend FIFOs, i.e. first-in first-out data structures.

The first command creates a FIFO, with an optional list initializing it.

The second and third commands add an element, or append a list, to the FIFO.

Elements are removed via `NextIterator(f)`, and the FIFO is tested for emptyness via `IsDoneIterator(f)`. Thus, a typical use is the following code, which tests in breadth-first manner that all numbers in `[1..1000]` have a successor which is prime:

```
 ———————————— Example ————————————
  gap> f := NewFIFO([1..10000]);
  <iterator>
  gap> for i in f do if not IsPrime(i) then Add(f,i+1); fi; od;
```

11.8.10   ProductIdeal

▷ ProductIdeal(`a, b`)   (function)
▷ ProductBOIIdeal(`a, b`)   (function)

Returns: the product of the ideals `a` and `b`.

The first command computes the product of the left ideal `a` and the right ideal `b`. If they are not appropriately-sided ideals, the command first attempts to convert them.

The second command assumes that the ring of these ideals has a basis made of invertible elements. It is then much easier to compute the product.

11.8.11   DimensionSeries

▷ DimensionSeries(`a[, n]`)   (function)

Returns: A nested list of ideals in the algebra-with-one `a`.

This command computes the powers of the augmentation ideal of a, and returns their list. The list stops when the list becomes stationary.

The optional second argument gives a limit to the number of terms to put in the series.

```
───────────────────── Example ─────────────────────
gap> a := ThinnedAlgebraWithOne(GF(2),GrigorchukGroup);
<self-similar algebra-with-one on alphabet GF(2)^2 with 4 generators>
gap> q := MatrixQuotient(a,3);
<algebra-with-one of dimension 22 over GF(2)>
gap> l := DimensionSeries(q);
[ <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (5 generators)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 21)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 18)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 14)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 10)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 6)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 3)>,
  <two-sided ideal in <algebra-with-one of dimension 22 over GF(2)>, (dimension 1)>,
  <algebra of dimension 0 over GF(2)> ]
```

11.8.12   AlgebraHomomorphismByFunction

▷ AlgebraHomomorphismByFunction(A, B, f)   (operation)
▷ AlgebraWithOneHomomorphismByFunction(A, B, f)   (operation)
   Returns: A homomorphism from the algebra A to the algebra B.

These functions construct an algebra homomorphism from a one-argument function. They do not check that the function actually defines a homomorphism.

```
───────────────────── Example ─────────────────────
gap> A := MatrixAlgebra(Rationals,2);
( Rationals^[ 2, 2 ] )
gap> e1 := AlgebraHomomorphismByFunction(Rationals,A,f->[[f,0],[0,0]]);
MappingByFunction( Rationals, ( Rationals^[ 2, 2 ] ), function( f ) ... end )
gap> 11^e1;
[ [ 11, 0 ], [ 0, 0 ] ]
```

11.8.13   IsFpLieAlgebra

▷ IsFpLieAlgebra   (filter)

The category of Lie algebras coming from a finitely presented group. They appear as the JenningsLieAlgebra (**Reference: JenningsLieAlgebra**) of a finitely presented group.

If G is an infinite, finitely presented group, then the original implementation of JenningsLieAlgebra (**Reference: JenningsLieAlgebra**) does not return. On the other hand, the implementation in FR constructs a graded object, for which the graded components are computed on-demand; see JenningsLieAlgebra (11.8.14).

11.8.14   JenningsLieAlgebra

▷ JenningsLieAlgebra(ring, fpgroup)   (operation)
   Returns: The Jennings Lie algebra of fpgroup.

This method does not compute the Jennings Lie algebra *per se*; it merely constructs a placeholder to contain the result.

```
——————————— Example ———————————
  gap> f := FreeGroup(4);
  <free group on the generators [ f1, f2, f3, f4 ]>
  gap> surfacegp := f/[Comm(f.1,f.2)*Comm(f.3,f.4)];
  <fp group of size infinity on the generators [ f1, f2, f3, f4 ]>
  gap> j := JenningsLieAlgebra(Rationals,surfgp);
  <FP Lie algebra over Rationals>
  gap> List([1..4],Grading(j).hom_components);
  [ <vector space over Rationals, with 4 generators>,
    <vector space over Rationals, with 5 generators>,
    <vector space over Rationals, with 16 generators>,
    <vector space over Rationals, with 45 generators> ]
  gap> B := Basis(Grading(j).hom_components(1));
  gap> B[1]*B[2]+B[3]*B[4];
  <zero Lie element>
```

11.8.15 SolutionMatModN

▷ SolutionMatModN(*mat, vec, N*)   (operation)

Solve the linear system sol*mat=vec modulo *N*. The arguments are assumed to be an integer matrix and vector. Either returns an integer solution, or fail if no such solution exists.

11.8.16 SolutionMatMod1

▷ SolutionMatMod1(*mat, vec*)   (operation)

Solve the linear system sol*mat=vec in Q/Z. The arguments are assumed to be rational matrices. Assuming there are finitely many solutions, returns them all.

11.8.17 CyclotomicByArgument

▷ CyclotomicByArgument(*q*)   (operation)

Returns: The cyclotomic field element equal to $\exp(2\pi i q)$.

11.8.18 ArgumentOfCyclotomic

▷ ArgumentOfCyclotomic(*z*)   (operation)

Returns: The rational q such that $\exp(2\pi i q) = z$.

11.9 User settings

11.9.1 InfoFR

▷ InfoFR   (info class)

This is an `Info` class for the package FR. The command `SetInfoLevel(InfoFR,1);` switches on the printing of some information during the computations of certain FR functions; in particular all automatic conversions between FR machines and Mealy machines.

The command `SetInfoLevel(InfoFR,2);` requests a little more information, and in particular prints intermediate results in potentially long calculations such as `NucleusOfFRSemigroup` (7.2.19).

The command `SetInfoLevel(InfoFR,3);` ensures that FR will print information every few seconds or so. This is useful to gain confidence that the program is not stuck due to a programming bug by the author of FR.

11.9.2 SEARCH@

▷ `SEARCH@` (global variable)

This variable controls the search mechanism in FR groups. It is a record with in particular entries `radius` and `depth`.

`radius` limits the search in FR groups to balls of that radius in the generating set. For example, the command `x in G` will initiate a search in `G` to attempt to express `x` as a reasonably short word in the generators of `G`.

`depth` limits the level of the tree on which quotients of FR groups should be considered. Again for the command `x in G`, deeper and deeper quotients will be considered, in the hope of finding a quotient of `G` to which `x` does not belong.

A primitive mechanism is implemented to search alternatively for a quotient disproving `x in G` and a word proving `x in G`.

When the limits are reached and the search was unsuccessful, an interactive `Error()` is raised, to let the user increase their values.

Specific limits can be passed to any command via the options `FRdepth` and `FRradius`, as for example in `Size(G:FRdepth:=3,FRradius:=5)`.

References

[Ale83]   S. V. Aleshin. A free group of finite automata. *Vestnik Moskov. Univ. Ser. I Mat. Mekh.*, (4):12–14, 1983. 7, 97, 99

[Bac08]   R. Bacher. Determinants related to Dirichlet characters modulo 2, 4 and 8 of binomial co-efficients and the algebra of recurrence matrices. *Internat. J. Algebra Comput.*, 18(3):535–566, 2008. 106, 121

[Bar03a] L. Bartholdi. Endomorphic presentations of branch groups. *J. Algebra*, 268(2):419–443, 2003. 84, 96, 120

[Bar03b] L. Bartholdi. A Wilson group of non-uniformly exponential growth. *C. R. Math. Acad. Sci. Paris*, 336(7):549–554, 2003. 98

[Bar06]   L. Bartholdi. Branch rings, thinned rings, tree enveloping rings. *Israel J. Math.*, 154:93–139, 2006. 104

[Bar10]   L. Bartholdi. Self-similar lie algebras. arXiv:math/1003.1125, 2010. 105

[BBSZ13] I. V. Bondarenko, N. V. Bondarenko, S. N. Sidki, and F. R. Zapata. On the conjugacy problem for finite-state automorphisms of regular rooted trees. *Groups Geom. Dyn.*, 7(2):323–355, 2013. With an appendix by Rapha{\"e}l M. Jungers. 119

[BEH08]   L. Bartholdi, B. Eick, and R. Hartung. A nilpotent quotient algorithm for certain infinitely presented groups and its applications. *Internat. J. Algebra Comput.*, 18(8):1321–1344, 2008. 84

[Bel04]   P. Bellingeri. On presentations of surface braid groups. *J. Algebra*, 274(2):543–563, 2004. 128

[BG02]   L. Bartholdi and R. I. Grigorchuk. On parabolic subgroups and Hecke algebras of some fractal groups. *Serdica Math. J.*, 28(1):47–90, 2002. 96, 99

[BGN03]   L. Bartholdi, R. I. Grigorchuk, and V. Nekrashevych. From fractal groups to fractal sets. In *Fractals in Graz 2001*, Trends Math., pages 25–118. Birkhäuser, Basel, 2003. 6

[BGŠ03]   L. Bartholdi, R. I. Grigorchuk, and Z. Šuniḱ. Branch groups. In *Handbook of algebra, Vol. 3*, pages 989–1112. North-Holland, Amsterdam, 2003. 6, 94

[BKN+12] M. Begue, D. J. Kelleher, A. Nelson, H. Panzo, R. Pellico, and A. Teplyaev. Random walks on barycentric subdivisions and the Strichartz hexacarpet. *Exp. Math.*, 21(4):402–417, 2012. 101

[BM97]   M. Burger and S. Mozes. Finitely presented simple groups and products of trees. *C. R. Acad. Sci. Paris Sér. I Math.*, 324(7):747–752, 1997. 112

[BM00a]   M. Burger and S. Mozes. Groups acting on trees: from local to global structure. *Inst. Hautes Études Sci. Publ. Math.*, (92):113–150 (2001), 2000. 86, 99, 110, 120

[BM00b]   M. Burger and S. Mozes. Lattices in product of trees. *Inst. Hautes Études Sci. Publ. Math.*, (92):151–194 (2001), 2000. 86, 99, 110

[BR08]   L. Bartholdi and I. I. Reznykov. A Mealy machine with polynomial growth of irrational degree. *Internat. J. Algebra Comput.*, 18(1):59–82, 2008. 104

[BRS06]   L. Bartholdi, I. I. Reznykov, and V. I. Sushchansky. The smallest Mealy automaton of intermediate growth. *J. Algebra*, 295(2):387–414, 2006. 103

[BS62]   G. Baumslag and D. Solitar. Some two-generator one-relator non-Hopfian groups. *Bull. Amer. Math. Soc.*, 68:199–201, 1962. 102

[BŠ01]   L. Bartholdi and Z. Šuniḱ. On the word and period growth of some groups of tree automorphisms. *Comm. Algebra*, 29(11):4923–4964, 2001. 94

[BSV99]   A. M. Brunner, S. N. Sidki, and A. C. Vieira. A just nonsolvable torsion-free group defined on the binary tree. *J. Algebra*, 211(1):99–114, 1999. 96

[BV05]   L. Bartholdi and B. Virág. Amenability via random walks. *Duke Math. J.*, 130(1):39–56, 2005. 7, 92

[Cha95]   R. Charney. Geodesic automation and growth functions for Artin groups of finite type. *Math. Ann.*, 301(2):307–324, 1995. 128

[Dah05]   F. Dahmani. An example of non-contracting weakly branch automaton group. In *Geometric methods in group theory*, volume 372 of *Contemp. Math.*, pages 219–224. Amer. Math. Soc., Providence, RI, 2005. 100

[Ers04]   A. Erschler. Boundary behavior for groups of subexponential growth. *Ann. of Math. (2)*, 160(3):1183–1210, 2004. 95

[FG85]   J. Fabrykowski and N. Gupta. On groups with sub-exponential growth functions. *J. Indian Math. Soc. (N.S.)*, 49(3-4):249–256 (1987), 1985. 99

[FG91]   J. Fabrykowski and N. Gupta. On groups with sub-exponential growth functions. II. *J. Indian Math. Soc. (N.S.)*, 56(1-4):217–228, 1991. 99

[GM05]   Y. Glasner and S. Mozes. Automata and square complexes. *Geom. Dedicata*, 111:43–64, 2005. 99

[Gri80]   R. I. Grigorchuk. On Burnside's problem on periodic groups. *Funktsional. Anal. i Prilozhen.*, 14(1):53–54, 1980. 7, 95

[Gri84]   R. I. Grigorchuk. Degrees of growth of finitely generated groups and the theory of invariant means. *Izv. Akad. Nauk SSSR Ser. Mat.*, 48(5):939–985, 1984. 95

[GS83]   N. Gupta and S. N. Sidki. On the Burnside problem for periodic groups. *Math. Z.*, 182(3):385–388, 1983. 7, 98

[GŠ06]   R. I. Grigorchuk and Z. Šuniḱ. Asymptotic aspects of Schreier graphs and Hanoi Towers groups. *C. R. Math. Acad. Sci. Paris*, 342(8):545–550, 2006. 100

[GŻ02]   R. I. Grigorchuk and A. Żuk. On a torsion-free weakly branch group defined by a three state automaton. *Internat. J. Algebra Comput.*, 12(1-2):223–246, 2002. International Conference on Geometric and Combinatorial Methods in Group Theory and Semigroup Theory (Lincoln, NE, 2000). 7, 92

[Lys85]   I. G. Lysënok. A set of defining relations for the Grigorchuk group. *Mat. Zametki*, 38(4):503–516, 634, 1985. 95

[Mam03]   M. J. Mamaghani. A fractal non-contracting class of automata groups. *Bull. Iranian Math. Soc.*, 29(2):51–64, 92, 2003. 101

[MNS00]   O. Macedońska, V. V. Nekrashevych, and V. I. Sushchansky. Commensurators of groups and reversible automata. *Dopov. Nats. Akad. Nauk Ukr. Mat. Prirodozn. Tekh. Nauki*, (12):36–39, 2000. 42, 43

[Nek05]   V. Nekrashevych. *Self-similar groups*, volume 117 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 2005. 22, 97

[Nek08a]   V. Nekrashevych. Combinatorial models of expanding dynamical systems. arXiv:math.GR/0810.4936, 2008. 80

[Nek08b]   V. Nekrashevych. The julia set of a post-critically finite endomorphism of pc^2. arXiv:math.GR/0811.2777, 2008. 92

[Neu86]   P. M. Neumann. Some questions of Edjvet and Pride about infinite groups. *Illinois J. Math.*, 30(2):301–316, 1986. 98

[Pet06]   V. M. Petrogradsky. Examples of self-iterating Lie algebras. *J. Algebra*, 302(2):881–886, 2006. 104

[PSZ]   V. Petrogradsky, I. Shestakov, and E. Zelmanov. Nil graded self-similar algebras. Submitted. 104

[Rat04]   D. Rattaggi. *Computations in Groups Acting on a Product of Trees: Normal Subgroup Structures and Quaternion Lattices.* PhD thesis, Eidgenössische Technische Hochschule Zürich, 2004. 99, 110

[Sid97]   S. N. Sidki. A primitive ring associated to a Burnside 3-group. *J. London Math. Soc. (2)*, 55(1):55–64, 1997. 105

[Sid00]   S. N. Sidki. Automorphisms of one-rooted trees: growth, circuit structure, and acyclicity. *J. Math. Sci. (New York)*, 100(1):1925–1943, 2000. Algebra, 12. 44

[Sid05]   S. N. Sidki. Tree-wreathing applied to generation of groups by finite automata. *Internat. J. Algebra Comput.*, 15(5-6):1205–1212, 2005. 21, 70

[SS05]   P. V. Silva and B. Steinberg. On a class of automata groups generalizing lamplighter groups. *Internat. J. Algebra Comput.*, 15(5-6):1213–1234, 2005. 103

[Šun07]   Z. Šunić. Hausdorff dimension in a family of self-similar groups. *Geom. Dedicata*, 124:213–236, 2007. 94

[SVV06]   B. Steinberg, M. Vorobets, and Y. Vorobets. Automata over a binary alphabet generating free groups of even rank. arXiv:math.GR/0610033, 2006. 97

[SW03]   S. N. Sidki and J. S. Wilson. Free subgroups of branch groups. *Arch. Math. (Basel)*, 80(5):458–463, 2003. 71

[SZ08]   I. P. Shestakov and E. Zelmanov. Some examples of nil Lie algebras. *J. Eur. Math. Soc. (JEMS)*, 10(2):391–398, 2008. 104

[vN29]   J. von Neumann. Zur allgemeinen Theorie des Masses. *Fund. Math.*, 13:73–116 and 333, 1929. (= Collected works, vol. I, pages 599–643). 85

[VV06]   M. Vorobets and Y. Vorobets. On a series of finite automata defining free transformation groups. arXiv:math.GR/0604328, 2006. 97

[VV07]   M. Vorobets and Y. Vorobets. On a free group of transformations defined by an automaton. *Geom. Dedicata*, 124:237–249, 2007. 97

Index