Predicata

Deciding Presburger Arithmetic using Automata Theory

Version 1.0

July 1, 2017

Fritz Kliemann

Fritz Kliemann Email: fritz.kliemann@gmx.at

Address: Linz, Austria

Abstract

J. Shallit has successfully used automata theory to find properties of automatic sequences [Sha13]. In a summer school course at RISC, JKU Linz [AEC16], he explained also how to use finite automata to decide Presburger arithmetic [Pre29].
This package, written as a Master thesis, implements the decision procedure which goes back to J. R. Büchi [Büc60]. Furthermore, it allows to construct a deterministic finite automaton from any first-order formula with the addition as the only operation.
The package Automata [DLM11] is used for the data structure of finite automata.

Copyright

Acknowledgements

I would like to thank my supervisor Erhard Aichinger for the opportunity to write this package and my parents for their support and patience.
The work was partially supported by the Austrian Science Fund (FWF), P29931.

Contents

Chapter 1

Introduction

The possibilities of the package Predicata, a combination of the words predicate and automata, can be best described with the following example.
For which natural numbers n does the formula, describing the McNuggets numbers,

$$\exists x : \exists y : \exists z : 6 \cdot x + 9 \cdot y + 20 \cdot z = n$$

hold? Furthermore, denoting the previous formula as $P[n]$, for which natural number n does

$$(\forall m : m > n \Rightarrow P[m]) \land \neg P[n]$$

hold?
The idea is to create a deterministic finite automaton which corresponds to a first-order formula such that upon interpretation of every accepted word of the automaton the first-order formula is satisfied (Automata theory: [HMU01], [Pip97], [Koz97]).
The main object type `Predicaton` consists of an automaton and a list and represents first-order formulas containing the nullary operations 0 and 1 and the binary operation $+$. A first-order formula with n different free variables, where each free variable is assigned to pairwise distinct natural numbers, is represented by an automaton over the alphabet $\{0,1\}^n$. The variables are stored internally as a list of these n natural numbers, where the list coincides with the letters. The i-th position in a letter, i.e. in the n-tuple, corresponds to the variable at the i-th position in the list, i.e. to the variable which is assigned to the natural number at the i-th position. Leaving this technical details aside, the object type `Predicaton` (4.1.3) can also be called with a mathematically more intuitive first-order formula, which internally creates the deterministic finite automaton and takes care of the variables.
The special case are the first-order formulas with no free variable which can be seen as deterministic finite automaton with one state. This deterministic finite automatons can be either interpreted as `True` if the only state is a final state or as `False` otherwise. Thus this procedure, going back to J. R. Büchi ([Büc60]), decides the Presburger arithmetic (by Mojżesz Presburger, 1929, [Pre29]), the first-order theory of the natural numbers with the operation $+$.
For first-time users it is recommended to start with chapter 4, especially to start with the examples in section 4.2. The structure of the manual follows the structure of the package, thus the chapter 2 and 3 gives insight on how in the background a first-order formula is transferred into deterministic finite automaton. However this is quite lengthy and definitely not recommended to begin with.

Hence it's more interesting to start with the example from above:

- We start with `A:=Predicaton("(E x:(E y:(E z:6*x+9*y+20*z=n)))");`, consisting of a deterministic finite automaton with 17 states. The deterministic finite automaton displays the alphabet on the left, the states on the top, the transitions as entries in the table and the initial and final states at the bottom.

- Furthermore we can also display the `Predicaton` anytime with: `Display(A);`. Additionally, we can draw it with `DrawPredicaton(A);`, using the external program `graphviz` (for requirements refer to the manual of the package Automata).

- We can also test for accepted natural numbers with `AcceptedByPredicaton(A, 20);` where the optional second parameter gives an upper bound. `DisplayAcceptedByPredicaton(A, 99);` prints the accepted words converted into natural numbers in a nice format.

- To conclude with the example, we ask for the greatest natural number which cannot be purchased with the function `B:=GreatestNonAcceptedNumber(A);` and test for `AcceptedWordsByPredicaton(B, 50);` or sum up the regular expression `PredicatonToRatExp(B)` (note: here the binary representation is read form behind.)

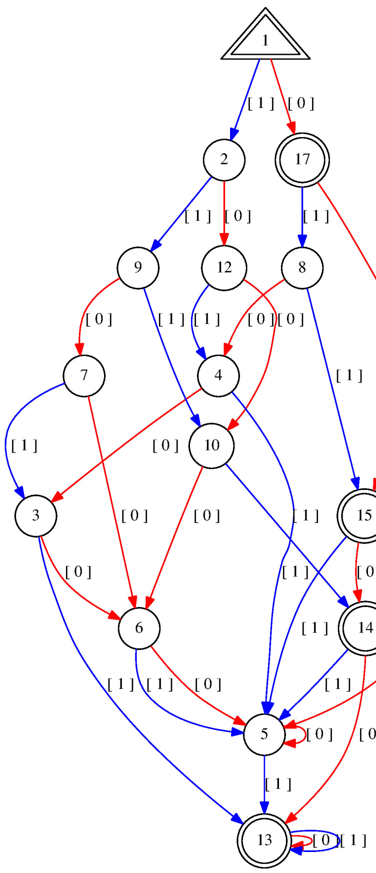

Figure 1.1: A minimal DFA recognizing the numbers which can be purchased by the formula of `A`.

Chapter 2

Creating Predicata

2.1 Predicaton – an extended finite automaton

2.1.1 Predicaton (Automaton with variable position list)

▷ `Predicaton(Automaton, VariablePositionList)` (function)

A `Predicaton` represents a first-order formulas, with n free variables, containing the nullary operations 0 and 1 and the binary operation $+$. It consists of an `Automaton` and a `VariablePositionList`. The first parameter is an *`Automaton`* from the package Automata, which is created as follows: `Automaton(Type, Size, Alphabet, TransitionTable, Initial, Final)`. In order to create a `Predicaton` the *`Type`* must either be `"det"` or `"nondet"`. The *`Size`* is a positive integer giving the number of states. The *`Alphabet`* must be a list of length 2^n, i.e. the list of all n-tuples $\{0,1\}^n$. The *`TransitionTable`* gives the transition matrix, where the entry at (i,j) denotes the state reached with the i-th letter (i-th row) and the j-th state (j-th column). The *`Initial`* and *`Final`* are the initial and final state sets.
The second parameter *`VariablePositionList`* must be of length n and must contain n pairwise distinct positive integers. It internally represents the occurring variables in the first-order formula by assigning pairwise distinct natural numbers to each free variable. The *`VariablePositionList`* coincides with the letters, i.e. the i-th position in the n-tuples correspond to the variable position at the i-th position in the list.
A word over the alphabet $\{0,1\}^n$ can be turned into n reversed binary representations of natural numbers by extracting the components of the letters. The i-th row of a word (choosing the i-th component of each letter) corresponds to the i-th entry in the *`VariablePositionList`*. The accepted words of the automaton represent those n natural numbers, such that upon interpretation the first-order formula is satisfied.
In the following example the `Automaton A` represents the formula $x+y=z$ with the following variables: the variable x is assigned to 1, the variable y is assigned to 2 and the variable z is assigned to 3 The `Predicaton P` is created with the deterministic finite automaton `A` and the variable position list `[ 1, 2, 3 ]`. This means the first entry in the letters corresponds to the variable with the assigned natural number 1, i.e. x, the second entry to the number 2, i.e. the variable y and the third entry to the number 3, i.e. the variable z.
Later also a mathematically more intuitive method is introduced, see `Predicaton` (4.1.3) for creating a `Predicaton` from a first-order formula.

Example
```
gap> A:=Automaton("det", 3,
> [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ], [ 1, 1, 0 ],
> [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ],
> [ 3, 1, 3 ], [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ],
> [ 1 ], [ 1 ]);
< deterministic automaton on 8 letters with 3 states >
gap> P:=Predicaton( A, [ 1, 2, 3 ]);
< Predicaton: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
```

2.1.2 BuildPredicaton

▷ BuildPredicaton(*Type, Size, Alphabet, TransitionTable, Initial, Final, VariablePositionList*) (function)

The function BuildPredicaton allows the creation of a Predicaton without specifying an Automaton.

Example
```
gap> P:=BuildPredicaton("det", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ],
> [ 1, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ], [ 1, 2, 3 ]);
< Predicaton: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
--------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.1.3 IsPredicaton

▷ IsPredicaton(*P*) (function)

The function IsPredicaton checks if *P* is a Predicaton.

Example
```
gap> P:=BuildPredicaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ], [ 1, 2 ]);
< Predicaton: deterministic finite automaton on 4 letters with 2 states
```

```
and the variable position list [ 1, 2 ]. >
gap> IsPredicaton(P);
true
```

2.1.4 Display (Predicaton)

▷ Display(*P*) (method)

The method Display prints the transition table of the Predicaton *P*. The left side are the letters of the alphabet, the top row are the states and the transition from the *i*-th letter (row) and *j*-th state (column) is the entry (i, j).

Example

```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ],
> [ 1, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2, 3 ]);
< Predicaton: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.1.5 View (Predicaton)

▷ View(*P*) (method)

The method View applied on a Predicaton *P* returns the object text.

Example

```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 3 ],
> [ 3, 2, 2 ] ], [ 1 ], [ 3 ]), [ 1 ]);;
gap> View(P);
< Predicaton: deterministic finite automaton on 2 letters with 3 states
and the variable position list [ 1 ]. >
```

2.1.6 Print (Predicaton)

▷ Print(*P*) (method)

The method `Print` applied on a `Predicaton` P prints the input as a string.

Example
```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]),
> [ 1, 2 ]);;
gap> Print(P);
Predicaton(Automaton("det", 3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ], [ [ \
1, 3, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;
gap> String(P);
"Predicaton(Automaton(\"det\", 3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ], [ [\
1, 3, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;"
```

2.1.7 GetAlphabet

▷ `GetAlphabet(n)` (function)

The function `GetAlphabet` returns the alphabet A^n for $A := \{0,1\}$.

Example
```
gap> a1:=GetAlphabet(3);
[ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ], [ 1, 1, 0 ],
  [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ]
gap> P1:=Predicaton(Automaton("det", 3, a1,
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2, 3 ]);;
gap> Display(P1);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
--------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> a2:=GetAlphabet(0);
[ [  ] ]
gap> P2:=Predicaton(Automaton("det", 1, a2, [ [ 1 ] ], [ 1 ], [ 1 ]), [ ]);;
gap> Display(P2);
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.2 Basic functions on Automata and Predicata

The package Automata allows lists of lists as input for the alphabet, but unfortunately is lacking in further support. The functions regarding the alphabet takes only `ShallowCopy` whereas a list of lists `StructuralCopy` is needed, as well as the method `Display` for automata prints with some weird spacing. Therefore this package reintroduces the basic Automata functions with another name to ensure full control. Nevertheless all credit belongs to the creators of the package Automata.
Note that the `Predicata` in the following examples corresponds to first-order formulas. The accepted natural numbers can be displayed with the functions from section 2.3.
Furthermore, note that the following functions can be either called with an `Automaton` or a `Predicaton`.

2.2.1 DisplayAut

▷ `DisplayAut(P)` (function)

The function `DisplayAut` prints the `Automaton` or `Predicaton` *P* (called by `Display` (2.1.4)).

Example
```
gap> A:=Automaton("det", 4, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 3, 2, 2, 4 ], [ 2, 2, 4, 2 ], [ 2, 2, 3, 2 ], [ 3, 2, 2, 4 ] ],
> [ 1 ], [ 4 ]);
< deterministic automaton on 4 letters with 4 states >
gap> DisplayAut(A);
deterministic finite automaton on 4 letters with 4 states
and the following transitions:
            |  1  2  3  4
---------------------------
  [ 0, 0 ]  |  3  2  2  4
  [ 1, 0 ]  |  2  2  4  2
  [ 0, 1 ]  |  2  2  3  2
  [ 1, 1 ]  |  3  2  2  4
 Initial states: [ 1 ]
 Final states:  [ 4 ]
```

2.2.2 DrawPredicaton

▷ `DrawPredicaton(P)` (function)

The function `DrawPredicaton` calls the function `DrawAutomaton` from the package Automata which uses `graphviz` [DEG+02], a software for drawing graphs developed at AT & T Labs, that can be obtained at `http://www.graphviz.org/`. For further details please refer to the manual of the package Automata.

Example
```
gap> A:=Automaton("det", 4, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 3, 2, 2, 4 ], [ 2, 2, 4, 2 ], [ 2, 2, 3, 2 ], [ 3, 2, 2, 4 ] ],
> [ 1 ], [ 4 ]);
< deterministic automaton on 4 letters with 4 states >
gap> DisplayAut(A);
deterministic finite automaton on 4 letters with 4 states
and the following transitions:
```

```
             |  1  2  3  4
---------------------------
  [ 0, 0 ]   |  3  2  2  4
  [ 1, 0 ]   |  2  2  4  2
  [ 0, 1 ]   |  2  2  3  2
  [ 1, 1 ]   |  3  2  2  4
 Initial states: [ 1 ]
 Final states:  [ 4 ]
```

2.2.3 IsDeterministicAut

▷ IsDeterministicAut(*P*) (function)

The function `IsDeterministicAut` checks if the `Type` of an `Automaton` or a `Predicaton` *P* is `"det"`. If yes then `true`, otherwise `false`.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ], [ [ 1, 2, 2, 3, 2 ],
> [ 2, 2, 1, 2, 4 ] ], [ 5 ], [ 1 ]), [ 1 ]);
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> IsDeterministicAut(P);
true
```

2.2.4 IsNonDeterministicAut

▷ IsNonDeterministicAut(*P*) (function)

The function `IsNonDeterministicAut` checks if the `Type` of an `Automaton` or a `Predicaton` *P* is `"nondet"`. If yes then `true`, otherwise `false`.

Example

```
gap> P:=Predicaton(Automaton("nondet", 2, [ [ 0 ], [ 1 ] ], [ [ 1  ], [  ] ],
> [ 1 ], [ 1 ]), [ 1 ]);
< Predicaton: nondeterministic finite automaton on 2 letters with 2 states
and the variable position list [ 1 ]. >
gap> Display(P);
Predicaton: nondeterministic finite automaton on 2 letters with 2 states,
the variable position list [ 1 ] and the following transitions:
         |  1      2
--------------------------
  [ 0 ]  |  [ 1 ]  [ ]
  [ 1 ]  |  [ ]    [ ]
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> IsNonDeterministicAut(P);
true
```

2.2.5 TypeOfAut

▷ TypeOfAut(*P*) (function)

The function TypeOfAut returns the Type of an Automaton or a Predicaton *P*, either "det", "nondet" or "epsilon". Note that a Predicaton can only be a deterministic or nondeterministic finite automaton.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 2, 2, 2, 5 ], [ 2, 2, 5, 2, 2 ], [ 2, 2, 2, 3, 2 ], [ 4, 2, 2, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> TypeOfAut(P);
"det"
```

2.2.6 AlphabetOfAut

▷ AlphabetOfAut(*P*) (function)

The function AlphabetOfAut returns the size of an Alphabet of an Automaton or a Predicaton *P*.

Example
```
gap> P:=Predicaton(Automaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;
gap> AlphabetOfAut(P);
4
```

2.2.7 AlphabetOfAutAsList

▷ AlphabetOfAutAsList(*P*) (function)

The function AlphabetOfAutAsList returns a StructuralCopy of the Alphabet of an Automaton or a Predicaton *P*.

Example
```
gap> # Continued
gap> AlphabetOfAutAsList(P);
[ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ]
```

2.2.8 NumberStatesOfAut

▷ NumberStatesOfAut(*P*) (function)

The function NumberStatesOfAut returns the number of the States of an Automaton or a Predicaton *P*.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 2, 2, 2, 5 ], [ 4, 2, 5, 3, 2 ], [ 4, 2, 5, 3, 2 ], [ 2, 2, 2, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> NumberStatesOfAut(P);
5
```

2.2.9 SortedStatesAut

▷ SortedStatesAut(*P*) (function)

The function SortedStatesAut returns the Automaton or the Predicaton *P* with sorted States, such that the initial states have the lowest and the final states the highest number.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ], [ [ 1, 2, 2, 2, 2 ],
> [ 2, 2, 1, 3, 4 ] ], [ 5 ], [ 1 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  1  2  2  2  2
  [ 1 ]  |  2  2  1  3  4
 Initial states: [ 5 ]
 Final states:  [ 1 ]
gap> S:=SortedStatesAut(P);;
gap> Display(S);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  2  2  2  2  5
  [ 1 ]  |  4  2  5  3  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
```

2.2.10 TransitionMatrixOfAut

▷ TransitionMatrixOfAut(*P*) (function)

The function TransitionMatrixOfAut returns a StructuralCopy of the TransitionMatrix of an Automaton or a Predicaton *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ], [ [ 1, 2, 2, 2, 2 ],
> [ 2, 2, 1, 3, 4 ] ], [ 5 ], [ 1 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  1  2  2  2  2
  [ 1 ]  |  2  2  1  3  4
 Initial states: [ 5 ]
 Final states:  [ 1 ]
gap> TransitionMatrixOfAut(P);
[ [ 1, 2, 2, 2, 2 ], [ 2, 2, 1, 3, 4 ] ]
```

2.2.11 InitialStatesOfAut

▷ InitialStatesOfAut(*P*) (function)

The function `InitialStatesOfAut` returns the `Initial` states of an `Automaton` or a `Predicaton` *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 2, 3, 5 ],
> [ 4, 2, 5, 2, 2 ] ], [ 1 ], [ 5 ]), [ 1 ]);;
gap> InitialStatesOfAut(P);
[ 1 ]
```

2.2.12 SetInitialStatesOfAut

▷ `SetInitialStatesOfAut(P)` (function)

The function `SetInitialStatesOfAut` sets the `Initial` states of an `Automaton` or a `Predicaton` *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 2, 3, 5 ],
> [ 4, 2, 5, 2, 2 ] ], [ 1 ], [ 5 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  2  2  2  3  5
  [ 1 ]  |  4  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> SetInitialStatesOfAut(P, 3);
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  2  2  2  3  5
  [ 1 ]  |  4  2  5  2  2
 Initial states: [ 3 ]
 Final states:  [ 5 ]
```

2.2.13 FinalStatesOfAut

▷ `FinalStatesOfAut(P)` (function)

The function `FinalStatesOfAut` returns the `Final` states of an `Automaton` or a `Predicaton` *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 2, 4 ],
> [ 3, 2, 4, 2 ] ], [ 1 ], [ 4 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  2  2  2  4
```

```
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 4 ]
gap> FinalStatesOfAut(P);
[ 4 ]
```

2.2.14 SetFinalStatesOfAut

▷ SetFinalStatesOfAut(*P*) (function)

The function SetFinalStatesOfAut sets the Final states of an Automaton or a Predicaton *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 2, 4 ],
> [ 3, 2, 4, 2 ] ], [ 1 ], [ 4 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  2  2  2  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 4 ]
gap> SetFinalStatesOfAut(P, [ 1, 2, 3 ]);
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  2  2  2  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 1, 2, 3 ]
```

2.2.15 SinkStatesOfAut

▷ SinkStatesOfAut(*P*) (function)

The function SinkStatesOfAut returns the sink states of an Automaton or a Predicaton *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 3 ],
> [ 3, 2, 2 ] ], [ 1 ], [ 3 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  2  2  3
  [ 1 ]  |  3  2  2
 Initial states: [ 1 ]
 Final states:  [ 3 ]
```

```
gap> SinkStatesOfAut(P);
[ 2 ]
```

2.2.16 PermutedStatesAut

▷ PermutedStatesAut(*P*, *p*) (function)

The function `PermutedStatesAut` permutes the names of the states of an `Automaton` or a `Predicaton` *P*. The list *p* contains all states, where the state `i` (i.e. `i`-th position) is mapped to the state `p[i]`.

Example
```
gap> P:=Predicaton(Automaton("det", 6, [ [ 0 ], [ 1 ] ], [ [ 5, 2, 2, 3, 4, 6 ],
> [ 2, 2, 6, 2, 2, 2 ] ], [ 1 ], [ 6 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  5  2  2  3  4  6
  [ 1 ]  |  2  2  6  2  2  2
 Initial states: [ 1 ]
 Final states:   [ 6 ]
gap> Q:=PermutedStatesAut(P,[1,6,4,3,2,5]);;
gap> Display(Q);
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  2  3  4  6  5  6
  [ 1 ]  |  6  6  6  5  6  6
 Initial states: [ 1 ]
 Final states:   [ 5 ]
```

2.2.17 CopyAut

▷ CopyAut(*P*) (function)

▷ CopyPredicaton(*P*) (function)

The function `CopyAut` copies either the `Automaton` or the `Predicaton` *P*.

Example
```
gap> P:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 1 ]), [ 1 ]);;
gap> C:=CopyAut(P);;
gap> SetFinalStatesOfAut(C, 2);
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 2 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2
------------------
  [ 0 ]  |  1  2
```

```
  [ 1 ]  |  2  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> Display(C);
Predicaton: deterministic finite automaton on 2 letters with 2 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2
------------------
  [ 0 ]  |  1  2
  [ 1 ]  |  2  2
 Initial states: [ 1 ]
 Final states:  [ 2 ]
```

2.2.18 MinimalAut

▷ `MinimalAut(P)` (function)

The function `MinimalAut` returns the minimal deterministic finite automaton of an `Automaton` *P*. Given a `Predicaton` *P* its automaton is minimized and returned as a `Predicaton` with the same variable position list.

Example

```
gap> P:=Predicaton(Automaton("det", 9, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 6, 7, 4, 5, 4, 5, 8, 9 ], [ 3, 6, 6, 4, 4, 4, 4, 8, 8 ],
> [ 4, 4, 5, 4, 5, 8, 9, 4, 5 ], [ 5, 4, 4, 4, 4, 8, 8, 4, 4 ] ],
> [ 1 ], [ 9 ]), [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 9 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5  6  7  8  9
------------------------------------------
  [ 0, 0 ]  |  2  6  7  4  5  4  5  8  9
  [ 1, 0 ]  |  3  6  6  4  4  4  4  8  8
  [ 0, 1 ]  |  4  4  5  4  5  8  9  4  5
  [ 1, 1 ]  |  5  4  4  4  4  8  8  4  4
 Initial states: [ 1 ]
 Final states:  [ 9 ]
gap> M:=MinimalAut(P);;
gap> Display(M);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  1  2  2  3  2
  [ 1, 0 ]  |  2  2  2  2  4
  [ 0, 1 ]  |  2  2  1  2  2
  [ 1, 1 ]  |  2  2  2  2  2
 Initial states: [ 5 ]
 Final states:  [ 1 ]
gap> P:=Predicaton(Automaton("nondet", 8, [ [ 0 ], [ 1 ] ],
> [ [ [ 2 ], [ 2 ], [ 2 ], [ 4 ], [ 7 ], [ 6 ], [ 6 ], [ 8 ] ],
> [ [ 3 ], [ 2 ], [ 4 ], [ 2 ], [ 6 ], [ 6 ], [ 8 ], [ 6 ] ] ],
> [ 1, 5 ], [ 4, 8 ]), [ 1 ]);;
```

```
gap> M:=MinimalAut(P);;
gap> Display(M);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  1  2  2  3
  [ 1 ]  |  2  2  1  3
 Initial states: [ 4 ]
 Final states:  [ 1 ]
```

2.2.19 NegatedAut

▷ `NegatedAut(P)` (function)

The function `NegatedAut` changes the `Final` states to non-final ones and the non-final states to `Final` ones.

Example

```
gap> P:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 2, 4 ],
> [ 3, 2, 4, 2 ] ], [ 1 ], [ 4 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  2  2  2  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 4 ]
gap> Q:=NegatedAut(P);;
gap> Display(Q);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  2  2  2  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 1, 2, 3 ]
```

2.2.20 IntersectionAut

▷ `IntersectionAut(P)` (function)

The function `IntersectionAut` returns the intersection of two `Automata` or `Predicata` *P*. Note that the for intersection of two automata both must have the same ordered alphabet. For the intersection of two `Predicata` with different alphabets use `IntersectionPredicata` (2.3.19).

Example

```
gap> P1:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 2, 2, 3, 5 ], [ 2, 2, 2, 3, 5 ], [ 4, 2, 5, 2, 2 ], [ 4, 2, 5, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
```

```
gap> P2:=Predicaton(Automaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;
gap> P3:=IntersectionAut(P1, P2);;
gap> Display(P3);
Predicaton: deterministic finite automaton on 4 letters with 9 states,
the variable position list [ 1, 2 ] and the following transitions:
             |  1  2  3  4  5  6  7  8  9
------------------------------------------
  [ 0, 0 ]   |  2  2  3  6  7  3  2  8  9
  [ 1, 0 ]   |  3  3  3  6  6  3  3  8  8
  [ 0, 1 ]   |  4  3  3  3  3  8  8  3  3
  [ 1, 1 ]   |  5  2  3  3  2  8  9  3  2
 Initial states: [ 1 ]
 Final states:  [ 9 ]
gap> P4:=MinimalAut(P3);;
gap> Display(P4);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
             |  1  2  3  4  5
------------------------------
  [ 0, 0 ]   |  1  2  2  3  2
  [ 1, 0 ]   |  2  2  2  2  2
  [ 0, 1 ]   |  2  2  2  2  2
  [ 1, 1 ]   |  2  2  1  2  4
 Initial states: [ 5 ]
 Final states:  [ 1 ]
```

2.2.21 UnionAut

▷ UnionAut(*P*) (function)

The function UnionAut returns the union of two Automata or Predicata *P*. Note that for the union of two automata both must have the same ordered alphabet. For the union of two Predicata with different alphabets use UnionPredicata (2.3.20).

Example

```
gap> P1:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ] ], [ 1 ], [ 1 ]), [ 1 ]);;
gap> P2:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ],
> [ [ 3, 2, 2, 4 ], [ 2, 2, 4, 2 ] ], [ 1 ], [ 4 ]), [ 1 ]);;
gap> P3:=UnionAut(P1, P2);;
gap> Display(P3);
Predicaton: nondeterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
          |  1      2      3      4      5      6
------------------------------------------------------
  [ 0 ]   |  [ 1 ]  [ 2 ]  [ 5 ]  [ 4 ]  [ 4 ]  [ 6 ]
  [ 1 ]   |  [ 2 ]  [ 2 ]  [ 4 ]  [ 4 ]  [ 6 ]  [ 4 ]
 Initial states: [ 1, 3 ]
 Final states:  [ 1, 6 ]
gap> M:=MinimalAut(P3);;
gap> Display(M);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
```

```
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  1  2  2  3
  [ 1 ]  |  1  1  2  1
 Initial states: [ 4 ]
 Final states:  [ 2, 3, 4 ]
```

2.2.22 IsRecognizedByAut

▷ `IsRecognizedByAut(P, word)` (function)

The function `IsRecognizedByAut` checks if a *word*, given by its letters, is accepted by the `Automaton` or `Predicaton` *P*.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ],
> [ [ 5, 5, 5, 4, 5 ], [ 2, 3, 4, 5, 5 ] ], [ 1 ], [ 4 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  5  5  5  4  5
  [ 1 ]  |  2  3  4  5  5
 Initial states: [ 1 ]
 Final states:  [ 4 ]
gap> IsRecognizedByAut(P,[[1],[1],[1]]);
true
gap> IsRecognizedByAut(P,[[1],[1],[1],[0],[0]]);
true
gap> IsRecognizedByAut(P,[[1],[1],[0]]);
false
```

2.3 Basic functions on Predicata

The following functions act only on Predicata, accessing and modifying the alphabet $A := \{0,1\}^n$ for a natural number n (including 0).

2.3.1 DecToBin

▷ `DecToBin(D)` (function)

The function `DecToBin` returns for a natural numbers *D* or the list of its binary representation. Note that here, motivated on how the automata read the words, the binary representation are read in the other direction than usual, for example $4 = [0,0,1]_2$.

Example
```
gap> DecToBin(4);
[ 0, 0, 1 ]
gap> DecToBin(0);
[ 0 ]
```

2.3.2 BinToDec

▷ `BinToDec(B)` (function)

The function `BinToDec` returns for a list B (i.e. a binary representation), containing `0`s and `1`s, the corresponding natural number. Note again that here the $\sum b_{i+1} * 2^i$ starting at $i = 0$ is evaluated the other way around than it's usually done.

Example

```
gap> BinToDec([ 0, 0, 1 ]);
4
gap> BinToDec([ 0, 0, 1, 0, 0, 0, 0 ]);
4
gap> BinToDec([ ]);
0
```

2.3.3 IsAcceptedWordByPredicaton

▷ `IsAcceptedWordByPredicaton(P, L)` (function)
▷ `IsAcceptedByPredicaton(P, L)` (function)

The function `IsAcceptedWordByPredicaton` checks if a list of natural numbers L or a list of binary representation L is accepted by the `Predicaton` P. Compare with `IsRecognizedByAut` (2.2.22), which uses the letters instead of the words.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 2, 2, 2, 5 ], [ 4, 2, 2, 3, 2 ], [ 2, 2, 2, 2, 2 ], [ 2, 2, 5, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  2  2  2  2  5
  [ 1, 0 ]  |  4  2  2  3  2
  [ 0, 1 ]  |  2  2  2  2  2
  [ 1, 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> IsAcceptedWordByPredicaton(P, [ 7, 4 ]);
true
gap> IsAcceptedWordByPredicaton(P, [ DecToBin(7), DecToBin(4) ]);
true
gap> IsAcceptedWordByPredicaton(P, [ [ 1, 1, 1, 0 ], [ 0, 0, 1, 0, 0, 0 ] ]);
true
gap> IsRecognizedByAut(P, [ [ 1, 0 ], [ 1, 0 ], [ 1, 1 ] ]); # 1st row = 7, 2nd row = 4
true
```

2.3.4 AcceptedWordsByPredicaton

▷ AcceptedWordsByPredicaton(*P*[, *b*]) (function)
▷ AcceptedByPredicaton(*P*[, *b*]) (function)

The function AcceptedWordsByPredicaton returns the accepted words of the Predicaton *P* up to an upper bound *b* (on default b=10), either a positive integer or a list with positive integers as an individual bound for each variable. Alternatively, list of lists where each list contains the to be tested values is also allowed.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 2, 2, 2, 5 ], [ 4, 2, 2, 3, 2 ], [ 2, 2, 2, 2, 2 ], [ 2, 2, 5, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> AcceptedWordsByPredicaton(P, [ 10, 20 ]);
[ [ 7, 4 ] ]
gap> P:=Predicaton(Automaton("det", 3, [ [ 0 ], [ 1 ] ],
> [ [ 1, 3, 2 ], [ 2, 1, 3 ] ], [ 1 ], [ 1 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
----------------------
  [ 0 ]  |  1  3  2
  [ 1 ]  |  2  1  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> AcceptedWordsByPredicaton(P, 29);
[ [ 0 ], [ 3 ], [ 6 ], [ 9 ], [ 12 ], [ 15 ], [ 18 ], [ 21 ], [ 24 ], [ 27 ] ]
gap> AcceptedWordsByPredicaton(P, [ [121..144] ]);
[ [ 123 ], [ 126 ], [ 129 ], [ 132 ], [ 135 ], [ 138 ], [ 141 ], [ 144 ] ]
```

2.3.5 DisplayAcceptedWordsByPredicaton

▷ DisplayAcceptedWordsByPredicaton(*P*[, *b*, *t*]) (function)
▷ DisplayAcceptedByPredicaton(*P*[, *b*, *t*]) (function)

The function DisplayAcceptedWordsByPredicaton prints the accepted words of the Predicaton *P* in a nice way. For one variable as a "list" with YES/no, for two variables as a "matrix" containing YES/no and for three variables as a "matrix", which entries are the third accepted natural numbers. The optional parameter *b* gives an upper bound for the displayed natural numbers, where either a positive integer or a list of positive integers denotes the maximal natural numbers which are asked for. The second optional parameter, if true allows to reduce YES/no to Y/n for the case of one variable.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 2, 2, 2, 3, 5 ], [ 4, 2, 2, 3, 2 ], [ 2, 2, 2, 3, 2 ], [ 2, 2, 5, 3, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> AcceptedWordsByPredicaton(P);
[ [ 5, 4 ], [ 5, 6 ], [ 7, 4 ], [ 7, 6 ] ]
gap> DisplayAcceptedWordsByPredicaton(P, [8,10]);
 If the following words are accepted by the given automaton, then: YES,
 otherwise if not accepted: no.
```

```
    | 0   1   2   3   4   5   6   7   8   9   10
 ------------------------------------------------
   0 | no  no  no  no  no  no  no  no  no  no  no
   1 | no  no  no  no  no  no  no  no  no  no  no
   2 | no  no  no  no  no  no  no  no  no  no  no
   3 | no  no  no  no  no  no  no  no  no  no  no
   4 | no  no  no  no  no  no  no  no  no  no  no
   5 | no  no  no  no  YES no  YES no  no  no  no
   6 | no  no  no  no  no  no  no  no  no  no  no
   7 | no  no  no  no  YES no  YES no  no  no  no
   8 | no  no  no  no  no  no  no  no  no  no  no

gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ],
> [ [ 3, 2, 5, 4, 4 ], [ 3, 2, 4, 2, 4 ] ],
> [ 1 ], [ 3, 4, 5, 1 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  3  2  5  4  4
  [ 1 ]  |  3  2  4  2  4
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4, 5 ]
gap> AcceptedWordsByPredicaton(P, 19);
[ [ 0 ], [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
gap> DisplayAcceptedWordsByPredicaton(P, 29, true);
 If the following words are accepted by the given automaton, then: Y,
 otherwise if not accepted: n.
   0: Y    1: Y    2: Y    3: Y    4: Y    5: Y    6: n    7: n    8: n    9: n
  10: n   11: n   12: n   13: n   14: n   15: n   16: n   17: n   18: n   19: n
  20: n   21: n   22: n   23: n   24: n   25: n   26: n   27: n   28: n   29: n
```

2.3.6 DisplayAcceptedWordsByPredicatonInNxN

▷ DisplayAcceptedWordsByPredicatonInNxN(*P*[, *b*]) (function)

▷ DisplayAcceptedByPredicatonInNxN(*P*[, *b*]) (function)

The function DisplayAcceptedWordsByPredicatonInNxN prints the accepted words of the Predicaton *P* with a variable position list of length two in a fancy way in $\mathbb{N} \times \mathbb{N}$. It "draws" the natural number solutions of linear equations, which can be seen, due to the linearity, as "lines". The optional parameter *l* gives an upper bound for the displayed accepted words, it must be a list containing two positive integers.

Example

```
gap> P:=Predicaton(Automaton("det", 14, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 6, 2, 2, 3, 4, 2, 2, 3, 7, 2, 10, 12, 12, 14 ],
> [ 2, 2, 12, 2, 9, 11, 7, 7, 2, 13, 2, 2, 7, 2 ],
> [ 2, 2, 12, 2, 7, 8, 14, 14, 2, 14, 2, 2, 14, 2 ],
> [ 5, 2, 2, 12, 3, 2, 2, 12, 7, 2, 13, 2, 2, 14 ] ],
```

```
> [ 1 ], [ 12, 13, 14 ]), [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 14 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5  6  7  8  9  10 11 12 13 14
---------------------------------------------------------
  [ 0, 0 ]  |  6  2  2  3  4  2  2  3  7  2  10 12 12 14
  [ 1, 0 ]  |  2  2  12 2  9  11 7  7  2  13 2  2  7  2
  [ 0, 1 ]  |  2  2  12 2  7  8  14 14 2  14 2  2  14 2
  [ 1, 1 ]  |  5  2  2  12 3  2  2  12 7  2  13 2  2  14
 Initial states: [ 1 ]
 Final states:  [ 12, 13, 14 ]
gap> DisplayAcceptedWordsByPredicatonInNxN(P, [ 15, 15 ]);
  15 -                                      o
     |
  14 -                                  o
     |
  13 -                              o
     |
  12 -                          o
     |
  11 -                      o
     |
  10 -  o               o
     |
   9 -      o       o
     |
   8 -          o
     |
   7 -      o       o
     |
   6 -  o               o
     |
   5 -                      o
     |
   4 -                          o
     |
   3 -                              o
     |
   2 -                                  o
     |
   1 -                                      o
     |
   0 -                                          o
     |
   --+--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---->
     |  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

2.3.7 AutomatonOfPredicaton

▷ AutomatonOfPredicaton(*P*) (function)

▷ AutOfPredicaton(*P*) (function)

The function `AutomatonOfPredicaton` returns the `Automaton` of a `Predicaton` *P*.

Example
```
gap> P:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ],
> [ [ 4, 2, 3, 3 ], [ 3, 2, 2, 3 ] ], [ 1 ], [ 3, 4, 1 ]), [ 1 ]);
< Predicaton: deterministic finite automaton on 2 letters with 4 states
and the variable position list [ 1 ]. >
gap> AutomatonOfPredicaton(P);
< deterministic automaton on 2 letters with 4 states >
```

2.3.8 VariablePositionListOfPredicaton

▷ `VariablePositionListOfPredicaton(P)` (function)
▷ `VarPosListOfPredicaton(P)` (function)

The function `VariablePositionListOfPredicaton` returns the variable position list of a `Predicaton` *P*.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2, 2, 3, 2 ], [ 4, 2, 2, 5, 2 ], [ 2, 2, 1, 2, 3 ], [ 2, 2, 4, 2, 5 ] ],
> [ 1 ], [ 1 ]), [ 4, 9 ]);;
gap> VariablePositionListOfPredicaton(P);
[ 4, 9 ]
```

2.3.9 SetVariablePositionListOfPredicaton

▷ `SetVariablePositionListOfPredicaton(P, l)` (function)
▷ `SetVarPosListOfPredicaton(P, l)` (function)

The function `SetVariablePositionListOfPredicaton` sets the variable position list of a `Predicaton` *P*, permuting the alphabet if necessary, see `PermutedAlphabetPredicaton` (2.3.21).

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2, 2, 3, 2 ], [ 4, 2, 2, 5, 2 ], [ 2, 2, 1, 2, 3 ], [ 2, 2, 4, 2, 5 ] ],
> [ 1 ], [ 1 ]), [ 4, 9 ]);;
gap> SetVariablePositionListOfPredicaton(P, [ 1, 2 ]);
gap> VariablePositionListOfPredicaton(P);
[ 1, 2 ]
```

2.3.10 ProductLZeroPredicaton

▷ `ProductLZeroPredicaton(P)` (function)

The function `ProductLZeroPredicaton` takes the `Predicaton` *P* and adds a new state. This new state is final and is reached through `[0,...,0]` from all `Final` states. Hence the returned `Predicaton` recognizes the product of the languages of the given `Predicaton` and the language containing all the zero words.

Example

```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ], [ [ 3, 2, 4, 2, 2 ],
> [ 2, 2, 2, 5, 2 ] ], [ 1 ], [ 5 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  3  2  4  2  2
  [ 1 ]  |  2  2  2  5  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 0, 1 ] ]);
true
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 0, 1, 0 ] ]);
false
gap> PredicatonToRatExp(P);
[ 0 ][ 0 ][ 1 ]
gap> Q:=ProductLZeroPredicaton(P);;
gap> Display(Q);
Predicaton: nondeterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1         2         3         4         5         6
-------------------------------------------------------------------------
  [ 0 ]  |  [ 3 ]     [ 2 ]     [ 4 ]     [ 2 ]     [ 2, 6 ]  [ 6 ]
  [ 1 ]  |  [ 2 ]     [ 2 ]     [ 2 ]     [ 5 ]     [ 2 ]     [ ]
 Initial states: [ 1 ]
 Final states:  [ 5, 6 ]
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 0, 1 ] ]);
true
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 0, 1, 0 ] ]);
true
gap> PredicatonToRatExp(Q);
[ 0 ][ 0 ][ 1 ]([ 0 ][ 0 ]*U@)
gap> M:=MinimalAut(Q);;
gap> M:=PermutedStatesAut(M, [ 5, 2, 4, 3, 1 ]);;
gap> Display(M);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  3  2  4  2  5
  [ 1 ]  |  2  2  2  5  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> PredicatonToRatExp(M);
[ 0 ][ 0 ][ 1 ][ 0 ]*
```

2.3.11 RightQuotientLZeroPredicaton

▷ `RightQuotientLZeroPredicaton(P)` (function)

The function `RightQuotientLZeroPredicaton` takes the `Predicaton` P and runs through all final states. If a `Final` state is reached with `[0,...,0]` then this state is added to the final states. Hence the returned `Predicaton` recognizes the right quotient of the language of the given `Predicaton` with the language containing only the zero words.

Example

```
gap> P:=Predicaton(Automaton("det", 6, [ [ 0 ], [ 1 ] ], [ [ 3, 2, 4, 2, 6, 2 ],
> [ 2, 2, 2, 5, 2, 2 ] ], [ 1 ], [ 6 ]), [ 1 ]);;
gap> Display(P);
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  3  2  4  2  6  2
  [ 1 ]  |  2  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 6 ]
gap> IsAcceptedWordByPredicaton(P, [ 4 ]);
false
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 0, 1 ] ]);
false
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 0, 1, 0 ] ]);
true
gap> PredicatonToRatExp(P);
[ 0 ][ 0 ][ 1 ][ 0 ]
gap> Q:=RightQuotientLZeroPredicaton(P);;
gap> Display(Q);
Predicaton: nondeterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1      2      3      4      5      6
------------------------------------------------------
  [ 0 ]  |  [ 3 ]  [ 2 ]  [ 4 ]  [ 2 ]  [ 6 ]  [ 2 ]
  [ 1 ]  |  [ 2 ]  [ 2 ]  [ 2 ]  [ 5 ]  [ 2 ]  [ 2 ]
 Initial states: [ 1 ]
 Final states:  [ 5, 6 ]
gap> IsAcceptedWordByPredicaton(Q, [ 4 ]);
true
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 0, 1 ] ]);
true
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 0, 1, 0 ] ]);
true
gap> PredicatonToRatExp(Q);
[ 0 ][ 0 ][ 1 ]([ 0 ]U@)
gap> M:=MinimalAut(Q);;
gap> M:=PermutedStatesAut(M, [ 6, 2, 5, 4, 3, 1 ]);;
gap> Display(M);
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  3  2  4  2  6  2
  [ 1 ]  |  2  2  2  5  2  2
 Initial states: [ 1 ]
```

```
 Final states:  [ 5, 6 ]
gap> IsAcceptedWordByPredicaton(M, [ 4 ]);
true
gap> PredicatonToRatExp(M);
[ 0 ][ 0 ][ 1 ]([ 0 ]U@)
```

2.3.12 NormalizedLeadingZeroPredicaton

▷ `NormalizedLeadingZeroPredicaton(P)` (function)

The function `NormalizedLeadingZeroPredicaton` returns the union of `ProductLZeroPredicaton` (2.3.10) and `RightQuotientLZeroPredicaton` (2.3.11) of the given `Predicaton` *P*. Therefore the returned `Predicaton` accepts any previously accepted words with cancelled or added leading zeros.

Example

```
gap> P:=Predicaton(Automaton("det", 7, [ [ 0 ], [ 1 ] ], [ [ 3, 2, 4, 2, 6, 2, 2],
> [ 2, 2, 7, 5, 2, 2, 2 ] ], [ 1 ], [ 6, 7 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 7 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7
---------------------------------
  [ 0 ]  |  3  2  4  2  6  2  2
  [ 1 ]  |  2  2  7  5  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 6, 7 ]
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 1 ] ]);
true
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 1, 0 ] ]);
false
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 0, 1 ] ]);
false
gap> IsAcceptedWordByPredicaton(P, [ [ 0, 0, 1, 0 ] ]);
true
gap> PredicatonToRatExp(P);
[ 0 ]([ 1 ]U[ 0 ][ 1 ][ 0 ])
gap> Q:=NormalizedLeadingZeroPredicaton(P);;
gap> Display(Q);
Predicaton: nondeterministic finite automaton on 2 letters with 16 states,
the variable position list [ 1 ] and the following transitions:
         | 1       2       3       4       5       6       7       8
---------------------------------------------------------------------------
  [ 0 ]  |  [ 2 ]   [ 4 ]   [ 3 ]   [ 3 ]   [ 7 ]   [ 8 ]   [ 9 ]   [ 7 ]
  [ 1 ]  |  [ 3 ]   [ 5 ]   [ 3 ]   [ 6 ]   [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]
 ...
         | 9       10      11      12      13      14      15      16
---------------------------------------------------------------------------
  [ 0 ]  |  [ 9 ]   [ 11 ]  [ 13 ]  [ 12 ]  [ 12 ]  [ 12 ]  [ 16 ]  [ 12 ]
  [ 1 ]  |  [ 3 ]   [ 12 ]  [ 14 ]  [ 12 ]  [ 15 ]  [ 12 ]  [ 12 ]  [ 12 ]
 Initial states: [ 1, 10 ]
 Final states:  [ 5, 7, 8, 9, 14, 15, 16 ]
gap> AcceptedWordsByPredicaton(Q, 10);
```

```
[ [ 2 ], [ 4 ] ]
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 1 ] ]);
true
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 1, 0 ] ]);
true
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 0, 1 ] ]);
true
gap> IsAcceptedWordByPredicaton(Q, [ [ 0, 0, 1, 0 ] ]);
true
gap> M:=MinimalAut(Q);;
gap> M:=PermutedStatesAut(M, [ 3, 5, 1, 4, 2 ]);;
gap> Display(M);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  3  2  4  2  5
  [ 1 ]  |  2  2  5  5  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> AcceptedWordsByPredicaton(M, 10);
[ [ 2 ], [ 4 ] ]
gap> PredicatonToRatExp(M);
[ 0 ]([ 0 ][ 1 ]U[ 1 ])[ 0 ]*
```

2.3.13 SortedAlphabetPredicaton

▷ SortedAlphabetPredicaton(*P*) (function)

▷ SortedAbcPredicaton(*P*) (function)

The function SortedAlphabetPredicaton returns the Predicaton *P* with the component-wise sorted Alphabet (from right to left with $0 < 1$).

Example

```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ],
> [ 1, 0, 1 ], [ 0, 1, 0 ], [ 0, 1, 1 ], [ 1, 1, 0 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2, 3 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
 ---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 0, 0, 1 ]  |  3  2  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 1, 0, 1 ]  |  2  3  3
  [ 0, 1, 0 ]  |  3  1  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 0 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

```
gap> Q:=SortedAlphabetPredicaton(P);;
gap> Display(Q);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
 ---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  1  3
  [ 1, 1, 0 ]  |  1  3  3
  [ 0, 0, 1 ]  |  3  2  3
  [ 1, 0, 1 ]  |  2  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.3.14 FormattedPredicaton

▷ FormattedPredicaton(*P*) (function)

The function computes first the NormalizedLeadingZeroPredicaton (2.3.12) and then the MinimalAut (2.2.18) of the Predicaton *P*.

Example

```
gap> P:=Predicaton(Automaton("det", 7, [ [ 0 ], [ 1 ] ], [ [ 3, 2, 4, 2, 6, 2, 2],
> [ 2, 2, 7, 5, 2, 2, 2 ] ], [ 1 ], [ 6, 7 ]), [ 1 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 7 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7
---------------------------------
  [ 0 ]  |  3  2  4  2  6  2  2
  [ 1 ]  |  2  2  7  5  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 6, 7 ]
gap> PredicatonToRatExp(P);
[ 0 ]([ 1 ]U[ 0 ][ 1 ][ 0 ])
gap> M:=FormattedPredicaton(P);;
gap> Display(M);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  2  1  5  5
  [ 1 ]  |  2  5  5  2  5
 Initial states: [ 3 ]
 Final states:  [ 2 ]
gap> PredicatonToRatExp(M);
[ 0 ]([ 0 ][ 1 ]U[ 1 ])[ 0 ]*
```

2.3.15 IsValidInput

▷ IsValidInput(*P*, *n*) (function)

The function IsValidInput checks if the list *n* contains positive integers and if it is a valid variable position list of the given Predicaton *P*, i.e. variable position list is a subset of *n*.

Example
```
gap> P:=Predicaton(Automaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 1, 2 ], [ 2, 2 ] ], [ 1 ], [ 1 ]), [ 2, 4 ]);;
gap> IsValidInput(P, [ 1, 2, 3 ]);
The new variable position list must contain the old one of the Predicaton.
Compare [ 2, 4 ] with [ 1, 2, 3 ].
false
gap> IsValidInput(P, [ 1, 2, 3, 4 ]);
true
```

2.3.16 ExpandedPredicaton

▷ ExpandedPredicaton(*P*, *n*) (function)

The function ExpandedPredicaton returns the Predicaton *P* with the new variable position list *n*. For each new variable position in *n*, the alphabet size doubles. In each step 0s and 1s are added at the correct position in all letters of the alphabet, whereas the transition matrix rows are copied accordingly. Formally this corresponds to the preimage of the homomorphism ignoring a component of the letters applied to the deterministic finite automaton.

Example
```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0 ], [ 1 ] ], [ [ 2, 2, 3 ],
> [ 3, 2, 2 ] ], [ 1 ], [ 3 ]), [ 1 ]);
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  2  2  3
  [ 1 ]  |  3  2  2
 Initial states: [ 1 ]
 Final states:  [ 3 ]
gap> Q:=ExpandedPredicaton(P, [ 1, 2, 3 ]);;
gap> Display(Q);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  2  2  3
  [ 1, 0, 0 ]  |  3  2  2
  [ 0, 1, 0 ]  |  2  2  3
  [ 1, 1, 0 ]  |  3  2  2
  [ 0, 0, 1 ]  |  2  2  3
  [ 1, 0, 1 ]  |  3  2  2
  [ 0, 1, 1 ]  |  2  2  3
  [ 1, 1, 1 ]  |  3  2  2
```

```
 Initial states: [ 1 ]
 Final states:  [ 3 ]
```

2.3.17 ProjectedPredicaton

▷ `ProjectedPredicaton(P, p)` (function)

The function `ProjectedPredicaton` returns the `Predicaton` *P* with the new variable position list without *p*. The alphabet is halved, ignoring the 0s and 1s entries at position *p* relative to the `VariablePositionList`, whereas the transition matrix rows are combined accordingly. Formally this corresponds to the image of homomorphism which ignores the *p*-th component of the letters applied to the deterministic finite automaton. This function is used for the interpretation of the existence quantifier.

Example

```
gap> P:=Predicaton(Automaton("det", 3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]),
> [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  1  3  3
  [ 1, 0 ]  |  2  3  3
  [ 0, 1 ]  |  3  1  3
  [ 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> Q:=ProjectedPredicaton(P, 1);;
gap> Display(Q);
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 2 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  1  2  2
  [ 1 ]  |  1  2  1
 Initial states: [ 3 ]
 Final states:  [ 2, 3 ]
gap> AcceptedWordsByPredicaton(P, 10);
[ [ 0, 0 ], [ 1, 2 ], [ 2, 4 ], [ 3, 6 ], [ 4, 8 ], [ 5, 10 ] ]
gap> AcceptedWordsByPredicaton(Q, 10);
[ [ 0 ], [ 2 ], [ 4 ], [ 6 ], [ 8 ], [ 10 ] ]
gap> PredicatonToRatExp(P);
([ 1, 0 ][ 1, 1 ]*[ 0, 1 ]U[ 0, 0 ])*
gap> PredicatonToRatExp(Q);
[ 0 ]([ 0 ]U[ 1 ])*U@
```

2.3.18 NegatedProjectedNegatedPredicaton

▷ `NegatedProjectedNegatedPredicaton(P, p)` (function)

The function `NegatedProjectedNegatedPredicaton` returns the negated (`NegatedAut` (2.2.19)), projected (`ProjectedPredicaton` (2.3.17) with *p*) and negated `Predicaton` *P*. This function is used for the interpretation of the for all quantifier.

Example

```
gap> P:=Predicaton(Automaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 2 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  2  2
  [ 0, 1 ]  |  2  2
  [ 1, 1 ]  |  1  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> AcceptedWordsByPredicaton(P, 5);
[ [ 0, 0 ], [ 1, 1 ], [ 2, 2 ], [ 3, 3 ], [ 4, 4 ], [ 5, 5 ] ]
gap> Q1:=ProjectedPredicaton(P, 1);;
gap> Display(Q1);
Predicaton: deterministic finite automaton on 2 letters with 1 state,
the variable position list [ 2 ] and the following transitions:
         |  1
---------------
  [ 0 ]  |  1
  [ 1 ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> AcceptedWordsByPredicaton(Q1, 5);
[ [ 0 ], [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
gap> Q2:=NegatedProjectedNegatedPredicaton(Q1, 2);;
gap> Display(Q2);
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> AcceptedWordsByPredicaton(Q2);
[ true ]
```

2.3.19 IntersectionPredicata

▷ `IntersectionPredicata(P1, P2, n)` (function)

The function `IntersectionPredicata` returns the intersection (`IntersectionAut` (2.2.20)) of the `Predicata` of *P1* and *P2* after resizing (`ExpandedPredicaton` (2.3.16)) and sorting (`SortedAlphabetPredicaton` (2.3.13)) the alphabet to match the new variable position list *n*.

Example

```
gap> P1:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
```

```
> [ [ 4, 2, 2, 2, 5 ], [ 2, 2, 5, 2, 2 ], [ 2, 2, 2, 3, 2 ], [ 4, 2, 2, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> Display(P1);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  4  2  2  2  5
  [ 1, 0 ]  |  2  2  5  2  2
  [ 0, 1 ]  |  2  2  2  3  2
  [ 1, 1 ]  |  4  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> AcceptedByPredicaton(P1, 10);
[ [ 4, 2 ], [ 5, 3 ] ]
gap> P2:=Predicaton(Automaton("det", 6, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 5, 2, 2, 3, 2, 6 ], [ 2, 2, 6, 2, 2, 2 ], [ 4, 2, 2, 2, 3, 2 ],
> [ 2, 2, 2, 2, 2, 2 ] ], [ 1 ], [ 6 ]), [ 1, 2 ]);;
gap> Display(P2);
Predicaton: deterministic finite automaton on 4 letters with 6 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5  6
---------------------------------
  [ 0, 0 ]  |  5  2  2  3  2  6
  [ 1, 0 ]  |  2  2  6  2  2  2
  [ 0, 1 ]  |  4  2  2  2  3  2
  [ 1, 1 ]  |  2  2  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 6 ]
gap> AcceptedByPredicaton(P2, 10);
[ [ 4, 1 ], [ 4, 2 ] ]
gap> P3:=IntersectionPredicata(P1, P2, [ 1, 2 ]);;
gap> Display(P3);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  1  2  2  2  4
  [ 1, 0 ]  |  2  2  1  2  2
  [ 0, 1 ]  |  2  2  2  3  2
  [ 1, 1 ]  |  2  2  2  2  2
 Initial states: [ 5 ]
 Final states:  [ 1 ]
gap> AcceptedByPredicaton(P3, 10);
[ [ 4, 2 ] ]
```

2.3.20 UnionPredicata

▷ `UnionPredicata(P)` (function)

The function `UnionPredicata` returns union (`UnionAut` (2.2.21)) of the `Predicata` of *P1* and *P2* after resizing (`ExpandedPredicaton` (2.3.16)) and sorting (`SortedAlphabetPredicaton` (2.3.13)) the alphabet to match the new variable position list n.

Example

```
gap> P1:=Predicaton(Automaton("det", 5, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 4, 2, 2, 2, 5 ], [ 2, 2, 5, 2, 2 ], [ 2, 2, 2, 3, 2 ], [ 4, 2, 2, 2, 2 ] ],
> [ 1 ], [ 5 ]), [ 1, 2 ]);;
gap> Display(P1);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
-----------------------------
  [ 0, 0 ]  |  4  2  2  2  5
  [ 1, 0 ]  |  2  2  5  2  2
  [ 0, 1 ]  |  2  2  2  3  2
  [ 1, 1 ]  |  4  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]
gap> AcceptedByPredicaton(P1, 10);
[ [ 4, 2 ], [ 5, 3 ] ]
gap> P2:=Predicaton(Automaton("det", 6, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 5, 2, 2, 3, 2, 6 ], [ 2, 2, 6, 2, 2, 2 ], [ 4, 2, 2, 2, 3, 2 ],
> [ 2, 2, 2, 2, 2, 2 ] ], [ 1 ], [ 6 ]), [ 1, 2 ]);;
gap> Display(P2);
Predicaton: deterministic finite automaton on 4 letters with 6 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5  6
--------------------------------
  [ 0, 0 ]  |  5  2  2  3  2  6
  [ 1, 0 ]  |  2  2  6  2  2  2
  [ 0, 1 ]  |  4  2  2  2  3  2
  [ 1, 1 ]  |  2  2  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 6 ]
gap> AcceptedByPredicaton(P2, 10);
[ [ 4, 1 ], [ 4, 2 ] ]
gap> P3:=UnionPredicata(P1, P2, [ 1, 2 ]);;
gap> Display(P3);
Predicaton: deterministic finite automaton on 4 letters with 6 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5  6
--------------------------------
  [ 0, 0 ]  |  1  6  6  3  2  6
  [ 1, 0 ]  |  6  6  1  6  6  6
  [ 0, 1 ]  |  6  3  6  6  4  6
  [ 1, 1 ]  |  6  6  6  6  2  6
 Initial states: [ 5 ]
 Final states:  [ 1 ]
gap> AcceptedWordsByPredicaton(P3, 9);
[ [ 4, 1 ], [ 4, 2 ], [ 5, 3 ] ]
```

2.3.21 PermutedAlphabetPredicaton

▷ `PermutedAlphabetPredicaton(A, l)` (function)
▷ `PermutedAbcPredicaton(A, l)` (function)

The function `PermutedAlphabetPredicaton` returns the `Predicaton` of the `Automaton` *A* with permuted alphabet according to *l* and accordingly swapped transition matrix rows. This is relevant for the first call of specific automata, where the variable order matters. E.g. the following automaton corresponds to the formula $x+y=z$, where the the variable x is at position 1, y at 2 and z at 3. So creating the the automaton recognizing the same formula but with variable x at position 3, y at 2 and z at 1 needs the permuted alphabet, i.e. each letter is permuted according to the given variable position list *l* (here `l=[ 3, 2, 1 ]`).

Example

```
gap> A:=Automaton("det", 3, [ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ], [ 1, 0, 1 ],
> [ 0, 1, 0 ], [ 0, 1, 1 ], [ 1, 1, 0 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 1, 3, 3 ],
> [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]);;
gap> DisplayAut(A);
deterministic finite automaton on 8 letters with 3 states
and the following transitions:
                |  1  2  3
---------------------------
  [ 0, 0, 0 ]   |  1  3  3
  [ 0, 0, 1 ]   |  3  2  3
  [ 1, 0, 0 ]   |  3  2  3
  [ 1, 0, 1 ]   |  2  3  3
  [ 0, 1, 0 ]   |  3  1  3
  [ 0, 1, 1 ]   |  1  3  3
  [ 1, 1, 0 ]   |  1  3  3
  [ 1, 1, 1 ]   |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> P:=PermutedAlphabetPredicaton(A, [3,2,1]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
                |  1  2  3
---------------------------
  [ 0, 0, 0 ]   |  1  3  3
  [ 1, 0, 0 ]   |  3  2  3
  [ 0, 0, 1 ]   |  3  2  3
  [ 1, 0, 1 ]   |  2  3  3
  [ 0, 1, 0 ]   |  3  1  3
  [ 1, 1, 0 ]   |  1  3  3
  [ 0, 1, 1 ]   |  1  3  3
  [ 1, 1, 1 ]   |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.3.22 PredicatonFromAut

▷ `PredicatonFromAut(A, l, n)` (function)

The function `PredicatonFromAut` returns the according to *n* resized (`ExpandedPredicaton` (2.3.16)) `Predicaton` of the `Automaton` *A* with the permuted alphabet (`PermutedAlphabetPredicaton` (2.3.21)), if the `VariablePositionList` *l* isn't sorted.

Example

```
gap> A:=Automaton("det", 3, [ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ], [ 1, 0, 1 ],
> [ 0, 1, 0 ], [ 0, 1, 1 ], [ 1, 1, 0 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 1, 3, 3 ],
> [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]);;
gap> DisplayAut(A);
deterministic finite automaton on 8 letters with 3 states
and the following transitions:
                |  1  2  3
---------------------------
  [ 0, 0, 0 ]   |  1  3  3
  [ 0, 0, 1 ]   |  3  2  3
  [ 1, 0, 0 ]   |  3  2  3
  [ 1, 0, 1 ]   |  2  3  3
  [ 0, 1, 0 ]   |  3  1  3
  [ 0, 1, 1 ]   |  1  3  3
  [ 1, 1, 0 ]   |  1  3  3
  [ 1, 1, 1 ]   |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> P:=PredicatonFromAut(A,[3,2,1],[1,2,3,4]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 16 letters with 3 states,
the variable position list [ 1, 2, 3, 4 ] and the following transitions:
                   |  1  2  3
------------------------------
  [ 0, 0, 0, 0 ]   |  1  3  3
  [ 1, 0, 0, 0 ]   |  3  2  3
  [ 0, 0, 1, 0 ]   |  3  2  3
  [ 1, 0, 1, 0 ]   |  2  3  3
  [ 0, 1, 0, 0 ]   |  3  1  3
  [ 1, 1, 0, 0 ]   |  1  3  3
  [ 0, 1, 1, 0 ]   |  1  3  3
  [ 1, 1, 1, 0 ]   |  3  2  3
  [ 0, 0, 0, 1 ]   |  1  3  3
  [ 1, 0, 0, 1 ]   |  3  2  3
  [ 0, 0, 1, 1 ]   |  3  2  3
  [ 1, 0, 1, 1 ]   |  2  3  3
  [ 0, 1, 0, 1 ]   |  3  1  3
  [ 1, 1, 0, 1 ]   |  1  3  3
  [ 0, 1, 1, 1 ]   |  1  3  3
  [ 1, 1, 1, 1 ]   |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.3.23 FinitelyManyWordsAccepted

▷ FinitelyManyWordsAccepted(*A*) (function)

The function `FinitelyManyWordsAccepted` checks if a `Predicaton` has only finitely many solutions, except the leading zero completion.

Example
```
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ],
> [ [ 4, 2, 2, 3, 5 ], [ 2, 2, 5, 2, 2 ] ], [ 1 ], [ 5 ]), [ 1 ]);;
gap> AcceptedWordsByPredicaton(P);
[ [ 4 ] ]
gap> FinitelyManyWordsAccepted(P);
true
```

2.3.24 PredicatonToRatExp

▷ PredicatonToRatExp(*P*) (function)

The function `PredicatonToRatExp` returns the regular expression of the `Automaton` or `Predicaton` *P*.

Example
```
gap> # Continued
gap> P:=Predicaton(Automaton("det", 5, [ [ 0 ], [ 1 ] ],
> [ [ 5, 5, 5, 4, 5 ], [ 2, 3, 4, 5, 5 ] ], [ 1 ], [ 4 ]), [ 1 ]);;
gap> PredicatonToRatExp(P);
[ 1 ][ 1 ][ 1 ][ 0 ]*
```

2.3.25 WordsOfRatExp

▷ WordsOfRatExp(*r*, *depth*) (function)

The function `WordsOfRatExp` returns all words which can be created from the regular expression *r* by applying the star operator at most *depth* times.

Example
```
gap> A:=Automaton("det", 3,
> [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ], [ 1, 1, 0 ],
> [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ],
> [ 3, 1, 3 ], [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ],
> [ 1 ], [ 1 ]);
< deterministic automaton on 8 letters with 3 states >
gap> r:=PredicatonToRatExp(A);
([ 1, 1, 0 ]([ 1, 0, 0 ]U[ 0, 1, 0 ]U[ 1, 1, 1 ])*
 [ 0, 0, 1 ]U[ 0, 0, 0 ]U[ 1, 0, 1 ]U[ 0, 1, 1 ])*
gap> WordsOfRatExp(r, 1);
[ [ [ 1, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ],
  [ [ 1, 1, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 1, 1, 0 ], [ 1, 1, 1 ], [ 0, 0, 1 ] ],
  [ [ 1, 1, 0 ], [  ], [ 0, 0, 1 ] ],
  [ [ 0, 0, 0 ] ], [ [ 1, 0, 1 ] ],
```

```
  [ [ 0, 1, 1 ] ],
  [ [  ] ] ]
```

2.3.26 WordsOfRatExpInterpreted

▷ WordsOfRatExpInterpreted(*r*[, *depth*]) (function)

The function `WordsOfRatExpInterpreted` returns all words which can be created from the regular expression *r* by applying the star operator at most *depth* times (default `depth=1`) as a list of natural numbers.

Example
```
gap> A:=Automaton("det", 3,
> [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ], [ 1, 1, 0 ],
> [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ],
> [ 3, 1, 3 ], [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ],
> [ 1 ], [ 1 ]);
< deterministic automaton on 8 letters with 3 states >
gap> r:=PredicatonToRatExp(A);
([ 1, 1, 0 ]([ 1, 0, 0 ]U[ 0, 1, 0 ]U[ 1, 1, 1 ])*
 [ 0, 0, 1 ]U[ 0, 0, 0 ]U[ 1, 0, 1 ]U[ 0, 1, 1 ])*
gap> WordsOfRatExpInterpreted(r, 1);
[ [ 0, 0, 0 ], [ 0, 1, 1 ], [ 1, 0, 1 ], [ 1, 1, 2 ], [ 1, 3, 4 ],
  [ 3, 1, 4 ], [ 3, 3, 6 ] ]
```

2.4 Special functions on Predicata

2.4.1 IsValidInputList

▷ IsValidInputList(*l*, *n*) (function)

The function `IsValidInputList` checks if the lists *l* and *n* correct lists for calling a `Predicaton`, i.e. both lists must contain positive unique integers and *l* must be a subset of *n*.

Example
```
gap> IsValidInputList([1,2,3], [1,2,3,4]);
true
gap> IsValidInputList([1,1,2,3], [1,2,3,4]);
Variable position list must contain unique positive integers.
false
gap> IsValidInputList([4,3,5], [4,5]);
Variable position list must be a subset of requested size list.
Compare: [ 4, 3, 5 ] with [ 4, 5 ]
false
gap> IsValidInputList([4,3,5], [3,4,5,6]);
true
```

2.4.2 BooleanPredicaton

▷ BooleanPredicaton(*B*, *n*) (function)

The function `BooleanPredicaton` returns the `Predicaton` which consists of one state. This state is a final state if *B* is `"true"` and a non-final state if *B* is `"false"`. The list *n* gives the resized variable position list.

Example

```
gap> P1:=BooleanPredicaton("true",[]);;
gap> Display(P1);
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> P2:=BooleanPredicaton("false", [ 1, 2 ]);;
gap> Display(P2);
Predicaton: deterministic finite automaton on 4 letters with 1 state,
the variable position list [ 1, 2 ] and the following transitions:
            |  1
------------------
  [ 0, 0 ]  |  1
  [ 1, 0 ]  |  1
  [ 0, 1 ]  |  1
  [ 1, 1 ]  |  1
 Initial states: [ 1 ]
 Final states:  [ ]
```

2.4.3 PredicataEqualAut

▷ `PredicataEqualAut` (global variable)

The variable `PredicataEqualAut` returns the `Automaton` which recognizes the language of $x = y$.

Example

```
gap> A:=PredicataEqualAut;
< deterministic automaton on 4 letters with 2 states >
gap> DisplayAut(A);
deterministic finite automaton on 4 letters with 2 states
and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  2  2
  [ 0, 1 ]  |  2  2
  [ 1, 1 ]  |  1  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.4.4 EqualPredicaton

▷ `EqualPredicaton(l, n)` (function)

The function EqualPredicaton returns the Predicaton which recognizes the language of $x = y$, where x is at position *l[1]* and y is at position *l[2]*. The list *n* gives the resized variable position list.

Example

```
gap> P1:=EqualPredicaton([ 1, 2 ], [ 1, 2 ]);;
gap> Display(P1);
Predicaton: deterministic finite automaton on 4 letters with 2 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  2  2
  [ 0, 1 ]  |  2  2
  [ 1, 1 ]  |  1  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> P2:=EqualPredicaton([ 3, 4 ], [ 1, 2, 3, 4 ]);;
gap> Display(P2);
Predicaton: deterministic finite automaton on 16 letters with 2 states,
the variable position list [ 1, 2, 3, 4 ] and the following transitions:
                  |  1  2
---------------------------
  [ 0, 0, 0, 0 ]  |  1  2
  [ 0, 0, 1, 0 ]  |  2  2
  [ 0, 0, 0, 1 ]  |  2  2
  [ 0, 0, 1, 1 ]  |  1  2
  [ 1, 0, 0, 0 ]  |  1  2
  [ 1, 0, 1, 0 ]  |  2  2
  [ 1, 0, 0, 1 ]  |  2  2
  [ 1, 0, 1, 1 ]  |  1  2
  [ 0, 1, 0, 0 ]  |  1  2
  [ 0, 1, 1, 0 ]  |  2  2
  [ 0, 1, 0, 1 ]  |  2  2
  [ 0, 1, 1, 1 ]  |  1  2
  [ 1, 1, 0, 0 ]  |  1  2
  [ 1, 1, 1, 0 ]  |  2  2
  [ 1, 1, 0, 1 ]  |  2  2
  [ 1, 1, 1, 1 ]  |  1  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.4.5 PredicataAdditionAut

▷ PredicataAdditionAut (global variable)

The variable PredicataAdditionAut returns the Automaton which recognizes the language $x + y = z$.

Example

```
gap> A:=PredicataAdditionAut;
< deterministic automaton on 8 letters with 3 states >
gap> DisplayAut(A);
deterministic finite automaton on 8 letters with 3 states
```

```
and the following transitions:
                |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.4.6 AdditionPredicaton

▷ AdditionPredicaton(*l*, *n*) (function)

The function AdditionPredicaton returns the Predicaton which recognizes the language $x+y=z$, where x is at position *l[1]*, y is at position *l[2]* and z is at position *l[3]*. The list *n* gives the resized variable position list.

Example

```
gap> P:=AdditionPredicaton([1,2,3],[1,2,3]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
                |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> DisplayAcceptedByPredicaton(P, [ 15, 15, 30 ]);

     | 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 -------------------------------------------------------------------
   0 | 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
   1 | 1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
   2 | 2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17
   3 | 3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18
   4 | 4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
   5 | 5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
   6 | 6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21
   7 | 7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22
   8 | 8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23
```

```
 9 | 9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24
10 | 10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
11 | 11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26
12 | 12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
13 | 13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28
14 | 14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29
15 | 15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
```

2.4.7 AdditionPredicatonNSummands

▷ `AdditionPredicatonNSummands(N, l, n)` (function)

The function `AdditionPredicatonNSummands` calls the function `AdditionPredicatonNSummandsExplicit` (2.5.2) and returns the `Predicaton` recognizing the language $x_1 + \ldots x_N = x_{N+1}$. The variables position list *l* gives the positions of the variables x_i and the list *n* gives the resized variable position list. The two functions `AdditionPredicatonNSummandsIterative` (2.5.3) and `AdditionPredicatonNSummandsRecursive` (2.5.4) create it in a more naive way, i.e. the first creates the `Predicaton` from the simple automaton recognizing $x+y=z$ step by step and the second creates the `Predicaton` recursively by splitting the variable position list.

Example

```
gap> P:=AdditionPredicatonNSummands(3, [ 1, 6, 3, 9 ], [ 1, 3, 6, 9 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 16 letters with 4 states,
the variable position list [ 1, 3, 6, 9 ] and the following transitions:
                 |  1  2  3  4
--------------------------------
  [ 0, 0, 0, 0 ]  |  1  4  2  4
  [ 1, 0, 0, 0 ]  |  4  2  4  4
  [ 0, 0, 1, 0 ]  |  4  2  4  4
  [ 1, 0, 1, 0 ]  |  2  4  3  4
  [ 0, 1, 0, 0 ]  |  4  2  4  4
  [ 1, 1, 0, 0 ]  |  2  4  3  4
  [ 0, 1, 1, 0 ]  |  2  4  3  4
  [ 1, 1, 1, 0 ]  |  4  3  4  4
  [ 0, 0, 0, 1 ]  |  4  1  4  4
  [ 1, 0, 0, 1 ]  |  1  4  2  4
  [ 0, 0, 1, 1 ]  |  1  4  2  4
  [ 1, 0, 1, 1 ]  |  4  2  4  4
  [ 0, 1, 0, 1 ]  |  1  4  2  4
  [ 1, 1, 0, 1 ]  |  4  2  4  4
  [ 0, 1, 1, 1 ]  |  4  2  4  4
  [ 1, 1, 1, 1 ]  |  2  4  3  4
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

2.4.8 TimesNPredicaton

▷ `TimesNPredicaton(N, l, n)` (function)

The function `TimesNPredicaton` returns the `Predicaton` calls the function `TimesNPredicatonExplicit` (2.5.5) and returns the `Predicaton` recognizing the language $N \cdot x = y$, where x is at position `l[1]` and y is at position `l[2]`. Note that $N \cdot x$ is a shortcut for N-times the addition of x. The list `n` gives the resized variable position list. The function `TimesNPredicatonRecursive` creates the `Predicaton` recursively from multiplications with $N < 10$.

Example

```
gap> P:=TimesNPredicaton(10, [ 1, 2 ], [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 11 states,
the variable position list [ 1, 2 ] and the following transitions:
             |  1   2   3   4   5   6   7   8   9   10 11
-------------------------------------------------
  [ 0, 0 ]   |  1   11 2   11 3   11 4   11 5   11 11
  [ 1, 0 ]   |  6   11 7   11 8   11 9   11 10 11 11
  [ 0, 1 ]   |  11 1   11 2   11 3   11 4   11 5   11
  [ 1, 1 ]   |  11 6   11 7   11 8   11 9   11 10 11
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> AcceptedByPredicaton(P, [ 10, 60 ]);
[ [ 0, 0 ], [ 1, 10 ], [ 2, 20 ], [ 3, 30 ], [ 4, 40 ], [ 5, 50 ], [ 6, 60 ] ]
```

2.4.9 SumOfProductsPredicaton

▷ `SumOfProductsPredicaton(l, m, n)` (function)

The function `SumOfProductsPredicatonExplicit` returns the `Predicaton` recognizing the language $\sum m_i \cdot x_i = 0$. The variables position list `l` gives the positions of the variables x_i, the list `m` gives the integers (positive or negative) and the list `n` gives the resized variable position list.

Example

```
gap> P:=SumOfProductsPredicaton([ 1, 2, 3 ], [ 7, 4, -5 ], [ 1, 2, 3 ]);
< Predicaton: deterministic finite automaton on 8 letters with 16 states
and the variable position list [ 1, 2, 3 ]. >
gap> DisplayAcceptedByPredicaton(P, [ 15, 15, 100 ]);

     | 0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
 ---------------------------------------------------------------------
   0 | 0    --   --   --   --   4    --   --   --   --   8    --   --   --   --   12
   1 | --   --   3    --   --   --   --   7    --   --   --   --   11   --   --   --
   2 | --   --   --   --   6    --   --   --   --   10   --   --   --   --   14   --
   3 | --   5    --   --   --   --   9    --   --   --   --   13   --   --   --   --
   4 | --   --   --   8    --   --   --   --   12   --   --   --   --   16   --   --
   5 | 7    --   --   --   --   11   --   --   --   --   15   --   --   --   --   19
   6 | --   --   10   --   --   --   --   14   --   --   --   --   18   --   --   --
   7 | --   --   --   --   13   --   --   --   --   17   --   --   --   --   21   --
   8 | --   12   --   --   --   --   16   --   --   --   --   20   --   --   --   --
   9 | --   --   --   15   --   --   --   --   19   --   --   --   --   23   --   --
```

```
   10 | 14  --  --  --  --  18  --  --  --  --  22  --  --  --  --  26
   11 | --  --  17  --  --  --  --  21  --  --  --  --  25  --  --  --
   12 | --  --  --  --  20  --  --  --  --  24  --  --  --  --  28  --
   13 | --  19  --  --  --  --  23  --  --  --  --  27  --  --  --  --
   14 | --  --  --  22  --  --  --  --  26  --  --  --  --  30  --  --
   15 | 21  --  --  --  --  25  --  --  --  --  29  --  --  --  --  33
```

2.4.10 TermEqualTermPredicaton

▷ `TermEqualTermPredicaton(l1, m1, i1, l2, m2, i2, n)` (function)

The function `TermEqualTermPredicaton` returns the `Predicaton` recognizing the language $\sum m_{1i} \cdot x_i + \sum i_1 = \sum m_{2i} \cdot y_i + \sum i_2$. The variables position lists *l1* and *l2* gives the positions of the variables x_i and y_i respectively, the lists *m1* and *m2* gives the integers (positive or negative) and the list *n* gives the resized variable position list. Note: This function allows double occurrences of the same variable in both variable position lists *l1* and *l2*. The lists *i1* and *i1* gives the integer additions on the left and right side, whereas *l1* and *m1* or *l2* and *m2* must contain at it's position `"int"`. This function calls `SumOfProductsPredicaton` (2.4.9).

Example

```
gap> # 5*x1 + 2*x1 + 4 = 6*x2 + 1*x3
gap> P:=TermEqualTermPredicaton( [ 1, 1, "int" ], [ 5, 2, "int" ], [ 4 ],
> [ 2, 3 ], [ 6, 1 ], [ ], [ 1, 2, 3 ]);
< Predicaton: deterministic finite automaton on 8 letters with 14 states
and the variable position list [ 1, 2, 3 ]. >
gap> AcceptedByPredicaton(P);
[ [ 0, 0, 4 ], [ 1, 1, 5 ], [ 2, 2, 6 ], [ 2, 3, 0 ], [ 3, 3, 7 ], [ 3, 4, 1 ],
  [ 4, 4, 8 ], [ 4, 5, 2 ], [ 5, 5, 9 ], [ 5, 6, 3 ], [ 6, 6, 10 ], [ 6, 7, 4 ],
  [ 7, 8, 5 ], [ 8, 9, 6 ], [ 8, 10, 0 ], [ 9, 10, 7 ] ]
gap> DisplayAcceptedByPredicaton(P, [10, 15, 100]);

      | 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
  -----------------------------------------------------------------
    0 | 4   --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
    1 | 11  5   --  --  --  --  --  --  --  --  --  --  --  --  --  --
    2 | 18  12  6   0   --  --  --  --  --  --  --  --  --  --  --  --
    3 | 25  19  13  7   1   --  --  --  --  --  --  --  --  --  --  --
    4 | 32  26  20  14  8   2   --  --  --  --  --  --  --  --  --  --
    5 | 39  33  27  21  15  9   3   --  --  --  --  --  --  --  --  --
    6 | 46  40  34  28  22  16  10  4   --  --  --  --  --  --  --  --
    7 | 53  47  41  35  29  23  17  11  5   --  --  --  --  --  --  --
    8 | 60  54  48  42  36  30  24  18  12  6   0   --  --  --  --  --
    9 | 67  61  55  49  43  37  31  25  19  13  7   1   --  --  --  --
   10 | 74  68  62  56  50  44  38  32  26  20  14  8   2   --  --  --
```

2.4.11 GreaterEqualNPredicaton

▷ `GreaterEqualNPredicaton(N, l, n)` (function)

The function `GreaterEqualNPredicaton` returns the `Predicaton` recognizing the language $x >= N$.

Example

```
gap> P:=GreaterEqualNPredicaton(15, [ 1 ], [ 1 ]);;
gap> P:=SortedStatesAut(P);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 8 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8
------------------------------------
  [ 0 ]  |  7  6  3  3  4  4  6  8
  [ 1 ]  |  2  5  8  3  3  4  6  8
 Initial states: [ 1 ]
 Final states:  [ 8 ]
gap> DisplayAcceptedByPredicaton(P, 29, true);
 If the following words are accepted by the given automaton, then: Y,
 otherwise if not accepted: n.
   0: n   1: n   2: n   3: n   4: n   5: n   6: n   7: n   8: n   9: n
  10: n  11: n  12: n  13: n  14: n  15: Y  16: Y  17: Y  18: Y  19: Y
  20: Y  21: Y  22: Y  23: Y  24: Y  25: Y  26: Y  27: Y  28: Y  29: Y
```

2.4.12 GreaterNPredicaton

▷ `GreaterNPredicaton(N, l, n)` (function)

The function `GreaterNPredicaton` returns the `Predicaton` recognizing the language $x > N$.

Example

```
gap> P:=GreaterNPredicaton(15, [ 1 ], [ 1 ]);;
gap> P:=SortedStatesAut(P);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  5  2  2  3  4  6
  [ 1 ]  |  5  6  2  3  4  6
 Initial states: [ 1 ]
 Final states:  [ 6 ]
gap> DisplayAcceptedByPredicaton(P, 29, true);
 If the following words are accepted by the given automaton, then: Y,
 otherwise if not accepted: n.
   0: n   1: n   2: n   3: n   4: n   5: n   6: n   7: n   8: n   9: n
  10: n  11: n  12: n  13: n  14: n  15: n  16: Y  17: Y  18: Y  19: Y
  20: Y  21: Y  22: Y  23: Y  24: Y  25: Y  26: Y  27: Y  28: Y  29: Y
```

2.4.13 SmallerEqualNPredicaton

▷ `SmallerEqualNPredicaton(N, l, n)` (function)

The function SmallerEqualNPredicaton returns the Predicaton recognizing the language $x <= N$.

Example

```
gap> P:=SmallerEqualNPredicaton(15, [ 1 ], [ 1 ]);;
gap> P:=SortedStatesAut(P);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  6  2  3  3  4  5
  [ 1 ]  |  6  2  2  3  4  5
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4, 5, 6 ]
gap> DisplayAcceptedByPredicaton(P, 29, true);
 If the following words are accepted by the given automaton, then: Y,
 otherwise if not accepted: n.
   0: Y   1: Y   2: Y   3: Y   4: Y   5: Y   6: Y   7: Y   8: Y   9: Y
  10: Y  11: Y  12: Y  13: Y  14: Y  15: Y  16: n  17: n  18: n  19: n
  20: n  21: n  22: n  23: n  24: n  25: n  26: n  27: n  28: n  29: n
```

2.4.14 SmallerNPredicaton

▷ SmallerNPredicaton(*N*, *l*, *n*) (function)

The function SmallerNPredicaton returns the Predicaton recognizing the language $x < N$.

Example

```
gap> P:=SmallerNPredicaton(15, [ 1 ], [ 1 ]);;
gap> P:=SortedStatesAut(P);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 8 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8
------------------------------------
  [ 0 ]  |  8  2  7  4  4  5  5  7
  [ 1 ]  |  3  2  6  2  4  4  5  7
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4, 5, 6, 7, 8 ]
gap> DisplayAcceptedByPredicaton(P, 29, true);
 If the following words are accepted by the given automaton,  then: Y,
 otherwise if not accepted: n.
   0: Y   1: Y   2: Y   3: Y   4: Y   5: Y   6: Y   7: Y   8: Y   9: Y
  10: Y  11: Y  12: Y  13: Y  14: Y  15: n  16: n  17: n  18: n  19: n
  20: n  21: n  22: n  23: n  24: n  25: n  26: n  27: n  28: n  29: n
```

2.4.15 GreaterEqualPredicaton

▷ GreaterEqualPredicaton(*l*, *n*) (function)

The function `GreaterEqualPredicaton` returns the `Predicaton` recognizing the language $x >= y$ with the variables position list *l* giving the positions of the variables x and y. The list *n* gives the resized variable position list.

Example
```
gap> P:=GreaterEqualPredicaton([1,2],[1,2]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 2 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  2  2
  [ 0, 1 ]  |  1  1
  [ 1, 1 ]  |  1  2
 Initial states: [ 2 ]
 Final states:  [ 2 ]
gap> DisplayAcceptedByPredicaton(P, 10);
 If the following words are accepted by the given automaton, then: YES,
 otherwise if not accepted: no.
     | 0   1   2   3   4   5   6   7   8   9   10
 -------------------------------------------------
   0 | YES no  no  no  no  no  no  no  no  no  no
   1 | YES YES no  no  no  no  no  no  no  no  no
   2 | YES YES YES no  no  no  no  no  no  no  no
   3 | YES YES YES YES no  no  no  no  no  no  no
   4 | YES YES YES YES YES no  no  no  no  no  no
   5 | YES YES YES YES YES YES no  no  no  no  no
   6 | YES YES YES YES YES YES YES no  no  no  no
   7 | YES YES YES YES YES YES YES YES no  no  no
   8 | YES YES YES YES YES YES YES YES YES no  no
   9 | YES YES YES YES YES YES YES YES YES YES no
  10 | YES YES YES YES YES YES YES YES YES YES YES
```

2.4.16 GreaterPredicaton

▷ `GreaterPredicaton(l, n)` (function)

The function `GreaterPredicaton` returns the `Predicaton` recognizing the language $x > y$ with the variables position list *l* giving the positions of the variables x and y. The list *n* gives the resized variable position list.

Example
```
gap> P:=GreaterPredicaton([1,2],[1,2]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 2 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  1  1
  [ 0, 1 ]  |  2  2
  [ 1, 1 ]  |  1  2
 Initial states: [ 2 ]
```

```
 Final states:  [ 1 ]
gap> DisplayAcceptedByPredicaton(P, 10);
 If the following words are accepted by the given automaton, then: YES,
 otherwise if not accepted: no.
     | 0   1   2   3   4   5   6   7   8   9   10
 ------------------------------------------------
   0 | no  no  no  no  no  no  no  no  no  no  no
   1 | YES no  no  no  no  no  no  no  no  no  no
   2 | YES YES no  no  no  no  no  no  no  no  no
   3 | YES YES YES no  no  no  no  no  no  no  no
   4 | YES YES YES YES no  no  no  no  no  no  no
   5 | YES YES YES YES YES no  no  no  no  no  no
   6 | YES YES YES YES YES YES no  no  no  no  no
   7 | YES YES YES YES YES YES YES no  no  no  no
   8 | YES YES YES YES YES YES YES YES no  no  no
   9 | YES YES YES YES YES YES YES YES YES no  no
  10 | YES YES YES YES YES YES YES YES YES YES no
```

2.4.17 SmallerEqualPredicaton

▷ `SmallerEqualPredicaton(l, n)` (function)

The function `SmallerEqualPredicaton` returns the `Predicaton` recognizing the language $x <= y$ with the variables position list `l` giving the positions of the variables x and y. The list `n` gives the resized variable position list.

Example

```
gap> P:=SmallerEqualPredicaton([ 1, 2 ], [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 2 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  1  1
  [ 0, 1 ]  |  2  2
  [ 1, 1 ]  |  1  2
 Initial states: [ 2 ]
 Final states:  [ 2 ]
gap> DisplayAcceptedByPredicaton(P, 15);
 If the following words are accepted by the given automaton, then: YES,
 otherwise if not accepted: no.
     | 0   1   2   3   4   5   6   7   8   9   10
 ------------------------------------------------
   0 | YES YES YES YES YES YES YES YES YES YES YES
   1 | no  YES YES YES YES YES YES YES YES YES YES
   2 | no  no  YES YES YES YES YES YES YES YES YES
   3 | no  no  no  YES YES YES YES YES YES YES YES
   4 | no  no  no  no  YES YES YES YES YES YES YES
   5 | no  no  no  no  no  YES YES YES YES YES YES
   6 | no  no  no  no  no  no  YES YES YES YES YES
   7 | no  no  no  no  no  no  no  YES YES YES YES
   8 | no  no  no  no  no  no  no  no  YES YES YES
```

```
     9 | no   no   no   no   no   no   no   no   no   YES YES
    10 | no   no   no   no   no   no   no   no   no   no   YES
```

2.4.18 SmallerPredicaton

▷ `SmallerPredicaton(l, n)` (function)

The function `SmallerPredicaton` returns the `Predicaton` recognizing the language $x < y$ with the variables position list `l` giving the positions of the variables x and y. The list `n` gives the resized variable position list.

Example

```
gap> P:=SmallerPredicaton([ 1, 2 ], [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 2 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  2  2
  [ 0, 1 ]  |  1  1
  [ 1, 1 ]  |  1  2
 Initial states: [ 2 ]
 Final states:  [ 1 ]
gap> DisplayAcceptedByPredicaton(P, 15);
 If the following words are accepted by the given automaton, then: YES,
 otherwise if not accepted: no.
    | 0   1   2   3   4   5   6   7   8   9   10
 -------------------------------------------------
   0 | no  YES YES YES YES YES YES YES YES YES YES
   1 | no  no  YES YES YES YES YES YES YES YES YES
   2 | no  no  no  YES YES YES YES YES YES YES YES
   3 | no  no  no  no  YES YES YES YES YES YES YES
   4 | no  no  no  no  no  YES YES YES YES YES YES
   5 | no  no  no  no  no  no  YES YES YES YES YES
   6 | no  no  no  no  no  no  no  YES YES YES YES
   7 | no  no  no  no  no  no  no  no  YES YES YES
   8 | no  no  no  no  no  no  no  no  no  YES YES
   9 | no  no  no  no  no  no  no  no  no  no  YES
  10 | no  no  no  no  no  no  no  no  no  no  no
```

2.5 Detailed look at the special functions on Predicata

This section explains how the sums and products are computed and describes different methods. The explicit method, which computes the transition matrix with a transition formula, is more efficient than the other given methods. The recursive and iterative methods explain a more naive way how to compute the requested automaton, but are lacking in speed. Therefore they are not used in any further computation.

2.5.1 AdditionPredicaton3Summands

▷ AdditionPredicaton3Summands(*l*, *n*) (function)
▷ AdditionPredicaton4Summands(*l*, *n*) (function)
▷ AdditionPredicaton5Summands(*l*, *n*) (function)

The functions `AdditionPredicatonNSummands` returns the `Predicaton` recognizing the language $x_1 + \ldots x_N = x_{N+1}$ for $N = 3,4,5$.

Example

```
gap> P:=AdditionPredicaton3Summands([ 1, 2, 3, 4 ],[ 1, 2, 3, 4 ]);
< Predicaton: deterministic finite automaton on 16 letters with 4 states
and the variable position list [ 1, 2, 3, 4 ]. >
gap> P:=AdditionPredicaton4Summands([ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ]);
< Predicaton: deterministic finite automaton on 32 letters with 5 states
and the variable position list [ 1, 2, 3, 4, 5 ]. >
gap> P:=AdditionPredicaton5Summands([ 1, 2, 3, 4, 5, 6 ], [ 1, 2, 3, 4, 5, 6 ]);
< Predicaton: deterministic finite automaton on 64 letters with 6 states
and the variable position list [ 1, 2, 3, 4, 5, 6 ]. >
```

2.5.2 AdditionPredicatonNSummandsExplicit

▷ AdditionPredicatonNSummandsExplicit(*N*, *l*, *n*) (function)

The function `AdditionPredicatonNSummandsExplicit` returns the `Predicaton` recognizing the language $x_1 + \ldots x_N = x_{N+1}$. The `TransitionTable` is assigned explicitly with the following transition property: The i-th state denotes carry i and there is a transition from state i to state j for the letter a if $\sum_{i=1}^{N} a_i = a_{N+1} + i + 2(j-i)$ holds. The variables position list *l* gives the positions of the variables x_i and the list *n* gives the resized variable position list.

Example

```
gap> P:=AdditionPredicatonNSummandsExplicit(3, [6, 11, 2, 9], [2, 3, 6, 7, 9, 11]);
< Predicaton: deterministic finite automaton on 64 letters with 4 states
and the variable position list [ 2, 3, 6, 7, 9, 11 ]. >
gap> P:=AdditionPredicatonNSummandsExplicit(11, [ 1..12 ], [ 1..12 ]);
< Predicaton: deterministic finite automaton on 4096 letters with 12 states
and the variable position list
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]. >
```

2.5.3 AdditionPredicatonNSummandsIterative

▷ AdditionPredicatonNSummandsIterative(*N*, *l*, *n*) (function)

The function `AdditionPredicatonNSummandsIterative` returns the `Predicaton` recognizing the language $x_1 + \ldots x_N = x_{N+1}$. The `Predicaton` is created by intersection $(N-1)$-times the simple automaton recognizing $x + y = z$. Due to intersecting and minimizing that often, this function shouldn't be used for large N (for example $N > 10$).

Example

```
gap> P:=AdditionPredicatonNSummandsIterative(7, [ 1..8 ], [ 1..8 ]);
< Predicaton: deterministic finite automaton on 256 letters with 8 states
and the variable position list [ 1, 2, 3, 4, 5, 6, 7, 8 ]. >
```

2.5.4 AdditionPredicatonNSummandsRecursive

▷ `AdditionPredicatonNSummandsRecursive(N, l, n)` (function)

The function `AdditionPredicatonNSummandsRecursive` returns the `Predicaton` recognizing the language $x_1 + \ldots x_N = x_{N+1}$. The `Predicaton` is created recursively splitting the variable position list until a length of 3 is reached, where the base cases are the simple automaton recognizing $x+y=z$. It is slightly faster than `AdditionPredicatonNSummandsIterative` (2.5.3) but nevertheless it shouldn't be used for large N (for example $N > 10$).

Example
```
gap> P:=AdditionPredicatonNSummandsRecursive(7, [ 1..8 ], [ 1..8 ]);
< Predicaton: deterministic finite automaton on 256 letters with 8 states
and the variable position list [ 1, 2, 3, 4, 5, 6, 7, 8 ]. >
```

2.5.5 TimesNPredicatonExplicit

▷ `TimesNPredicatonExplicit(N, l, n)` (function)

The function `TimesNPredicatonExplicit` returns the `Predicaton` recognizing the language $N \cdot x = y$. The `TransitionTable` is assigned explicitly with the following transition property: The i-th state denotes carry i and there is a transition from state i to state j for the letter a if $N \cdot a_1 = a_2 + i + 2(j-i)$. The variables position list `l` gives the positions of the variables x_i and the list `n` gives the resized variable position list.

Example
```
gap> P:=TimesNPredicatonExplicit(1000, [ 1, 2 ], [ 1, 2 ]);
< Predicaton: deterministic finite automaton on 4 letters with 1001 states
and the variable position list [ 1, 2 ]. >
gap> IsAcceptedByPredicaton(P, [ 1, 1000 ]);
true
gap> IsAcceptedByPredicaton(P, [ 2, 2000 ]);
true
gap> IsAcceptedByPredicaton(P, [ 3, 3000 ]);
true
```

2.5.6 TimesNPredicatonRecursive

▷ `TimesNPredicatonRecursive(N, l, n)` (function)

The function `TimesNPredicatonRecursive` returns the `Predicaton` recognizing the language $N \cdot x = y$. It splits the the multiplication into a multiplications of N_1 and N_2, where $N = N_1 \cdot N_2$.

Example
```
gap> P:=TimesNPredicatonRecursive(100, [1,2],[1,2]);
< Predicaton: deterministic finite automaton on 4 letters with 101 states
and the variable position list [ 1, 2 ]. >
gap> P:=TimesNPredicatonRecursive(1000, [1,2],[1,2]);
< Predicaton: deterministic finite automaton on 4 letters with 1001 states
and the variable position list [ 1, 2 ]. >
```

2.5.7 Times2Predicaton

▷ Times2Predicaton(l, n) (function)
▷ Times3Predicaton(l, n) (function)
▷ Times4Predicaton(l, n) (function)
▷ Times5Predicaton(l, n) (function)
▷ Times6Predicaton(l, n) (function)
▷ Times7Predicaton(l, n) (function)
▷ Times8Predicaton(l, n) (function)
▷ Times9Predicaton(l, n) (function)

The functions Times2Predicaton, Times3Predicaton,... returns the Predicaton recognizing the language $N \cdot x = y$ for $N = 2, \ldots, 9$.

Example

```
gap> P:=Times2Predicaton([ 1, 2 ], [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  1  3  3
  [ 1, 0 ]  |  2  3  3
  [ 0, 1 ]  |  3  1  3
  [ 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]
gap> P:=Times3Predicaton([ 1, 2 ], [ 1, 2 ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 4 letters with 4 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4
---------------------------
  [ 0, 0 ]  |  1  4  2  4
  [ 1, 0 ]  |  4  3  4  4
  [ 0, 1 ]  |  4  1  4  4
  [ 1, 1 ]  |  2  4  3  4
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

Chapter 3

Parsing first-order formulas

3.1 PredicataFormula – strings representing first-order formulas

3.1.1 PredicataFormulaSymbols

▷ `PredicataFormulaSymbols` (global variable)

The variable `PredicataFormulaSymbols` stores all inbuilt function symbols.

Example
```
gap> PredicataFormulaSymbols;
[ "*", "+", "-", "=", "gr", "geq", "less", "leq", "and", "or", "equiv",
"equivalent", "implies", "not", "(", ")", "[", "]", ",", ":", "A", "E" ]
```

3.1.2 PredicataIsStringType

▷ `PredicataIsStringType(S, T)` (function)

The function `PredicataIsStringType` checks if the string S represents one types T=`"variable"`, `"integer" (greater equal 0)`, `"negativeinteger"`, `"boolean"`, `"predicate"`, `"internalpredicate"`, `"quantifier"`, `"symbol"`, `"binarysymbol"`, `"unarysymbol"`. `PredicataFormulaSymbols` (3.1.1).

Example
```
gap> PredicataIsStringType("x1", "variable");
true
gap> PredicataIsStringType("1", "integer");
true
gap> PredicataIsStringType("-1", "negativeinteger");
true
gap> PredicataIsStringType("true", "boolean");
true
gap> PredicataIsStringType("A", "quantifier");
true
gap> PredicataIsStringType("+", "symbol");
true
```

3.1.3 PredicataGrammarVerification

▷ `PredicataGrammarVerification(S[, P])` (function)

The function `PredicataGrammarVerification` checks if the string *S*, with the optional argument `PredicataRepresentation` (3.3.10) *P*, is a well-formed formula in the Presburger arithmetic. First a lexical analysis is performed, checking if all symbols are correct. Then it is checked if the formula can be produced from the predefined grammar (see `PredicataGrammar` (4.1.1)). Finally, the range of the quantified variables is checked, as well as if all bounded variables doesn't also occur as free ones. Additionally, if the amount of opening and closing parenthesis differs, a corresponding message is returned.

Example
```
gap> PredicataGrammarVerification("4+x=2*y");
true
gap> PredicataGrammarVerification("(E x:3+x=2*y)");
true
gap> PredicataGrammarVerification("= , 2 + <= x 4");
false
```

3.1.4 PredicataFormula

▷ `PredicataFormula(S[, P])` (function)

The function `PredicataFormula` takes a string *S*, checks if it's a formula in the language of Presburger arithmetic (using with `PredicataGrammarVerification` (3.1.3)) and returns a `PredicataFormula` (use `PredicataGrammar` (4.1.1) for an overview of the rules). The optional input *P* is explained at `PredicataRepresentation` (3.3.10), however on default the predefined variable `PredicataList` (3.3.23) is used.

Example
```
gap> PredicataFormula("(E y: x + y = z)");
< PredicataFormula: ( E y : x + y = z ) >
```

3.1.5 IsPredicataFormula

▷ `IsPredicataFormula(f)` (function)

The function `IsPredicataFormula` checks if *f* is a `PredicataFormula`.

Example
```
gap> f:=PredicataFormula("(E y: x + y = z)");
< PredicataFormula: ( E y : x + y = z ) >
gap> IsPredicataFormula(f);
true
```

3.1.6 Display (PredicataFormula)

▷ `Display(f)` (method)

The method `Display` displays the `PredicataFormula` *f*.

Example

```
gap> f:=PredicataFormula("(A x: (E y: x = y))");
< PredicataFormula: ( A x : ( E y : x = y ) ) >
gap> Display(f);
PredicataFormula: ( A x : ( E y : x = y ) ).
```

3.1.7 View (PredicataFormula)

▷ `View(f)` (method)

The method `View` applied on a `PredicataFormula` *f* returns the object text.

Example

```
gap> f:=PredicataFormula("x + y = z");;
gap> View(f);
< PredicataFormula: x + y = z >
```

3.1.8 Print (PredicataFormula)

▷ `Print(f)` (method)

The method `Print` prints the `PredicataFormula` *f* as a string.

Example

```
gap> f:=PredicataFormula("x = 4 and not x = 5");
< PredicataFormula: x = 4 and not x = 5 >
gap> Print(f);
PredicataFormula("x = 4 and not x = 5");
gap> String(f);
"PredicataFormula(\"x = 4 and not x = 5\");"
```

3.1.9 FreeVariablesOfPredicataFormula

▷ `FreeVariablesOfPredicataFormula(f)` (function)

The function `FreeVariablesOfPredicataFormula` returns the free variables of the `PredicataFormula` *f* as a list of strings.

Example

```
gap> f:=PredicataFormula("(E n: 3*n = x) or (E m: 4*m = x)");
< PredicataFormula: ( E n : 3 * n = x ) or ( E m : 4 * m = x ) >
gap> FreeVariablesOfPredicataFormula(f);
[ "x" ]
```

3.1.10 BoundedVariablesOfPredicataFormula

▷ `BoundedVariablesOfPredicataFormula(f)` (function)

The function `BoundedVariablesOfPredicataFormula` returns the bounded variables of the `PredicataFormula` *f* as a list of strings.

Example
```
gap> f:=PredicataFormula("(E n: 3*n = x) or (E m: 4*m = x)");
< PredicataFormula: ( E n : 3 * n = x ) or ( E m : 4 * m = x ) >
gap> BoundedVariablesOfPredicataFormula(f);
[ "n", "m" ]
```

3.1.11 PredicataFormulaFormatted

▷ PredicataFormulaFormatted(*f*[, *P*]) (function)

The function PredicataFormulaFormatted adds missing parenthesis to the PredicataFormula *f* for unambiguous parsing in PredicataFormulaFormattedToTree (3.2.14).

Example
```
gap> f:=PredicataFormula("(E y: x + y = z)");
< PredicataFormula: ( E y : x + y = z ) >
gap> F:=PredicataFormulaFormatted(f);
< PredicataFormulaFormatted: ( E y : ( ( x + y ) = z ) ) >
```

3.1.12 IsPredicataFormulaFormatted

▷ IsPredicataFormulaFormatted(*f*) (function)

The function IsPredicataFormulaFormatted checks if *f* is a PredicataFormula.

Example
```
gap> f:=PredicataFormula("(E y: x + y = z)");
< PredicataFormula: ( E y : x + y = z ) >
gap> F:=PredicataFormulaFormatted(f);
< PredicataFormulaFormatted: ( E y : ( ( x + y ) = z ) ) >
gap> IsPredicataFormulaFormatted(F);
true
```

3.1.13 Display (PredicataFormulaFormatted)

▷ Display(*F*) (method)

The method Display displays the PredicataFormulaFormatted *F*.

Example
```
gap> F:=PredicataFormulaFormatted(PredicataFormula("(E y: x + y = z)"));
< PredicataFormulaFormatted: ( E y : ( ( x + y ) = z ) ) >
gap> Display(F);
PredicataFormulaFormatted: [ "(", "E", "y", ":", "(", "(", "x", "+", "y", ")",
"=", "z", ")", ")" ].
Concatenation: (Ey:((x+y)=z)).
```

3.1.14 View (PredicataFormulaFormatted)

▷ View(*f*) (method)

The method View applied on a PredicataFormulaFormatted *F* returns the object text.

Example

```
gap> f:=PredicataFormula("x + y = z");;
gap> F:=PredicataFormulaFormatted(f);;
gap> View(F);
< PredicataFormulaFormatted: ( ( x + y ) = z ) >
```

3.1.15 Print (PredicataFormulaFormatted)

▷ `Print(F)` (method)

The method `Print` prints the `PredicataFormulaFormatted` *F* as a string.

Example

```
gap> F:=PredicataFormulaFormatted(PredicataFormula("x = 4 and not x = 5"));
< PredicataFormulaFormatted: ( ( x = 4 ) and ( not ( x = 5 ) ) ) >
gap> Print(F);
PredicataFormulaFormatted(PredicataFormula(s"((x=4)and(not(x=5)))"));
gap> String(F);
"PredicataFormulaFormatted(PredicataFormula(\"((x=4)and(not(x=5)))\"));"
```

3.2 PredicataTree – converting first-order formulas into trees

3.2.1 PredicataTree

▷ `PredicataTree([r])` (function)

The function `PredicataTree` creates the a tree with root *r*, which may be empty.

Example

```
gap> PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> PredicataTree();
< PredicataTree: [ "" ] >
```

3.2.2 IsPredicataTree

▷ `IsPredicataTree(t)` (function)

The function `IsPredicataTree` checks if *t* is a `PredicataTree`.

Example

```
gap> f:=PredicataFormula("(E y: x + y = z)");
< PredicataFormula: ( E y : x + y = z ) >
gap> IsPredicataFormula(f);
true
```

3.2.3 Display (PredicataTree)

▷ `Display(t)` (method)

The method `Display` prints the `PredicataTree` *t* as a nested list, i.e. it's internal structure.

Example
```
gap> t:=PredicataTree("only one element");
< PredicataTree: [ "only one element" ] >
gap> Display(t);
PredicataTree: [ "only one element" ].
```

3.2.4 View (PredicataTree)

▷ `View(t)` (method)

The method `View` applied on a `PredicataTree` *t* returns the object text.

Example
```
gap> t:=PredicataTree("root");;
gap> View(t);
< PredicataTree: [ "root" ] >
```

3.2.5 Print (PredicataTree)

▷ `Print(t)` (method)

The method `Print` prints the `PredicataTree` *t* as a string.

Example
```
gap> t:=PredicataTree("root");;
gap> Print(t);
PredicataTree: [ "root" ]
```

3.2.6 IsEmptyPredicataTree

▷ `IsEmptyPredicataTree(t)` (function)

The function `IsEmptyPredicataTree` checks if a given `PredicataTree` *t* is empty.

Example
```
gap> t:=PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> IsEmptyPredicataTree(t);
false
```

3.2.7 RootOfPredicataTree

▷ `RootOfPredicataTree(t)` (function)

The function `RootOfPredicataTree` returns the current root of the `PredicataTree` *t*.

Example
```
gap> t:=PredicataTree("current root");
< PredicataTree: [ "current root" ] >
gap> RootOfPredicataTree(t);
"current root"
```

3.2.8 SetRootOfPredicataTree

▷ `SetRootOfPredicataTree(t, n)` (function)

The function `SetRootOfPredicataTree` changes the current root of the `PredicataTree` *t* to the input *n*.

Example
```
gap> SetRootOfPredicataTree(t, "element #2");
gap> t:=PredicataTree("element #1");
< PredicataTree: [ "element #1" ] >
gap> SetRootOfPredicataTree(t, "element #2");
gap> Display(t);
PredicataTree: [ "element #2" ].
```

3.2.9 InsertChildToPredicataTree

▷ `InsertChildToPredicataTree(t)` (function)

The function `InsertChildToPredicataTree` inserts a child to the current `PredicataTree` *t*.

Example
```
gap> t:=PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> InsertChildToPredicataTree(t);
gap> Display(t);
PredicataTree: [ "root", [ ] ].
gap> InsertChildToPredicataTree(t);
gap> Display(t);
PredicataTree: [ "root", [ ], [ ] ].
```

3.2.10 ChildOfPredicataTree

▷ `ChildOfPredicataTree(t, i)` (function)

The function `ChildOfPredicataTree"` enters the *i*-th child of the current `PredicataTree` *t*.

Example
```
gap> t:=PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> InsertChildToPredicataTree(t);
gap> ChildOfPredicataTree(t, 1);
< PredicataTree: [ "root", [ ] ] >
gap> SetRootOfPredicataTree(t, "child 1");
gap> Display(t);
PredicataTree: [ "root", [ "child 1" ] ].
```

3.2.11 NumberOfChildrenOfPredicataTree

▷ `NumberOfChildrenOfPredicataTree(t)` (function)

The function `NumberOfChildrenOfPredicataTree` returns the number of children of the current `PredicataTree` *t*.

Example
```
gap> t:=PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> NumberOfChildrenOfPredicataTree(t);
0
gap> InsertChildToPredicataTree(t);
gap> InsertChildToPredicataTree(t);
gap> NumberOfChildrenOfPredicataTree(t);
2
gap> ChildOfPredicataTree(t, 1);
< PredicataTree: [ "root", [ ], [ ] ] >
gap> SetRootOfPredicataTree(t, "child 1");
gap> Display(t);
PredicataTree: [ "root", [ "child 1" ], [ ] ].
gap> NumberOfChildrenOfPredicataTree(t);
0
gap> InsertChildToPredicataTree(t);
gap> InsertChildToPredicataTree(t);
gap> NumberOfChildrenOfPredicataTree(t);
2
gap> Display(t);
PredicataTree: [ "root", [ "child 1", [ ], [ ] ], [ ] ].
```

3.2.12 ParentOfPredicataTree

▷ `ParentOfPredicataTree(t)` (function)

The function `ParentOfPredicataTree` goes back to the parent of the current `PredicataTree` *t*.

Example
```
gap> t:=PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> InsertChildToPredicataTree(t);
gap> InsertChildToPredicataTree(t);
gap> Display(t);
PredicataTree: [ "root", [ ], [ ] ].
gap> ChildOfPredicataTree(t, 1);
< PredicataTree: [ "root", [ ], [ ] ] >
gap> SetRootOfPredicataTree(t, "child 1");
gap> ParentOfPredicataTree(t);
< PredicataTree: [ "root", [ "child 1" ], [ ] ] >
gap> ChildOfPredicataTree(t, 2);
< PredicataTree: [ "root", [ "child 1" ], [ ] ] >
gap> SetRootOfPredicataTree(t, "child 2");
gap> Display(t);
PredicataTree: [ "root", [ "child 1" ], [ "child 2" ] ].
```

3.2.13 ReturnedChildOfPredicataTree

▷ `ReturnedChildOfPredicataTree(t, i)` (function)

The function `ReturnedChildOfPredicataTree` returns the *i*-th child of the current `PredicataTree` *t* as a new tree.

Example
```
gap> t:=PredicataTree("root");
< PredicataTree: [ "root" ] >
gap> InsertChildToPredicataTree(t);
gap> ChildOfPredicataTree(t, 1);
< PredicataTree: [ "root", [ ], [ ] ] >
gap> SetRootOfPredicataTree(t, "child 1");
gap> ParentOfPredicataTree(t);
gap> r:=ReturnedChildOfPredicataTree(t, 1);
< PredicataTree: [ "child 1" ] >
```

3.2.14 PredicataFormulaFormattedToTree

▷ `PredicataFormulaFormattedToTree(F)` (function)

The function converts a `PredicataFormulaFormatted` (3.1.11) *F* to a `PredicataTree`.

Example
```
gap> f:=PredicataFormula("(E y: x+y=z and y = x)");
< PredicataFormula: ( E y : x + y = z and y = x ) >
gap> F:=PredicataFormulaFormatted(f);
< PredicataFormulaFormatted: ( E y : ( ( ( x + y ) = z ) and ( y = x ) ) ) >
gap> t:=PredicataFormulaFormattedToTree(F);
< PredicataTree: [ "E", [ "y" ], [ "and",
[ "=", [ "+", [ "x" ], [ "y" ] ], [ "z" ] ], [ "=", [ "y" ], [ "x" ] ] ] ] >
```

3.2.15 FreeVariablesOfPredicataTree

▷ `FreeVariablesOfPredicataTree(t)` (function)

The function `FreeVariablesOfPredicataTree` returns the free variables of the `PredicataTree` *t*, which have been carried over from the `PredicataFormula` (3.1.4) and the `PredicataFormulaFormatted` (3.1.11).

Example
```
gap> f:=PredicataFormula("(E y: x+y=z and y = x)");
< PredicataFormula: ( E y : x + y = z and y = x ) >
gap> F:=PredicataFormulaFormatted(f);
< PredicataFormulaFormatted: ( E y : ( ( ( x + y ) = z ) and ( y = x ) ) ) >
gap> t:=PredicataFormulaFormattedToTree(F);
< PredicataTree: [ "E", [ "y" ], [ "and", [ "=", [ "+", [ "x" ], [ "y" ] ],
[ "z" ] ], [ "=", [ "y" ], [ "x" ] ] ] ] >
gap> FreeVariablesOfPredicataTree(t);
[ "x", "z" ]
```

3.2.16 BoundedVariablesOfPredicataTree

▷ `BoundedVariablesOfPredicataTree(t)` (function)

The function `BoundedVariablesOfPredicataTree` returns the bounded variables of the `PredicataTree` *t*, which have been carried over from the `PredicataFormula` (3.1.4) and the `PredicataFormulaFormatted` (3.1.11).

Example

```
gap> f:=PredicataFormula("(E y: x+y=z and y = x)");
< PredicataFormula: ( E y : x + y = z and y = x ) >
gap> F:=PredicataFormulaFormatted(f);
< PredicataFormulaFormatted: ( E y : ( ( ( x + y ) = z ) and ( y = x ) ) ) >
gap> t:=PredicataFormulaFormattedToTree(F);
< PredicataTree: [ "E", [ "y" ], [ "and", [ "=", [ "+", [ "x" ], [ "y" ] ],
[ "z" ] ], [ "=", [ "y" ], [ "x" ] ] ] ] >
gap> BoundedVariablesOfPredicataTree(t);
[ "y" ]
```

3.2.17 PredicataTreeToPredicaton

▷ `PredicataTreeToPredicaton(t[, V])` (function)

The function `PredicataTreeToPredicaton` calls `PredicataTreeToPredicatonRecursive` (3.2.18) to turn a `PredicataTree` (3.2.1) *t* into a `Predicaton` (2.1.1). The optional argument *V* allows to specify an order for the free variables in the tree.

Example

```
gap> f:=PredicataFormula("(E y: x+y=z and y = x)");
< PredicataFormula: ( E y : x + y = z and y = x ) >
gap> F:=PredicataFormulaFormatted(f);
< PredicataFormulaFormatted: ( E y : ( ( ( x + y ) = z ) and ( y = x ) ) ) >
gap> t:=PredicataFormulaFormattedToTree(F);
< PredicataTree: [ "E", [ "y" ], [ "and", [ "=", [ "+", [ "x" ], [ "y" ] ],
[ "z" ] ], [ "=", [ "y" ], [ "x" ] ] ] ] >
gap> P:=PredicataTreeToPredicaton(t);
[ "Pred", < Predicata: deterministic finite automaton on 4 letters with 3 states
and the variable position list [ 1, 2 ]. > ]
gap> Display(P[2]);
Predicata: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  3  2  3
  [ 1, 0 ]  |  3  1  3
  [ 0, 1 ]  |  2  3  3
  [ 1, 1 ]  |  1  3  3
 Initial states: [ 2 ]
 Final states:  [ 2 ]

 The alphabet corresponds to the following variable list: [ "x", "z" ].
```

3.2.18 PredicataTreeToPredicatonRecursive

▷ `PredicataTreeToPredicatonRecursive(t, V)` (function)

The function `PredicataTreeToPredicatonRecursive` is called by `PredicataTreeToPredicaton` (3.2.17) in order to convert a `PredicataTree` *t* into a `Predicata`. The list *V* contains as first entry a list of free variables (not necessarily occurring) and as second entry a list containing the previous variables together with all bounded variables. This function goes down into the tree recursively until it reaches its leaves. Upon going up it creates the automaton of the `Predicaton` with relation to the variable position list.

Example
```
gap> F:=PredicataFormulaFormatted(PredicataFormula("(E y: x+y=z and y = x)"));;
gap> t:=PredicataFormulaFormattedToTree(F);;
gap> P:=PredicataTreeToPredicatonRecursive(t, [[ "x", "z" ], [ "x", "y", "z" ]]);;
gap> Display(P[2]);
Predicata: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 3 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  3  2  3
  [ 1, 0 ]  |  3  1  3
  [ 0, 1 ]  |  2  3  3
  [ 1, 1 ]  |  1  3  3
 Initial states: [ 2 ]
 Final states:  [ 2 ]
```

3.2.19 PredicataRepresentationOfPredicataTree

▷ `PredicataRepresentationOfPredicataTree(t)` (function)

The function `PredicataRepresentationOfPredicataTree` returns the `PredicataRepresentation` of a `PredicataTree` *t*. For more details see `PredicataRepresentation` (3.3.10).

Example
```
gap> t:=PredicataTree("root");;
gap> PredicataRepresentationOfPredicataTree(t);
< PredicataRepresentation containing the following predicates: [ ]. >
```

3.3 PredicataRepresentation – Predicata assigned with names and an arites

This section explains how to assign a name and an arity to a `Predicata` such that it can be reused in a `PredicataFormula` (3.1.4). The idea is to create elements containing the name, arity and the `Predicata` and combining them in a `PredicataRepresentation` (3.3.10). Additionally, there is a predefined variable `PredicataList` (3.3.23), which is called by the `PredicataFormula` on default, trying to simplify these quite lengthy construction.

3.3.1 PredicatonRepresentation

▷ `PredicatonRepresentation(name, arity, automaton)` (function)

The function `PredicatonRepresentation` creates the representation of a Predicaton, assigned with a *name*, an *arity* and an *automaton* (input may also be a `Predicaton`), allowing it to be called in a `PredicataFormula` (3.1.4) with `Name[x1,...xN]` (where `N` is the arity).

Example

```
gap> A:=Predicaton(Automaton("det", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ],
> [ 1, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2, 3 ]);;
gap> p:=PredicatonRepresentation("MyAdd", 3, A);
< Predicaton represented with the name: "MyAdd", the arity: 3
and the deterministic automaton on 8 letters and 3 states. >
```

3.3.2 IsPredicatonRepresentation

▷ `IsPredicatonRepresentation(p)` (function)

The function `IsPredicatonRepresentation` checks if the argument *p* is a `PredicatonRepresentation`.

Example

```
gap> A:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 1 ]), [ 1 ]);;
gap> p:=PredicatonRepresentation("EqualZero", 1, A);
< Predicaton represented with the name "EqualZero", the arity 1 and
the deterministic automaton on 2 letters and 2 states. >
gap> IsPredicatonRepresentation(p);
true
```

3.3.3 Display (PredicatonRepresentation)

▷ `Display(p)` (method)

The method `Display` prints the `PredicatonRepresentation` *p*.

Example

```
gap> A:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 1 ]), [ 1 ]);;
gap> p:=PredicatonRepresentation("EqualZero", 1, A);;
gap> Display(p);
Predicata represented with the name: EqualZero, the arity: 1 and
the following automaton:
deterministic finite automaton on 2 letters with 2 states and
the following transitions:
         |  1  2
------------------
  [ 0 ]  |  1  2
  [ 1 ]  |  2  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

3.3.4 View (PredicatonRepresentation)

▷ `View(p)` (method)

The method `View` applied on a `PredicatonRepresentation` *p* returns the object text.

Example
```
gap> A:=Automaton("det", 4, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2, 2, 3 ], [ 2, 2, 4, 2 ], [ 2, 2, 1, 2 ], [ 3, 2, 2, 4 ] ],
> [ 1 ], [ 1 ]);;
gap> p:=PredicatonRepresentation("MultipleOfThree", 2, A);;
gap> View(p);
< Predicaton represented with the name "MultipleOfThree", the arity 2 and
the deterministic automaton on 4 letters and 4 states. >
```

3.3.5 Print (PredicatonRepresentation)

▷ `Print(p)` (method)

The method `Print` prints the `PredicatonRepresentation` *p* as a string.

Example
```
gap> A:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 2 ]), [ 1 ]);;
gap> p:=PredicatonRepresentation("GreaterZero", 1, A);;
gap> Print(p);
PredicatonRepresentation("GreaterZero", 1, Automaton("det", 2,\
[ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ], [ 1 ], [ 2 ]))
gap> String(p);
"PredicatonRepresentation(\"GreaterZero\", 1, Automaton(\"det\", 2,\
[ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ], [ 1 ], [ 2 ]))"
```

3.3.6 NameOfPredicatonRepresentation

▷ `NameOfPredicatonRepresentation(p)` (function)

The function `NameOfPredicatonRepresentation` returns the `name` of *p*.

Example
```
gap> A:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 4, 2, 3, 3 ],
> [ 3, 3, 3, 2 ] ], [ 1 ], [ 3, 4, 1 ]), [ 1 ]);;
gap> p:=PredicatonRepresentation("NotTwo", 1, A);;
gap> NameOfPredicatonRepresentation(p);
"NotTwo"
```

3.3.7 ArityOfPredicatonRepresentation

▷ `ArityOfPredicatonRepresentation(p)` (function)

The function `ArityOfPredicatonRepresentation` returns the `arity` of *p*.

Example
```
gap> A:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 4, 2, 3, 3 ],
> [ 3, 3, 3, 2 ] ], [ 1 ], [ 3, 4, 1 ]), [ 1 ]);;
```

```
gap> p:=PredicatonRepresentation("NotTwo", 1, A);;
gap> ArityOfPredicatonRepresentation(p);
1
```

3.3.8 AutOfPredicatonRepresentation

▷ AutOfPredicatonRepresentation(*p*) (function)

The function AutOfPredicatonRepresentation returns the automaton of *p*.

Example

```
gap> A:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 4, 2, 3, 3 ],
> [ 3, 3, 3, 2 ] ], [ 1 ], [ 3, 4, 1 ]), [ 1 ]);;
gap> p:=PredicatonRepresentation("NotTwo", 1, A);;
gap> AutOfPredicatonRepresentation(p);
< deterministic automaton on 2 letters with 4 states >
```

3.3.9 CopyPredicatonRepresentation

▷ CopyPredicatonRepresentation(*p*) (function)

The function CopyPredicatonRepresentation copies *p*.

Example

```
gap> A:=Automaton("det", 3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]);;
gap> p:=PredicatonRepresentation("CopyPred", 2, A);;
gap> q:=CopyPredicatonRepresentation(p);
< Predicaton represented with the name "CopyPred", the arity 2 and
the deterministic automaton on 4 letters and 3 states. >
```

3.3.10 PredicataRepresentation

▷ PredicataRepresentation(*[l]*) (function)

The function PredicataRepresentation creates a collection of elements (PredicatonRepresentation (3.3.1)), where the list *l* may contain arbitrary many of them. The PredicataRepresentation is an optional input for the function PredicataFormula (3.1.4) (On default it uses the predefined variable PredicataList (3.3.23)). Note that the variables must be unique within one predicate call.

Example

```
gap> A1:=Predicaton(Automaton("det", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ],
> [ 1, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2, 3 ]);;
gap> p1:=PredicatonRepresentation("MyAdd", 3, A1);
< Predicaton represented with the name "MyAdd", the arity 3 and
the deterministic automaton on 8 letters and 3 states. >
gap> A2:=Predicaton(Automaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;
gap> p2:=PredicatonRepresentation("MyEqual", 2, A2);;
```

```
gap> P:=PredicataRepresentation(p1, p2);
< PredicataRepresentation containing the following predicates:
[ "MyEqual", "MyAdd"]. >
gap> f:=PredicataFormula("MyAdd[x,y,z] and MyEqual[x,y]", P);
< PredicataFormula: MyAdd [ x , y , z ] and MyEqual [ x , y ] >
```

3.3.11 IsPredicataRepresentation

▷ `IsPredicataRepresentation(P)` (function)

The function `IsPredicataRepresentation` checks if the argument *P* is a `PredicataRepresentation`.

Example

```
gap> # Continued example: PredicataRepresentation
gap> IsPredicataRepresentation(P);
true
```

3.3.12 Display (PredicataRepresentation)

▷ `Display(p)` (method)

The method `Display` prints the `PredicataRepresentation` *P*.

Example

```
gap> # Continued example: PredicataRepresentation
gap> Display(P);
Predicata representation containing the following PredicatonRepresentations:
Predicata represented with the name: MyEqual, the arity: 2 and
the following automaton:
deterministic finite automaton on 4 letters with 2 states and
the following transitions:
            |  1  2
---------------------
  [ 0, 0 ]  |  1  2
  [ 1, 0 ]  |  2  2
  [ 0, 1 ]  |  2  2
  [ 1, 1 ]  |  1  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]
Predicata represented with the name: MyAdd, the arity: 3 and
the following automaton:
deterministic finite automaton on 8 letters with 3 states and
the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
```

```
 [ 1, 1, 1 ]  |  3  2  3
Initial states: [ 1 ]
Final states:  [ 1 ]
```

3.3.13 View (PredicataRepresentation)

▷ View(*P*) (method)

The method View applied on a PredicataRepresentation *P* returns the object text.

Example
```
gap> P:=PredicataRepresentation();;
gap> View(P);
< PredicataRepresentation containing the following predicates: [ ]. >
```

3.3.14 Print (PredicataRepresentation)

▷ Print(*P*) (method)

The method Print prints the PredicataRepresentation *P* as a string.

Example
```
gap> # Continued example: PredicataRepresentation
gap> Print(P);
PredicataRepresentation(PredicatonRepresentation("MyEqual", 2, Automaton\
("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ], [ [ 1, 2 ], [ 2, 2 ], [\
 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ])), PredicatonRepresentation("MyAdd", 3\
, Automaton("det", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ], [ 1, 1, 0 ], [ \
0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ], [ [ 1, 3, 3 ], [ 3, 2, 3 ]\
, [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ]\
 ], [ 1 ], [ 1 ])))
gap> String(P);
"PredicataRepresentation(PredicatonRepresentation(\"MyEqual\", 2, Automa\
ton(\"det\", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ], [ [ 1, 2 ], [ 2, 2\
 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ])), PredicatonRepresentation(\"MyA\
dd\", 3, Automaton(\"det\", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ], [ 1, 1\
, 0 ], [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ], [ [ 1, 3, 3 ], [ \
3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 1, 3, 3 ], [ 1, 3, 3 ], [ \
3, 2, 3 ] ], [ 1 ], [ 1 ])), PredicatonRepresentation(\"GreaterZero\", 1\
, Automaton(\"det\", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ], [ 1 ], [ 2 ]\
)))"
```

3.3.15 NamesOfPredicataRepresentation

▷ NamesOfPredicataRepresentation(*P*) (function)

The function NamesOfPredicataRepresentation returns the names of *P*.

Example
```
gap> # Continued example: PredicataRepresentation
gap> NamesOfPredicataRepresentation(P);
[ "MyEqual", "MyAdd" ]
```

3.3.16 AritiesOfPredicatonRepresentation

▷ `AritiesOfPredicatonRepresentation(P)` (function)

The function `AritiesOfPredicatonRepresentation` returns the arities of *P*.

Example
```
gap> # Continued example: PredicataRepresentation
gap> AritiesOfPredicataRepresentation(P);
[ 2, 3 ]
```

3.3.17 AutsOfPredicataRepresentation

▷ `AutsOfPredicataRepresentation(P)` (function)

The function `AutOfPredicataRepresentation` returns the automaton of *P*.

Example
```
gap> # Continued example: PredicataRepresentation
gap> AutsOfPredicataRepresentation(P);
[ < deterministic automaton on 4 letters with 2 states >,
  < deterministic automaton on 8 letters with 3 states > ]
```

3.3.18 ElementOfPredicataRepresentation

▷ `ElementOfPredicataRepresentation(P, i)` (function)

The function `ElementOfPredicataRepresentation` returns the *i*-th element as a `PredicatonRepresentation` (3.3.1).

Example
```
gap> # Continued example: PredicataRepresentation
gap> ElementOfPredicataRepresentation(P, 1);
< Predicaton represented with the name "MyEqual", the arity 2 and
the deterministic automaton on 4 letters and 2 states. >
```

3.3.19 Add (PredicataRepresenation)

▷ `Add(P, p)` (method)

The method `Add` adds the `PredicatonRepresentation` *p* to *P*.

Example
```
gap> # Continued example: PredicataRepresentation
gap> A3:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 2 ]), [ 1 ]);;
gap> p3:=PredicatonRepresentation("GreaterZero", 1, A3);;
gap> Add(P, p3);
gap> P;
< PredicataRepresentation containing the following predicates:
[ "MyEqual", "MyAdd", "GreaterZero" ]. >
```

3.3.20 Add (PredicataRepresenation (variant 2))

▷ Add(*P, name, arity, automaton*) (method)

The method Add adds the PredicatonRepresentation created from *name*, *arity* and *automaton* to *P*.

Example

```
gap> # Continued example: PredicataRepresentation
gap> A4:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 1 ]), [ 1 ]);;
gap> Add(P, "EqualZero", 1, A4);
gap> P;
< PredicataRepresentation containing the following predicates:
[ "MyEqual", "MyAdd", "GreaterZero", "EqualZero" ]. >
```

3.3.21 Remove (PredicataRepresenation)

▷ Remove(*P, i*) (method)

The method Remove removes the *i*-th element of *P*.

Example

```
gap> A5:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 4, 2, 3, 3 ],
> [ 3, 3, 3, 2 ] ], [ 1 ], [ 3, 4, 1 ]), [ 1 ]);;
gap> p5:=PredicatonRepresentation("NotTwo", 1, A5);;
gap> Add(P, p5);
gap> P;
< PredicataRepresentation containing the following predicates:
[ "NotTwo", "MyEqual", "MyAdd", "GreaterZero", "EqualZero" ]. >
gap> Remove(P, 1);
< Predicaton represented with the name "NotTwo", the arity 1 and
the deterministic automaton on 2 letters and 4 states. >
gap> P;
< PredicataRepresentation containing the following predicates:
[ "MyEqual", "MyAdd", "GreaterZero", "EqualZero" ]. >
```

3.3.22 CopyPredicataRepresentation

▷ CopyPredicataRepresentation(*P*) (function)

The function CopyPredicataRepresentation copies the PredicataRepresentation *P*.

Example

```
gap> A:=Automaton("det", 3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]);;
gap> p:=PredicatonRepresentation("CopyPred", 2, A);;
gap> P:=PredicataRepresentation(p);
< PredicataRepresentation containing the following predicates: [ "CopyPred" ]. >
gap> Q:=CopyPredicataRepresentation(P);
< PredicataRepresentation containing the following predicates: [ "CopyPred" ]. >
```

3.3.23 PredicataList

▷ PredicataList (global variable)

The variable PredicataList is a PredicataRepresentation (3.3.10) which is called on default by the PredicataFormula (3.1.4). Together with the functions AddToPredicataList (3.3.24) and RemoveFromPredicataList (3.3.26) the intention is to be able to use own predicates without specifying to much.

Example
```
gap> PredicataList;
< PredicataRepresentation containing the following predicates: [  ]. >
gap> A1:=Predicaton(Automaton("det", 3, [ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 1, 0 ],
> [ 1, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ],
> [ [ 1, 3, 3 ], [ 3, 2, 3 ], [ 3, 2, 3 ], [ 2, 3, 3 ], [ 3, 1, 3 ],
> [ 1, 3, 3 ], [ 1, 3, 3 ], [ 3, 2, 3 ] ], [ 1 ], [ 1 ]), [ 1, 2, 3 ]);;
gap> p1:=PredicatonRepresentation("MyAdd", 3, A1);;
gap> Add(PredicataList, p1);
gap> PredicataList;
< PredicataRepresentation containing the following predicates: [ "MyAdd" ]. >
gap> f:=PredicataFormula("MyAdd[x,y,z]");
< PredicataFormula: MyAdd [ x , y , z ] >
```

3.3.24 AddToPredicataList

▷ AddToPredicataList(*p*[, *arity*, *automaton*]) (function)

The function AddToPredicataList adds either a PredicatonRepresentation *p* or the created one with *p* being a string (name) as well as the *arity* and the *automaton* to PredicataList.

Example
```
gap> # Continued example: PredicataList
gap> A2:=Predicaton(Automaton("det", 2, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ],
> [ [ 1, 2 ], [ 2, 2 ], [ 2, 2 ], [ 1, 2 ] ], [ 1 ], [ 1 ]), [ 1, 2 ]);;
gap> p2:=PredicatonRepresentation("MyEqual", 2, A2);;
gap> AddToPredicataList(p2);
gap> A3:=Predicaton(Automaton("det", 2, [ [ 0 ], [ 1 ] ], [ [ 1, 2 ], [ 2, 2 ] ],
> [ 1 ], [ 2 ]), [ 1 ]);;
gap> AddToPredicataList("GreaterZero", 1, A3);
gap> PredicataList;
gap> f:=PredicataFormula("MyAdd[x,y,z] and MyEqual[x,y]");
< PredicataFormula: MyAdd [ x , y , z ] and MyEqual [ x , y ] >
```

3.3.25 ClearPredicataList

▷ ClearPredicataList() (function)

The function ClearPredicataList clears the PredicataList.

Example
```
gap> # Continued example: PredicataList
gap> ClearPredicataList();
gap> PredicataList;
< PredicataRepresentation containing the following predicates: [ ]. >
```

3.3.26 RemoveFromPredicataList

▷ `RemoveFromPredicataList(i)` (function)

The function `RemoveFromPredicataList` removes the `i`-th element of the `PredicataList`.

Example

```
gap> A:=Predicaton(Automaton("det", 4, [ [ 0 ], [ 1 ] ], [ [ 4, 2, 3, 3 ],
> [ 3, 3, 3, 2 ] ], [ 1 ], [ 3, 4, 1 ]), [ 1 ]);;
gap> AddToPredicataList("NotTwo", 1, A);
gap> PredicataList;
< PredicataRepresentation containing the following predicates: [ "NotTwo" ]. >
gap> RemoveFromPredicataList(1);
gap> PredicataList;
< PredicataRepresentation containing the following predicates: [ ]. >
```

3.4 Converting PredicataFormulas via PredicataTrees into Predicata

3.4.1 PredicataFormulaToPredicaton

▷ `PredicataFormulaToPredicaton(f[, V])` (function)

The function `PredicataFormulaToPredicaton` takes a `PredicataFormula` (3.1.4) `f` and returns a `Predicata` which language recognizes the solutions of formula `f`. The input `f` must be a first-order formula containing the operations addition$+$ and the constant multiplication $*$ (as a shortcut), see `PredicataGrammar` (4.1.1). The optional parameter `V` (list containing strings) allows to set an order of the free variables occurring in `f`. Note that the variables must not necessarily occur in the formula (for example `x = 4` and `V:=["x","y"]`).

Example

```
gap> f:=PredicataFormula("x = 4");
< PredicataFormula: x = 4 >
gap> A:=PredicataFormulaToPredicaton(f);
Predicata: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
----------------------------
  [ 0 ]  |  4  2  2  3  5
  [ 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicata: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
```

3.4.2 StringToPredicaton

▷ StringToPredicaton(*f*[, *V*]) (function)

The function `StringToPredicaton` is the simpler version of `PredicataFormulaToPredicaton` (3.4.1), it takes an `String` *f*, converts it to a `PredicataFormula` and returns a `Predicata`. The optional parameter *V* allows to set an order for the variables.

Example

```
gap> A:=StringToPredicaton("x+y = z");
Predicata: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y", "z" ].

 Regular expression of the automaton:
   ([ 1, 1, 0 ]([ 1, 0, 0 ]U[ 0, 1, 0 ]U[ 1, 1, 1 ])*
   [ 0, 0, 1 ]U[ 0, 0, 0 ]U[ 1, 0, 1 ]U[ 0, 1, 1 ])*

 Output:
< Predicata: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
```

Chapter 4

Using Predicata

The Presburger arithmetic, named after M. Presburger [Pre29] (translation: [Sta84]), is the first-order theory of natural numbers with a binary operation called addition. In 1929, M. Presburger proved the completeness and due to the constructive proof also the decidability with quantifier elimination. In this package the concepts of automata theory are used to decide Presburger arithmetic [Büc60].

4.1 Creating Predicata from first-order formulas

4.1.1 PredicataGrammar

▷ `PredicataGrammar()` (function)

The function `PredicataGrammar` returns the accepted grammar which is allowed as an input for `PredicataFormula` (3.1.4).

Example

```
gap> PredicataGrammar();
The accepted grammar is defined as follows:

  <formula>   ::= ( <formula> )
                | ( <quantifier> <var> : <formula> )
                | <formula> <logicconnective> <formula>
                | not <formula>
                | <term> = <term>
                | <var> <compare> <var>
                | <var> <compare> <int>
                | <int> <compare> <var>
                | <predicate> [ <varlist> ]
                | <predicate>
                | <boolean>

  <term>      ::= ( <term> )
                | <term> + <term>
                | <int> * <var>
                | ( - <int> ) * <var>
                | <var>
                | <int>

  <varlist>   ::= <var> , <varlist> | <var>
```

```
<quantifier>::= A | E

<logicconnective>::= and | or | implies | equiv

<compare>   ::= < | <= | => | > | less | leq | geq | gr

<var>       ::= a | b | c | ... | x | y | z | a1 | ... | z1 | ...

<int>       ::= 0 | 1 | 2 | 3 | 4 | ...

<boolean>   ::= true | false

<predicate> ::= P | Predicate1 | ... ; any name that isn't used above
```

4.1.2 PredicataPredefinedPredicates

▷ `PredicataPredefinedPredicates()` (function)

The function `PredicataPredefinedPredicates()` returns the predefined predicates which are allowed as an input for `PredicataFormula` (3.1.4).

Example

```
gap> PredicataPredefinedPredicates();
Predefined predicates:
  * Buechi[x,y]: V_2(x)=y, where the function V_2(x) returns
                           the highest power of 2 dividing x.
  * Power[x]   : V_2(x)=x
```

4.1.3 Predicaton (PredicataFormula)

▷ `Predicaton(f)` (method)

The method `Predicaton` with a `PredicataFormula` *f* as an argument calls `PredicataFormulaToPredicaton` (3.4.1) and returns a minimal `Predicaton`.

Example

```
gap> f:=PredicataFormula("2*x = y");
< PredicataFormula: 2 * x = y >
gap> Predicaton(f);
Predicaton: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  1  3  3
  [ 1, 0 ]  |  2  3  3
  [ 0, 1 ]  |  3  1  3
  [ 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y" ].
```

```
 Regular expression of the automaton:
   ([ 1, 0 ][ 1, 1 ]*[ 0, 1 ]U[ 0, 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 3 states
and the variable position list [ 1, 2 ]. >
```

4.1.4 Predicaton (PredicataFormula with variable list)

▷ `Predicaton(f, V)` (method)

The method `Predicaton` with a `PredicataFormula` *f* and a variable list *V* as arguments calls `PredicataFormulaToPredicaton` (3.4.1) and returns a minimal `Predicaton`.

Example

```
gap> f:=PredicataFormula("2*x = y");
< PredicataFormula: 2 * x = y >
gap> Predicaton(f, [ "y", "x" ]);
Predicaton: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  1  3  3
  [ 1, 0 ]  |  3  1  3
  [ 0, 1 ]  |  2  3  3
  [ 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "y", "x" ].

 Regular expression of the automaton:
   ([ 0, 1 ][ 1, 1 ]*[ 1, 0 ]U[ 0, 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 3 states
and the variable position list [ 1, 2 ]. >
```

4.1.5 Predicaton (String)

▷ `Predicaton(S)` (method)

The method `Predicaton` with a `String` *S* as an argument calls `StringToPredicaton` (3.4.2) and returns a minimal `Predicaton`.

Example

```
gap> Predicaton("(E y: x+y = z and x = y)");
Predicaton: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  1  3  3
  [ 1, 0 ]  |  2  3  3
```

```
  [ 0, 1 ]  |  3  1  3
  [ 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "z" ].

 Regular expression of the automaton:
   ([ 1, 0 ][ 1, 1 ]*[ 0, 1 ]U[ 0, 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 3 states
and the variable position list [ 1, 2 ]. >
```

4.1.6 Predicaton (String with variable list)

▷ `Predicaton(S, V)` (method)

The method `Predicaton` with a `String` *S* and a variable list *V* as arguments calls `StringToPredicaton` (3.4.2) and returns a minimal `Predicaton`.

Example

```
gap> Predicaton("(E y: x+y = z and x = y)", [ "z", "x" ]);
Predicaton: deterministic finite automaton on 4 letters with 3 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3
------------------------
  [ 0, 0 ]  |  1  3  3
  [ 1, 0 ]  |  3  1  3
  [ 0, 1 ]  |  2  3  3
  [ 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "z", "x" ].

 Regular expression of the automaton:
   ([ 0, 1 ][ 1, 1 ]*[ 1, 0 ]U[ 0, 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 3 states
and the variable position list [ 1, 2 ]. >
```

4.1.7 VariableListOfPredicaton

▷ `VariableListOfPredicaton(P)` (function)

The function `VariableListOfPredicaton` returns the variable list of a `Predicaton` *P* containing variable strings (see `PredicataIsStringType` (3.1.2)). Note that only the functions mentioned in this section preserve the variable list, since for the resizeable `Predicata` there are no reasons to implement it. There the variable position list may be increased but there's no information on how to increase the variable list, which usually will be eliminated again.

Example

```
gap> P:=Predicaton("u3+2 = z5");
Predicaton: deterministic finite automaton on 4 letters with 4 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4
---------------------------
  [ 0, 0 ]  |  3  2  2  4
  [ 1, 0 ]  |  2  2  3  2
  [ 0, 1 ]  |  2  2  4  2
  [ 1, 1 ]  |  3  2  2  4
 Initial states: [ 1 ]
 Final states:  [ 4 ]

 The alphabet corresponds to the following variable list: [ "u3", "z5" ].

 Regular expression of the automaton:
   ([ 0, 0 ]U[ 1, 1 ])[ 1, 0 ]*[ 0, 1 ]([ 0, 0 ]U[ 1, 1 ])*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 4 states
and the variable position list [ 1, 2 ]. >
gap> VariableListOfPredicaton(P);
[ "u3", "z5" ]
```

4.1.8 SetVariableListOfPredicaton

▷ SetVariableListOfPredicaton(*P*) (function)

The function SetVariableListOfPredicaton substitutes the variables of a Predicaton *P* with a new unique variables *V*. Here only the variable names are changed, compare with VariableAdjustedPredicaton (4.1.9) where the position of the variables, i.e. the variable position list is permuted.

Example

```
gap> P:=Predicaton("x+y = z");
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y", "z" ].
```

```
 Regular expression of the automaton:
   ([ 1, 1, 0 ]([ 1, 0, 0 ]U[ 0, 1, 0 ]U[ 1, 1, 1 ])*
    [ 0, 0, 1 ]U[ 0, 0, 0 ]U[ 1, 0, 1 ]U[ 0, 1, 1 ])*

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
gap> SetVariableListOfPredicaton(P, [ "z", "y", "x" ]);
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "z", "y", "x" ].
```

4.1.9 VariableAdjustedPredicaton

▷ `VariableAdjustedPredicaton(P, V)` (function)

The function `VariableAdjustedPredicaton` takes a `Predicaton` *P* and a permuted variable list *V* and returns the alphabet-permuted `Predicaton` corresponding to the old and the new variable list, each variable position of each variable may be changed. For $x+y=z$ with `[ "x", "y", "z" ]` the function `SetVariableListOfPredicaton` (4.1.8) with `[ "z", "y", "x" ]` will change this to $z+y=x$ but keep the automaton the same. Instead this function called with `[ "z", "y", "x" ]` won't change the formula $x+y=z$ (with `[ "x", "y", "z" ]`) but instead changes the alphabet and the variable position list such that the variable `"x"` is set to the third position, `"y"` remains at the second position and `"z"` is set to the first position. Compare with `PermutedAlphabetPredicaton` (2.3.21) and `SetVariablePositionListOfPredicaton` (2.3.9).

Example

```
gap> P:=Predicaton("x+y = z");
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
```

```
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y", "z" ].

 Regular expression of the automaton:
   ([ 1, 1, 0 ]([ 1, 0, 0 ]U[ 0, 1, 0 ]U[ 1, 1, 1 ])*
    [ 0, 0, 1 ]U[ 0, 0, 0 ]U[ 1, 0, 1 ]U[ 0, 1, 1 ])*

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
gap> Q:=VariableAdjustedPredicaton(P, [ "z", "y", "x" ]);;
gap> Display(Q);
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
                |  1  2  3
----------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  1  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  1  3  3
  [ 0, 0, 1 ]  |  3  2  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  2  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "z", "y", "x" ].
```

4.1.10 VariableAdjustedPredicata

▷ VariableAdjustedPredicata(*P1*, *P2*, *V*) (function)

The function VariableAdjustedPredicata does the same as VariableAdjustedPredicaton (4.1.9) just for two Predicata *P1* and *P2* and a variable list *V* at the same time. Required for the next functions.

Example

```
gap> P1:=Predicaton("x+y = z");;
gap> P2:=Predicaton("y = 4");;
gap> L:=VariableAdjustedPredicata(P1,P2, [ "x", "z", "y"]);
[ < Predicaton: deterministic finite automaton on 8 letters
  with 3 states and the variable position list [ 1, 2, 3 ]. >,
  < Predicaton: deterministic finite automaton on 8 letters
  with 5 states and the variable position list [ 1, 2, 3 ]. >
  , [ 1, 2, 3 ] ]
gap> VariableListOfPredicaton(L[1]);
[ "x", "z", "y" ]
```

```
gap> VariableListOfPredicaton(L[2]);
[ "x", "z", "y" ]
```

4.1.11 AndPredicata

▷ `AndPredicata(P1, P2[, V])` (function)

The function `AndPredicata` returns the intersection of the two `Predicata` *P1* and *P2* where the variable list (optional, by default *V* is the union of the variables of *P1* and *P2*) defines the intersection and not the variable position. This function can be used to connect the `Predicata` of two formulas instead of calling `Predicaton` on the two with `and` connected formulas . In the example $x+y=z$ and $y=4$, even if the variable "y" doesn't have the same variable position (in the first formula position 2 and in the second position 1), will be intersected regarding the variable names.

Example

```
gap> P1:=Predicaton("x+y = z");
Predicaton: deterministic finite automaton on 8 letters with 3 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3
---------------------------
  [ 0, 0, 0 ]  |  1  3  3
  [ 1, 0, 0 ]  |  3  2  3
  [ 0, 1, 0 ]  |  3  2  3
  [ 1, 1, 0 ]  |  2  3  3
  [ 0, 0, 1 ]  |  3  1  3
  [ 1, 0, 1 ]  |  1  3  3
  [ 0, 1, 1 ]  |  1  3  3
  [ 1, 1, 1 ]  |  3  2  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y", "z" ].

 Regular expression of the automaton:
   ([ 1, 1, 0 ]([ 1, 0, 0 ]U[ 0, 1, 0 ]U[ 1, 1, 1 ])*
    [ 0, 0, 1 ]U[ 0, 0, 0 ]U[ 1, 0, 1 ]U[ 0, 1, 1 ])*

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 3 states
and the variable position list [ 1, 2, 3 ]. >
gap> P2:=Predicaton("y = 4");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  2  2  3  5
  [ 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "y" ].
```

```
 Regular expression of the automaton:
   [ 0 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> P:=AndPredicata(P1, P2, [ "x", "y", "z"]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 8 letters with 6 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
                |  1  2  3  4  5  6
-----------------------------------
  [ 0, 0, 0 ]  |  1  2  2  2  4  5
  [ 1, 0, 0 ]  |  2  2  3  2  2  2
  [ 0, 1, 0 ]  |  2  2  2  2  2  2
  [ 1, 1, 0 ]  |  2  2  2  3  2  2
  [ 0, 0, 1 ]  |  2  2  1  2  2  2
  [ 1, 0, 1 ]  |  1  2  2  2  4  5
  [ 0, 1, 1 ]  |  2  2  2  1  2  2
  [ 1, 1, 1 ]  |  2  2  2  2  2  2
 Initial states: [ 6 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y", "z" ].
gap> Q:=Predicaton("x+y = z and y = 4", [ "x", "y", "z"]);
Predicaton: deterministic finite automaton on 8 letters with 6 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
                |  1  2  3  4  5  6
-----------------------------------
  [ 0, 0, 0 ]  |  5  2  2  2  4  6
  [ 1, 0, 0 ]  |  2  2  3  2  2  2
  [ 0, 1, 0 ]  |  2  2  2  2  2  2
  [ 1, 1, 0 ]  |  2  2  2  3  2  2
  [ 0, 0, 1 ]  |  2  2  6  2  2  2
  [ 1, 0, 1 ]  |  5  2  2  2  4  6
  [ 0, 1, 1 ]  |  2  2  2  6  2  2
  [ 1, 1, 1 ]  |  2  2  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 6 ]

 The alphabet corresponds to the following variable list: [ "x", "y", "z" ].

 Regular expression of the automaton:
   ([ 0, 0, 0 ]U[ 1, 0, 1 ])([ 0, 0, 0 ]U[ 1, 0, 1 ])
   ([ 1, 1, 0 ][ 1, 0, 0 ]*[ 0, 0, 1 ]U[ 0, 1, 1 ])([ 0, 0, 0 ]U[ 1, 0, 1 ])*

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 6 states
and the variable position list [ 1, 2, 3 ]. >
```

4.1.12 OrPredicata

▷ `OrPredicata(P1, P2[, V])` (function)

The function `OrPredicata` returns the union of the two `Predicata` *P1* and *P2* with the variable list (optional, by default *V* is the union of the variables of *P1* and *P2*). This function can be used to connect the `Predicata` of two formulas instead of calling `Predicaton` on the two with `or` connected formulas.

Example

```
gap> P1:=Predicaton("u = 4");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  2  2  3  5
  [ 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "u" ].

 Regular expression of the automaton:
   [ 0 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> P2:=Predicaton("u = 2");
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  3  2  2  4
  [ 1 ]  |  2  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 4 ]

 The alphabet corresponds to the following variable list: [ "u" ].

 Regular expression of the automaton:
   [ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 4 states
and the variable position list [ 1 ]. >
gap> P:=OrPredicata(P1, P2);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  1  2  2  5  3
  [ 1 ]  |  2  2  1  2  1
```

```
 Initial states: [ 4 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "u" ].
```

4.1.13 NotPredicaton

▷ NotPredicaton(*P*[, *V*]) (function)

The function `NotPredicaton` returns the negation of the `Predicaton` *P*. This function can be used to negate the `Predicaton` instead of calling `Predicaton` on the formula with prefix `not`. The optional parameter *V* allows to adjust the variable list (with `VariableAdjustedPredicaton` (4.1.9)).

Example

```
gap> P:=Predicaton("x < 4");
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  3  2  4  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   ([ 0 ]U[ 1 ])(([ 0 ]U[ 1 ])[ 0 ]*U@)U@

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 4 states
and the variable position list [ 1 ]. >
gap> NotPredicaton(P);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  3  2  4  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 2 ]

 The alphabet corresponds to the following variable list: [ "x" ].
```

4.1.14 ImpliesPredicata

▷ ImpliesPredicata(*P1*, *P2*[, *V*]) (function)

The function `ImpliesPredicata` returns the implication of the `Predicata` *P1* and *P2* with variable list (optional, by default *V* is the union of the variables of *P1* and *P2*). This function can be used to connect the `Predicata` of two formulas instead of calling `Predicaton` on the two with `implies` connected formulas.

Example

```
gap> P1:=Predicaton("(E m: x = 4*m)");
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  4  2  3  3
  [ 1 ]  |  2  2  3  2
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ]([ 0 ]([ 0 ]U[ 1 ])*U@)U@

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 4 states
and the variable position list [ 1 ]. >
gap> P2:=Predicaton("(E n: x = 2*n)");
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  3  2  3
  [ 1 ]  |  2  2  3
 Initial states: [ 1 ]
 Final states:  [ 1, 3 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ]([ 0 ]U[ 1 ])*U@

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 3 states
and the variable position list [ 1 ]. >
gap> P:=ImpliesPredicata(P1, P2, [ "x" ]);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 1 state,
the variable position list [ 1 ] and the following transitions:
         |  1
---------------
  [ 0 ]  |  1
  [ 1 ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x" ].
```

4.1.15 EquivalentPredicata

▷ EquivalentPredicata(*P1*, *P2*[, *V*]) (function)
▷ EquivPredicata(*P1*, *P2*[, *V*]) (function)

The function EquivalentPredicata returns the equivalence of the Predicata *P1* and *P2* with the variable list (optional, by default *V* is the union of the variables of *P1* and *P2*). This function can be used to connect the Predicata of two formulas instead of calling Predicaton on the two with equiv connected formulas.

Example

```
gap> P1:=Predicaton("(E y: 2*x = 7+3*y)");
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  5  2  4  3  4  6
  [ 1 ]  |  4  3  6  4  2  3
 Initial states: [ 1 ]
 Final states:  [ 6 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   (([ 0 ][ 0 ]U[ 1 ])[ 1 ]*[ 0 ]U[ 0 ][ 1 ][ 0 ]*[ 1 ])
    ([ 1 ][ 0 ]*[ 1 ]U[ 0 ][ 1 ]*[ 0 ])*[ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 6 states
and the variable position list [ 1 ]. >
gap> P2:=Predicaton("(E k: x = 5+3*k)");
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  5  2  4  3  4  6
  [ 1 ]  |  4  3  6  4  2  3
 Initial states: [ 1 ]
 Final states:  [ 6 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   (([ 0 ][ 0 ]U[ 1 ])[ 1 ]*[ 0 ]U[ 0 ][ 1 ][ 0 ]*[ 1 ])
    ([ 1 ][ 0 ]*[ 1 ]U[ 0 ][ 1 ]*[ 0 ])*[ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 6 states
and the variable position list [ 1 ]. >
gap> P:=EquivalentPredicata(P1, P2);;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 1 state,
the variable position list [ 1 ] and the following transitions:
         |  1
```

```
---------------
  [ 0 ]  |  1
  [ 1 ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x" ].
```

4.1.16 ExistsPredicaton

▷ `ExistsPredicaton(P, v[, V])` (function)

The function `ExistsPredicaton` returns the existence quantifier with the variable *v* applied on the `Predicaton` *P*. This function can be used to quantify the `Predicaton` instead of calling `(E v: ...)` on the formula. The optional parameter *V* allows to adjust the variable list (with `VariableAdjustedPredicaton` (4.1.9)).

Example

```
gap> P:=Predicaton("5*x+6*y = n");
Predicaton: deterministic finite automaton on 8 letters with 12 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3  4  5  6  7  8  9  10 11 12
------------------------------------------------------
  [ 0, 0, 0 ]  |  1  2  2  3  4  5  2  7  2  2  12 2
  [ 1, 0, 0 ]  |  2  2  1  2  2  2  3  2  7  5  2  4
  [ 0, 1, 0 ]  |  2  2  7  2  2  2  5  2  8  9  2  12
  [ 1, 1, 0 ]  |  4  2  2  7  5  8  2  12 2  2  9  2
  [ 0, 0, 1 ]  |  7  2  2  5  12 9  2  8  2  2  6  2
  [ 1, 0, 1 ]  |  2  2  7  2  2  2  5  2  8  9  2  12
  [ 0, 1, 1 ]  |  2  2  8  2  2  2  9  2  10 11 2  6
  [ 1, 1, 1 ]  |  12 2  2  8  9  10 2  6  2  2  11 2
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "n", "x", "y" ].

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 12 states
and the variable position list [ 1, 2, 3 ]. >
gap> P:=ExistsPredicaton(P, "x");;
gap> P:=ExistsPredicaton(P, "y");;
gap> Display(P);
Predicaton: deterministic finite automaton on 2 letters with 12 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8  9  10 11 12
------------------------------------------------
  [ 0 ]  |  9  2  3  2  4  5  3  7  8  2  4  11
  [ 1 ]  |  12 3  3  2  3  2  2  2  10 7  7  6
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 7, 8, 9 ]

 The alphabet corresponds to the following variable list: [ "n" ].
```

4.1.17 ForallPredicaton

▷ `ForallPredicaton(P, v[, V])` (function)

The function `ForallPredicaton` returns the for all quantifier with the variable *v* applied on the `Predicaton` *P*. This function can be used to quantify the `Predicaton` instead of calling `(A v: ...)` on the formula. The optional parameter *V* allows to adjust the variable list (with `VariableAdjustedPredicaton` (4.1.9)).

Example

```
gap> P1:=Predicaton("(E x: (E y: 5*x+6*y = n))");
Predicaton: deterministic finite automaton on 2 letters with 12 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8  9  10 11 12
-------------------------------------------------
  [ 0 ]  |  12 2  4  5  2  2  5  7  9  9  10 11
  [ 1 ]  |  8  9  2  9  2  10 10 3  9  2  2  6
 Initial states: [ 1 ]
 Final states:  [ 1, 9, 10, 11, 12 ]

 The alphabet corresponds to the following variable list: [ "n" ].

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 12 states
and the variable position list [ 1 ]. >
gap> P2:=Predicaton("n > 19");
Predicaton: deterministic finite automaton on 2 letters with 7 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7
----------------------------------
  [ 0 ]  |  6  2  2  3  4  5  7
  [ 1 ]  |  6  7  2  2  3  5  7
 Initial states: [ 1 ]
 Final states:  [ 7 ]

 The alphabet corresponds to the following variable list: [ "n" ].

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 7 states
and the variable position list [ 1 ]. >
gap> P3:=ImpliesPredicata(P2, P1);;
gap> P:=ForallPredicaton(P3, "n");;
gap> Display(P);
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

4.1.18 LeastAcceptedNumber

▷ `LeastAcceptedNumber(P[, b])` (function)

The function `LeastAcceptedNumber` returns the `Predicaton` recognizing the least number which is accepted by the given `Predicaton` *P*. If the argument *b* is `true` (by default), then the `Predicaton` recognizing the least number greater 0 is returned (if there is one), otherwise 0 is included.

Example
```
gap> P:=Predicaton("x >= 4");
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  3  2  2  4
  [ 1 ]  |  3  4  2  4
 Initial states: [ 1 ]
 Final states:  [ 4 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   ([ 0 ]U[ 1 ])([ 0 ]U[ 1 ])[ 0 ]*[ 1 ]([ 0 ]U[ 1 ])*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 4 states
and the variable position list [ 1 ]. >
gap> L:=LeastAcceptedNumber(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  2  2  3  5
  [ 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
```

4.1.19 GreatestAcceptedNumber

▷ `GreatestAcceptedNumber(P)` (function)

The function `GreatestAcceptedNumber` returns the `Predicaton` recognizing the greatest number which is accepted by the given `Predicaton` *P*.

Example
```
gap> P:=Predicaton("(E x: 2*x = y and x < 9)");
Predicaton: deterministic finite automaton on 2 letters with 8 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8
-----------------------------------
  [ 0 ]  |  3  2  7  4  4  5  6  5
  [ 1 ]  |  2  2  8  2  4  4  5  5
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4, 5, 6, 7, 8 ]

 The alphabet corresponds to the following variable list: [ "y" ].

 Regular expression of the automaton:
   [ 0 ]](([ 1 ]([ 0 ]U[ 1 ])U[ 0 ]([ 0 ][ 0 ]U[ 1 ]))
   (([ 0 ]U[ 1 ])[ 0 ]*U@)U[ 1 ]U[ 0 ]([ 0 ]([ 1 ][ 0 ]*U@)U@)U@)U@

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 8 states
and the variable position list [ 1 ]. >
gap> G:=GreatestAcceptedNumber(P);
Predicaton: deterministic finite automaton on 2 letters with 7 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7
--------------------------------
  [ 0 ]  |  6  2  2  3  4  5  7
  [ 1 ]  |  2  2  7  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 7 ]

 The alphabet corresponds to the following variable list: [ "y" ].

 Regular expression of the automaton:
   [ 0 ][ 0 ][ 0 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 7 states
and the variable position list [ 1 ]. >
```

4.1.20 LeastNonAcceptedNumber

▷ `LeastNonAcceptedNumber(P)` (function)

The function `LeastNonAcceptedNumber` returns the `Predicaton` recognizing the Least number which is not recognized by the given `Predicaton` *P*.

Example
```
gap> P:=Predicaton("x < 4 or x > 8");
Predicaton: deterministic finite automaton on 2 letters with 7 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7
--------------------------------
  [ 0 ]  |  7  2  3  3  4  4  6
```

```
  [ 1 ]  |  5  3  3  2  4  2  4
 Initial states: [ 1 ]
 Final states:  [ 1, 3, 4, 5, 6, 7 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   ([ 0 ]([ 0 ][ 0 ]U[ 1 ])U[ 1 ]([ 0 ]U[ 1 ]))(([ 1 ][ 0 ]*[ 1 ]U[ 0 ])
   ([ 0 ]U[ 1 ])*U@)U[ 0 ]([ 0 ]([ 1 ][ 0 ]*[ 1 ]([ 0 ]U[ 1 ])*U@)U@)U[ 1 ]U@

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 7 states and
the variable position list [ 1 ]. >
gap> L:=LeastNonAcceptedNumber(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  2  2  3  5
  [ 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
```

4.1.21 GreatestNonAcceptedNumber

▷ `GreatestNonAcceptedNumber(P)` (function)

The function `GreatestNonAcceptedNumber` returns the `Predicaton` recognizing the greatest number which is not recognized by the given `Predicaton` *P*.

Example

```
gap> P:=Predicaton("(E x: (E y: 3*x + 4*y = n))");
Predicaton: deterministic finite automaton on 2 letters with 6 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6
------------------------------
  [ 0 ]  |  6  2  2  3  5  5
  [ 1 ]  |  4  5  2  5  5  2
 Initial states: [ 1 ]
 Final states:  [ 1, 5, 6 ]

 The alphabet corresponds to the following variable list: [ "n" ].
```

```
 Regular expression of the automaton:
   ([ 0 ]([ 1 ][ 0 ]*[ 1 ]U[ 0 ])U[ 1 ][ 0 ]([ 0 ]U[ 1 ])
    [ 0 ]*[ 1 ]U[ 1 ][ 1 ])([ 0 ]U[ 1 ])*U[ 0 ]U@

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 6 states
and the variable position list [ 1 ]. >
gap> G:=GreatestNonAcceptedNumber(P);
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  2  2  2  3  5
  [ 1 ]  |  4  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "n" ].

 Regular expression of the automaton:
   [ 1 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> AcceptedByPredicaton(G);
[ [ 5 ] ]
```

4.1.22 InterpretedPredicaton

▷ `InterpretedPredicaton(P)` (function)

The function `InterpretedPredicaton` returns `true` if the `Predicaton` *P* has exactly one state which is also a final state, thus the `Predicaton` is interpreted as true (if free variable occurs it is true for all natural numbers). Otherwise, `false` is returned.

Example
```
gap> P:=Predicaton("(A x: (E y: x = y))");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 Regular expression of the automaton:
   [ ]*

 Due to the automaton/regular expression the formula is true.
   true

 Output:
```

```
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [  ]. >
gap> InterpretedPredicaton(P);
The Predicaton is interpreted as True.
true
```

4.1.23 AreEquivalentPredicata

▷ AreEquivalentPredicata(*P1*, *P2*[, *b*]) (function)

The function AreEquivalentPredicata returns either true if the Predicatas *P1* and *P2* are equivalent or false otherwise. If the optional parameter *b* is true (by default) then the equivalence is computed w.r.t. the variable names, if false it is computed w.r.t. to the variable position list.

Example

```
gap> P1:=Predicaton("x=4", ["x", "y"]);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  4  2  2  3  5
  [ 1, 0 ]  |  2  2  5  2  2
  [ 0, 1 ]  |  4  2  2  3  5
  [ 1, 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x", "y" ].

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 5 states
and the variable position list [ 1, 2 ]. >
gap> P2:=Predicaton("u=4", ["u", "v", "w"]);
Predicaton: deterministic finite automaton on 8 letters with 5 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3  4  5
---------------------------------
  [ 0, 0, 0 ]  |  4  2  2  3  5
  [ 1, 0, 0 ]  |  2  2  5  2  2
  [ 0, 1, 0 ]  |  4  2  2  3  5
  [ 1, 1, 0 ]  |  2  2  5  2  2
  [ 0, 0, 1 ]  |  4  2  2  3  5
  [ 1, 0, 1 ]  |  2  2  5  2  2
  [ 0, 1, 1 ]  |  4  2  2  3  5
  [ 1, 1, 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "u", "v", "w" ].

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 5 states
and the variable position list [ 1, 2, 3 ]. >
```

```
gap> AreEquivalentPredicata(P1, P2);
The Predicaton doesn't hold for all natural numbers and is interpreted as False.
false
gap> AreEquivalentPredicata(P1, P2, false);
The Predicaton holds for all natural numbers and is interpreted as True.
true
```

4.1.24 LinearSolveOverN

▷ LinearSolveOverN(*A*, *b*[, *V*]) (function)

The function `LinearSolveOverN` returns the `Predicaton` which language recognizes the solutions x of the linear equation $A \cdot x = b$. The argument *A* is a matrix (list of lists), the argument *b* a vector (list) and the optional argument *V* allows to specify an order (here the variables are named `"x1", "x2", ...`. Note that *A* and *b* may contain also negative integers, whereas the solution is over the natural numbers.

Example

```
gap> A:=LinearSolveOverN([ [ 1, -2, 3 ], [ 3, 4, -7 ] ], [ 2, 0 ]);
Predicaton: deterministic finite automaton on 8 letters with 17 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
                | 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
----------------------------------------------------------------------
  [ 0, 0, 0 ]  | 2  2  2  2  4  2  2  2  8  2  10 2  2  2  2  2  17
  [ 1, 0, 0 ]  | 2  2  2  14 2  9  2  2  2  8  2  2  12 2  2  2  2
  [ 0, 1, 0 ]  | 11 2  17 2  2  2  2  7  2  2  2  8  2  2  2  2  2
  [ 1, 1, 0 ]  | 2  2  2  2  2  2  6  2  2  2  2  2  2  9  14 8  2
  [ 0, 0, 1 ]  | 2  2  2  3  2  4  2  2  2  16 2  2  15 2  2  2  2
  [ 1, 0, 1 ]  | 2  2  2  2  13 2  2  2  12 2  4  2  2  2  2  2  5
  [ 0, 1, 1 ]  | 2  2  2  2  2  2  17 2  2  2  2  2  2  4  3  16 2
  [ 1, 1, 1 ]  | 17 2  5  2  2  2  2  14 2  2  2  12 2  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 17 ]

 The alphabet corresponds to the following variable list: [ "x1", "x2", "x3" ].

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 17 states and
the variable position list [ 1, 2, 3 ]. >
gap> AcceptedByPredicaton(A, 10);
[ [ 1, 1, 1 ], [ 2, 9, 6 ] ]
```

4.1.25 NullSpaceOverN

▷ NullSpaceOverN(*A*[, *V*]) (function)

The function `NullSpaceOverN` returns the `Predicaton` which language recognizes the solutions x of the linear equation $A \cdot x = 0$. The argument *A* is a matrix (list of lists) and the optional argument *V* allows to specify an order (here the variables are named `"x1", "x2", ...`. Note that *A* may contain also negative integers, whereas the solution is over the natural numbers.

Example
```
gap> N:=NullSpaceOverN([[1, -2, 3],[3, 4, -7]]);
Predicaton: deterministic finite automaton on 8 letters with 13 states,
the variable position list [ 1, 2, 3 ] and the following transitions:
               |  1  2  3  4  5  6  7  8  9  10 11 12 13
---------------------------------------------------------
  [ 0, 0, 0 ]  |  1  2  2  2  4  2  2  2  8  2  2  2  2
  [ 1, 0, 0 ]  |  2  2  2  12 2  9  2  2  2  2  10 2  2
  [ 0, 1, 0 ]  |  2  2  1  2  2  2  2  7  2  8  2  2  2
  [ 1, 1, 0 ]  |  2  2  2  2  2  2  6  2  2  2  2  9  12
  [ 0, 0, 1 ]  |  2  2  2  3  2  4  2  2  2  2  13 2  2
  [ 1, 0, 1 ]  |  5  2  2  2  11 2  2  2  10 2  2  2  2
  [ 0, 1, 1 ]  |  2  2  2  2  2  2  1  2  2  2  2  4  3
  [ 1, 1, 1 ]  |  2  2  5  2  2  2  2  12 2  10 2  2  2
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x1", "x2", "x3" ].

 Output:
< Predicaton: deterministic finite automaton on 8 letters with 13 states and
the variable position list [ 1, 2, 3 ]. >
gap> AcceptedByPredicaton(N);
[ [ 0, 0, 0 ], [ 1, 8, 5 ] ]
```

4.2 Examples

4.2.1 Example 1: Getting familiar

The following example introduces the two ways of getting a `Predicaton`, either created from a first-order formula (see `PredicataGrammar` (4.1.1)), the mathematically more intuitive way, or from an `Automaton`, which at first sight may not completely obvious.

Example
```
gap> # We want a Predicaton accepting the binary representation of the number 4:
gap> DecToBin(4);
[ 0, 0, 1 ]
gap> A:=Predicaton("x = 4");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  2  2  3  5
  [ 1 ]  |  2  2  5  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ][ 0 ][ 1 ][ 0 ]*
```

```
 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> # Accepted natural numbers?
gap> IsAcceptedByPredicaton(A, [ 4 ]);
true
gap> # Accepted binary representation, also with leading zero?
gap> IsAcceptedByPredicaton(A, [ [ 0, 0, 1 ] ]);
true
gap> IsAcceptedByPredicaton(A, [ [ 0, 0, 1, 0 ] ]);
true
gap> # Indeed any leading zeros can be added or cancelled:
gap> PredicatonToRatExp(A);
[ 0 ][ 0 ][ 1 ][ 0 ]*
gap> # Now we create the Predicaton recognizing "y = 1" by hand:
gap> # Parameters:  type, states, alphabet,
gap> Aut:=Automaton("det", 3,     [ [ 0 ], [ 1 ] ],
> # transitions from letter (row) and state (column) to state (row, column)
> [ [ 3, 2, 3 ], [ 2, 3, 3 ] ],
> # initial state, final states
> [ 1 ],          [ 2 ]);
< deterministic automaton on 2 letters with 3 states >
gap> # We create the Predicaton from the automaton and the variable position list.
gap> # Here we choose "y" to be at position 2.
gap> B:=Predicaton(Aut, [ 2 ]);
< Predicaton: deterministic finite automaton on 2 letters with 3 states
and the variable position list [ 2 ]. >
gap> # We want the Predicaton "x = 4 and y = 1", so we have to set a variable to B.
gap> SetVariableListOfPredicaton(B, [ "y" ]);
gap> # Then we use AndPredicata to apply "and" according to the variable names.
gap> # Hence the Predicaton is over the alphabet [[0, 0], [1, 0], [0, 1], [1, 1]],
gap> # where the first coordinate belong to "x" and the second to "y". Note that
gap> # [ "x", "y" ] is optional, by default it's sorted alphabetically.
gap> C:=AndPredicata(A, B, [ "x", "y" ]);;
gap> Display(C);
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  2  2  2  3  5
  [ 1, 0 ]  |  2  2  5  2  2
  [ 0, 1 ]  |  4  2  2  2  2
  [ 1, 1 ]  |  2  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x", "y" ].
gap> # So C accepts in the first component of the letter the variable x
gap> # and in the second component the variable y.
gap> IsAcceptedByPredicaton(C, [ 4, 1 ]);
true
gap> IsAcceptedByPredicaton(C, [ [ 0, 0, 1 ], [ 1 ] ]);
true
```

```
gap> # Alternatively, we could have created this Predicaton simply with
gap> D:=Predicaton("x = 4 and y = 1");
Predicaton: deterministic finite automaton on 4 letters with 5 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4  5
------------------------------
  [ 0, 0 ]  |  2  2  2  3  5
  [ 1, 0 ]  |  2  2  5  2  2
  [ 0, 1 ]  |  4  2  2  2  2
  [ 1, 1 ]  |  2  2  2  2  2
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x", "y" ].

 Regular expression of the automaton:
   [ 0, 1 ][ 0, 0 ][ 1, 0 ][ 0, 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 5 states
and the variable position list [ 1, 2 ]. >
gap> DrawPredicaton(D);
gap> # Furthermore, we can use the following function to see the allowed grammar:
gap> PredicataGrammar();
```

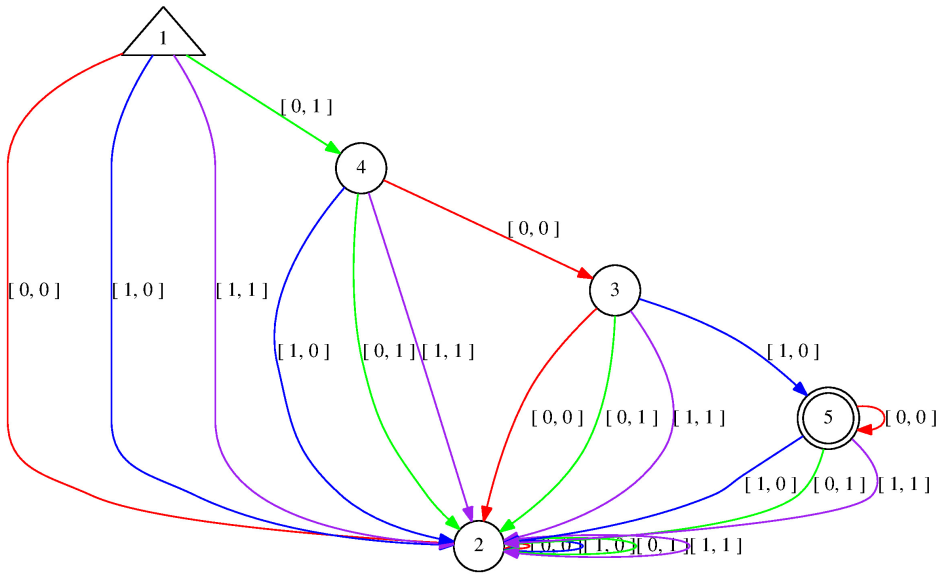

Figure 4.1: A minimal DFA recognizing $x = 4$ (x corresponding to the first component of each letter) and $y = 1$ (y corresponding to the second component).

4.2.2 Example 2: Recalling the motivation

We recall the example from the section 1. There we wanted to get the `Predicaton` recognizing all natural numbers which can be purchased by 6, 9 and 20.

Example

```
gap> # We create the Predicaton of the following formula
gap> A:=Predicaton("(E x:(E y:(E z:6*x+9*y+20*z=n)))");
Predicaton: deterministic finite automaton on 2 letters with 17 states,
 the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
---------------------------------------------------------------
  [ 0 ]  |  17 12 6  3  5  5  6  4  7  6  5  10 13 13 14 15 16
  [ 1 ]  |  2  9  13 5  13 5  3  15 10 14 14 4  13 5  5  11 8
 Initial states: [ 1 ]
 Final states:  [ 1, 13, 14, 15, 16, 17 ]

 The alphabet corresponds to the following variable list: [ "n" ].

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 17 states
and the variable position list [ 1 ]. >
gap> Display(A);
Predicaton: deterministic finite automaton on 2 letters with 17 states,
 the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
---------------------------------------------------------------
  [ 0 ]  |  17 12 6  3  5  5  6  4  7  6  5  10 13 13 14 15 16
  [ 1 ]  |  2  9  13 5  13 5  3  15 10 14 14 4  13 5  5  11 8
 Initial states: [ 1 ]
 Final states:  [ 1, 13, 14, 15, 16, 17 ]

 The alphabet corresponds to the following variable list: [ "n" ].
gap> # We ask for the accepted natural numbers.
gap> AcceptedByPredicaton(A, 20);
[ [ 0 ], [ 6 ], [ 9 ], [ 12 ], [ 15 ], [ 18 ], [ 20 ] ]
gap> DisplayAcceptedByPredicaton(A, 99, true);
 If the following words are accepted by the given automaton, then: Y,
 otherwise if not accepted: n.
   0: Y   1: n   2: n   3: n   4: n   5: n   6: Y   7: n   8: n   9: Y
  10: n  11: n  12: Y  13: n  14: n  15: Y  16: n  17: n  18: Y  19: n
  20: Y  21: Y  22: n  23: n  24: Y  25: n  26: Y  27: Y  28: n  29: Y
  30: Y  31: n  32: Y  33: Y  34: n  35: Y  36: Y  37: n  38: Y  39: Y
  40: Y  41: Y  42: Y  43: n  44: Y  45: Y  46: Y  47: Y  48: Y  49: Y
  50: Y  51: Y  52: Y  53: Y  54: Y  55: Y  56: Y  57: Y  58: Y  59: Y
  60: Y  61: Y  62: Y  63: Y  64: Y  65: Y  66: Y  67: Y  68: Y  69: Y
  70: Y  71: Y  72: Y  73: Y  74: Y  75: Y  76: Y  77: Y  78: Y  79: Y
  80: Y  81: Y  82: Y  83: Y  84: Y  85: Y  86: Y  87: Y  88: Y  89: Y
  90: Y  91: Y  92: Y  93: Y  94: Y  95: Y  96: Y  97: Y  98: Y  99: Y

gap> # We create the Predicaton accepting the greatest non-accepted number.
gap> # First we create a PredicatonRepresentation, containing a name,
gap> # an arity and an automaton (the input may also be a Predicaton).
gap> p:=PredicatonRepresentation("P", 1, A);
```

```
< Predicaton represented with the name "P", the arity 1 and
the deterministic automaton on 2 letters and 17 states. >
gap> AddToPredicataList(p);
gap> PredicataList;
< PredicataRepresentation containing the following predicates: [ "P" ]. >
gap> B:=Predicaton("(A m: m > n implies P[m]) and not P[n]");
Predicaton: deterministic finite automaton on 2 letters with 8 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8
------------------------------------
  [ 0 ]  |  2  2  2  3  2  5  2  8
  [ 1 ]  |  7  2  8  2  4  2  6  2
 Initial states: [ 1 ]
 Final states:   [ 8 ]

 The alphabet corresponds to the following variable list: [ "n" ].

 Regular expression of the automaton:
   [ 1 ][ 1 ][ 0 ][ 1 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 8 states
and the variable position list [ 1 ]. >
gap> AcceptedByPredicaton(B, 50);
[ [ 43 ] ]
gap> # We look at the regular expression and compute the natural number
gap> PredicatonToRatExp(B);
[ 1 ][ 1 ][ 0 ][ 1 ][ 0 ][ 1 ][ 0 ]*
gap> BinToDec([ 1, 1, 0, 1, 0, 1 ]);
43
gap> # Alternatively, we can also use the inbuilt function:
gap> C:=GreatestNonAcceptedNumber(A);
Predicaton: deterministic finite automaton on 2 letters with 8 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8
------------------------------------
  [ 0 ]  |  2  2  2  3  2  5  2  8
  [ 1 ]  |  7  2  8  2  4  2  6  2
 Initial states: [ 1 ]
 Final states:   [ 8 ]

 The alphabet corresponds to the following variable list: [ "n" ].

 Regular expression of the automaton:
   [ 1 ][ 1 ][ 0 ][ 1 ][ 0 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 8 states
and the variable position list [ 1 ]. >
```

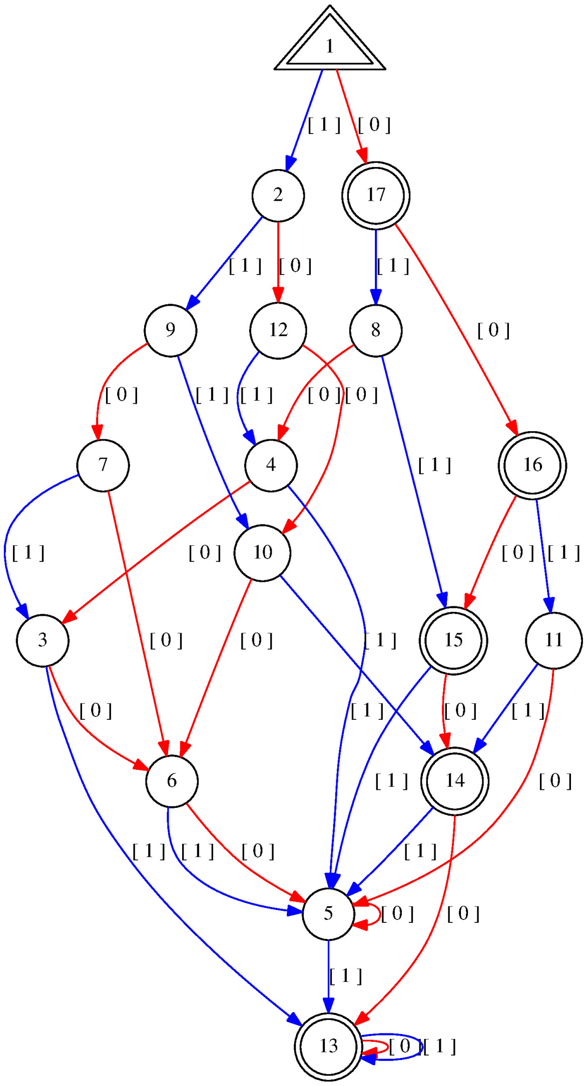

Figure 4.2: A minimal DFA recognizing the numbers which can be purchased by the formula of `A`.

4.2.3 Example 3: Divisible by three

A very common example from an automata theory lecture is finding the natural numbers which are divisible by three. Sometimes this example is solve with clear rules, sometimes with a lot of hand-waving.

However, the following way is a solid approach in the first-order language with $+$ using the shortcut `3*x:=x+x+x`.

Here, first the `Predicaton` for `3*y=x` is created with the transition rule with the `k`-th state having carry `(k-1): 3*a[1]=a[2]+(i-1)+2*((j-1)-(i-1))`. For the existence quantifier we ignore the second component of each letter, which yields a nondeterministic finite automaton. We apply the leading zero completion (see `NormalizedLeadingZeroPredicaton` (2.3.12)), i.e. any leading zero may be cancelled or added to the accepted words. Then we apply the subset construction and return the minimal automaton.

Example

```
gap> # We ask if there exists "y" s.t. 3*y=x.
gap> A:=Predicaton("(E y: 3*y = x)");
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  1  3  2
  [ 1 ]  |  2  1  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   ([ 1 ]([ 0 ][ 1 ]*[ 0 ])*[ 1 ]U[ 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 3 states
and the variable position list [ 1 ]. >
gap> # Compare with:
gap> B:=Predicaton("3*y = x");
Predicaton: deterministic finite automaton on 4 letters with 4 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4
---------------------------
  [ 0, 0 ]  |  1  2  2  3
  [ 1, 0 ]  |  2  2  1  2
  [ 0, 1 ]  |  2  2  4  2
  [ 1, 1 ]  |  3  2  2  4
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x", "y" ].

 Regular expression of the automaton:
   ([ 1, 1 ]([ 0, 1 ][ 1, 1 ]*[ 0, 0 ])*[ 1, 0 ]U[ 0, 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 4 letters with 4 states
and the variable position list [ 1, 2 ]. >
gap> Display(B);
Predicaton: deterministic finite automaton on 4 letters with 4 states,
the variable position list [ 1, 2 ] and the following transitions:
            |  1  2  3  4
---------------------------
  [ 0, 0 ]  |  1  2  2  3
  [ 1, 0 ]  |  2  2  1  2
  [ 0, 1 ]  |  2  2  4  2
  [ 1, 1 ]  |  3  2  2  4
 Initial states: [ 1 ]
 Final states:  [ 1 ]
```

```
 The alphabet corresponds to the following variable list: [ "x", "y" ].
gap> C:=ExistsPredicaton(B, "y");;
gap> Display(C);
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  1  3  2
  [ 1 ]  |  2  1  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x" ].
gap> DrawPredicaton(A);
```

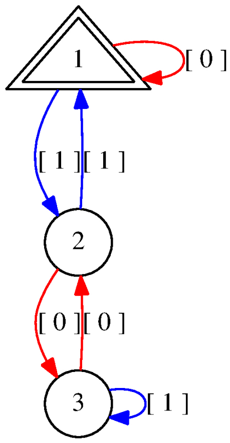

Figure 4.3: A minimal DFA recognizing the numbers divisible by 3.

4.2.4 Example 4: Linear congruences

We can solve the linear congruences $4 \cdot x = 7$ modulo 5 in the natural numbers.

Example
```
gap> A:=Predicaton("(E y: 4*x = 7+5*y)");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  1  2  3  5
  [ 1 ]  |  2  5  1  4  3
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
```

```
and the variable position list [ 1 ]. >
gap> AcceptedByPredicaton(A, 20);
[ [ 3 ], [ 8 ], [ 13 ], [ 18 ] ]
gap> # We asked for some accepted words and suggest as a solution x = 3+5*k.
gap> B:=Predicaton("(E k: x = 3+5*k)");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  4  1  2  3  5
  [ 1 ]  |  2  5  1  4  3
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> # Indeed:
gap> AreEquivalentPredicata(A, B);
The Predicaton holds for all natural numbers and is interpreted as True.
true
gap> DrawPredicaton(A);
gap> # Furthermore, we look at a system of linear congruences.
gap> C:=Predicaton("(E y1: x = 1 + 2*y1) and (E y2: x = 2 + 3*y2)");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  2  2  4  3  5
  [ 1 ]  |  4  2  5  4  3
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 1 ][ 1 ]*[ 0 ]([ 1 ][ 0 ]*[ 1 ]U[ 0 ][ 1 ]*[ 0 ])*[ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> AcceptedByPredicaton(C, 20);
[ [ 5 ], [ 11 ], [ 17 ] ]
gap> # We suggest:
gap> D:=Predicaton("(E k: x = 5 + 6 * k)");
Predicaton: deterministic finite automaton on 2 letters with 5 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5
---------------------------
  [ 0 ]  |  2  2  4  3  5
  [ 1 ]  |  4  2  5  4  3
```

```
 Initial states: [ 1 ]
 Final states:  [ 5 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 1 ][ 1 ]*[ 0 ]([ 1 ][ 0 ]*[ 1 ]U[ 0 ][ 1 ]*[ 0 ])*[ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 5 states
and the variable position list [ 1 ]. >
gap> AreEquivalentPredicata(C, D);
The Predicaton holds for all natural numbers and is interpreted as True.
true
```

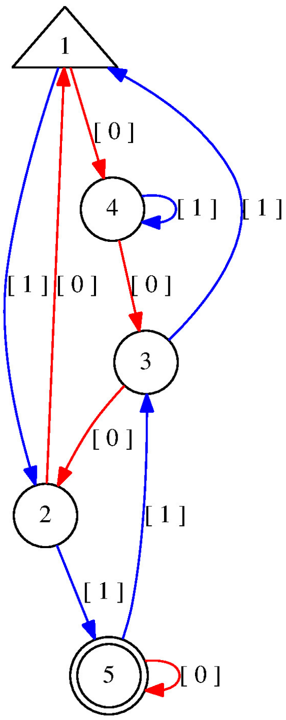

Figure 4.4: A minimal DFA recognizing the solutions of the linear congruence `A`.

4.2.5 Example 5: GCD and LCM

We can also compute the GCD and LCM of two natural numbers, however at the first sight it's not completely obvious how to obtain the GCD.

Example

```
gap> # All multiples of the GCD of 6 and 15. If there exists z such that
gap> # it is a multiple of the GCD(6, 15) after some number y, then also
gap> # z+x is a multiple of the GCD.
gap> A:=Predicaton("(E y: (A z: z>=y implies ((Ea : (Eb: z= 6*a+15*b))\
> implies (Ec: (Ed: z+x= 6*c+15*d)))))");
```

```
Predicaton: deterministic finite automaton on 2 letters with 3 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3
---------------------
  [ 0 ]  |  1  3  2
  [ 1 ]  |  2  1  3
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   ([ 1 ]([ 0 ][ 1 ]*[ 0 ])*[ 1 ]U[ 0 ])*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 3 states
and the variable position list [ 1 ]. >
gap> # This Predicaton is already known from Example 2 and we test for the least
gap> # accepted natural number greater 0 (>= 0 with optional parameter false):
gap> B:=LeastAcceptedNumber(A);
Predicaton: deterministic finite automaton on 2 letters with 4 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4
------------------------
  [ 0 ]  |  2  2  2  4
  [ 1 ]  |  3  2  4  2
 Initial states: [ 1 ]
 Final states:  [ 4 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 1 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 4 states
and the variable position list [ 1 ]. >
gap> AcceptedByPredicaton(B);
[ [ 3 ] ]
gap> # We get the multiples of the LCM(6, 15) straightforwardly.
gap> C:=Predicaton("(E a: 6*a = x) and (E b: 15*b = x)");
Predicaton: deterministic finite automaton on 2 letters with 17 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
---------------------------------------------------------------
  [ 0 ]  |  17 2  6  3  4  5  10 7  8  9  12 11 16 13 14 15 17
  [ 1 ]  |  2  2  17 6  7  13 5  3  11 16 10 4  12 8  15 9  14
 Initial states: [ 1 ]
 Final states:  [ 1, 17 ]

 The alphabet corresponds to the following variable list: [ "x" ].
```

```
 Output:
< Predicaton: deterministic finite automaton on 2 letters with 17 states
and the variable position list [ 1 ]. >
gap> D:=LeastAcceptedNumber(C);
Predicaton: deterministic finite automaton on 2 letters with 7 states,
the variable position list [ 1 ] and the following transitions:
         |  1  2  3  4  5  6  7
---------------------------------
  [ 0 ]  |  6  2  2  2  2  2  7
  [ 1 ]  |  2  2  7  3  4  5  2
 Initial states: [ 1 ]
 Final states:  [ 7 ]

 The alphabet corresponds to the following variable list: [ "x" ].

 Regular expression of the automaton:
   [ 0 ][ 1 ][ 1 ][ 1 ][ 1 ][ 0 ]*

 Output:
< Predicaton: deterministic finite automaton on 2 letters with 7 states
and the variable position list [ 1 ]. >
gap> AcceptedByPredicaton(D, 100);
[ [ 30 ] ]
```

4.2.6 Example 6: Theorems

Example

```
gap> # Which of the followings sentences are true?
gap> A1:=Predicaton("(E x:(A y: x = y))");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ ]

 Regular expression of the automaton:
   empty_set

 Due to the automaton/regular expression the formula is false.
   false

 Output:
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [ ]. >
gap> A2:=Predicaton("(A x:(E y: x = y))");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
```

```
 Final states:  [ 1 ]

 Regular expression of the automaton:
   [ ]*

 Due to the automaton/regular expression the formula is true.
   true

 Output:
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [ ]. >
gap> A3:=Predicaton("(A x:(E y: x = y+1))");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ ]

 Regular expression of the automaton:
   empty_set

 Due to the automaton/regular expression the formula is false.
   false

 Output:
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [ ]. >
gap> A4:=Predicaton("(A x:(E y: x = 2*y) or (E y: x=2*y+1))");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 Regular expression of the automaton:
   [ ]*

 Due to the automaton/regular expression the formula is true.
   true

 Output:
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [ ]. >
gap> A5:=Predicaton("(A n:(E n0: n > n0 implies (E x: (E y: 5*x+6*y=n))))");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
```

```
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 Regular expression of the automaton:
   [ ]*

 Due to the automaton/regular expression the formula is true.
   true

 Output:
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [ ]. >
gap> # Furthermore, we can use "true" and "false" as predicates;
gap> A6:=Predicaton("true and (false implies true) implies true");
Predicaton: deterministic finite automaton on 1 letter with 1 state,
the variable position list [ ] and the following transitions:
       |  1
-------------
  [ ]  |  1
 Initial states: [ 1 ]
 Final states:  [ 1 ]

 Regular expression of the automaton:
   [ ]*

 Due to the automaton/regular expression the formula is true.
   true

 Output:
< Predicaton: deterministic finite automaton on 1 letter with 1 state
and the variable position list [ ]. >
```

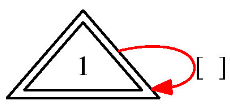

Figure 4.5: The minimal DFA which is interpreted as true.

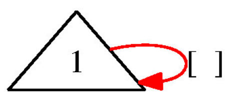

Figure 4.6: The minimal DFA which is interpreted as false.

References

[AEC16] 3rd Algorithmic and Enumerative Combinatorics Summer School 2016. https://www.risc.jku.at/conferences/aec2016/, 2016. Accessed: 2018-07-01. 2

[Büc60] J. R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6*, (6):66–92, 1960. 2, 4, 75

[DEG+02] D. Dobkin, J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz – Graph Drawing Programs. Technical report, AT&T Research and Lucent Bell Labs, 2002. 10

[DLM11] M. Delgado, S. Linton, and J. Morais. Automata, a package on automata, Version 1.13. http://www.fc.up.pt/cmup/mdelgado/automata/, 2011. Refereed GAP package. 2

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Cengage Learning, Boston, MA, USA, 2nd edition, 2001. 4

[Koz97] D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997. 4

[Pip97] N. Pippenger. *Theories of Computability*. Cambridge University Press, New York, NY, USA, 1st edition, 1997. 4

[Pre29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, page 92–101. Warszawa, 1929. 2, 4, 75

[Sha13] J. Shallit. Decidability and Enumeration for Automatic Sequences: A Survey. In A. A. Bulatov and A. M. Shur, editors, *Computer Science – Theory and Applications*, page 49–63, Berlin, Heidelberg, 2013. Springer-Verlag. 2

[Sta84] R. Stansifer. Presburger's Article on Integer Airthmetic: Remarks and Translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984. Accessed: 2018-07-01. 75

Index