JOHANNES KEPLER
UNIVERSITY LINZ

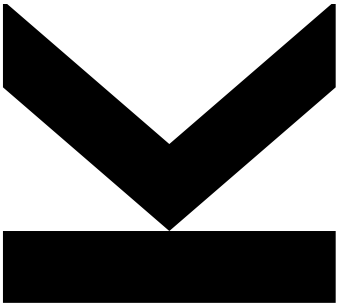
Submitted by
Fritz Kliemann, BSc

Submitted at
Institut für Algebra

Supervisor
Assoz. Univ.-Prof. Dipl.-Ing. Dr. Erhard Aichinger

October 2018

Deciding Presburger Arithmetic Using Automata Theory



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computermathematik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

The Presburger arithmetic, named after M. Presburger, is the first-order theory of the natural numbers with the addition as the only operation. In 1929, M. Presburger proved its completeness and due to his constructive proof he also showed, using quantifier elimination, that the Presburger arithmetic is decidable.

This thesis describes a decision procedure for the Presburger arithmetic based on J. R. Büchi (1960), which uses automata theory to describe the first-order formulas. The procedure describes a first-order formula as a set of natural numbers which satisfy the formula. This set, called the solutions of the formula, is associated with a minimal deterministic finite automaton such that the language of the automaton is equal to the solutions of the formula. Hence for a first-order formula, containing no free variables, the corresponding minimal deterministic finite automaton decides if the formula is an element of the Presburger arithmetic or not.

Additionally, the implemented decision procedure in *GAP – Groups, Algorithms, and Programming* is part of this thesis, allowing to construct a minimal deterministic finite automaton for any first-order formula.

Zusammenfassung

Die Presburger Arithmetik, benannt nach M. Presburger, ist die Theorie der natürlichen Zahlen mit der Addition als der einzigen Operation. M. Presburger bewies 1929 ihre Vollständigkeit und aufgrund des konstruktiven Beweises zeigte er außerdem, durch die Benützung von Quantorenelimination, dass die Presburger Arithmetik entscheidbar ist.

Diese Masterarbeit beschreibt eine Entscheidungsprozedur für die Presburger Arithmetik, die auf J. R. Büchi (1960) zurückgeht und Automatentheorie verwendet, um die Ausdrücke erster Ordnung zu beschreiben. Die Prozedur beschreibt einen Ausdruck erster Ordnung als eine Teilmenge von den natürlichen Zahlen, die den Ausdruck erfüllen. Diese Menge, genannt die Lösung des Ausdrucks, wird mit einem minimalen deterministischen endlichen Automaten so in Verbindung gebracht, dass die Sprache des Automaten gleich der Lösungen des Ausdrucks ist. Für einen Ausdruck erster Ordnung, welcher keine freien Variablen enthält, entscheidet der dazugehörige minimale deterministische endliche Automat, ob der Ausdruck ein Satz der Presburger Arithmetik ist oder nicht.

Zusätzlich ist das implementierte Entscheidungsverfahren in *GAP – Groups, Algorithms, and Programming* ein Teil dieser Masterarbeit, welches für jeden Ausdruck einen minimalen deterministischen Automaten erzeugt.

Acknowledgement

I would like to thank my thesis supervisor Erhard Aichinger for the opportunity to write this thesis and the associated GAP package. I would also like to thank my parents for their support and patience.

This thesis was partially supported by the Austrian Science Fund (FWF), P29931.

Fritz Kliemann
Linz, October 2018

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.
Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Oktober 2018

_____

Fritz Kliemann

Contents

1 Motivation

For which natural numbers $n \in \mathbb{N}_0$ does the following formula, denoted by $P[n]$ and describing the McNuggets numbers, hold:

$$\exists\, x, y, z\colon 6 \cdot x + 9 \cdot y + 20 \cdot z = n \ ?$$

Furthermore, what is the largest number that cannot be combined by multiples of 6, 9 and 20:

$$\forall m > n\ P[m] \wedge \neg P[n] \ ?$$

The idea is to take a first-order formula over the natural numbers with the addition as the only operation, allowing constant multiplication, and create a deterministic finite automaton recognizing all the natural number which satisfy the formula.

An answer to the formula $P[n]$ is the following deterministic finite automaton over the alphabet $\{0,1\}^1$, where the triangle describes the initial state, the double circles describes the final states and the transitions are either [0] or [1], building the reversed binary representation of a natural number.

Hence for example the natural numbers $6 \approx$ [0][1][1], $15 \approx$ [1][1][1][1] or $27 \approx$ [1][1][0][1][1] are accepted but not $43 \approx$ [1][1][0][1][0][1], the answer to the second question, i.e. largest number which cannot be combined by multiples of 6, 9 and 20.

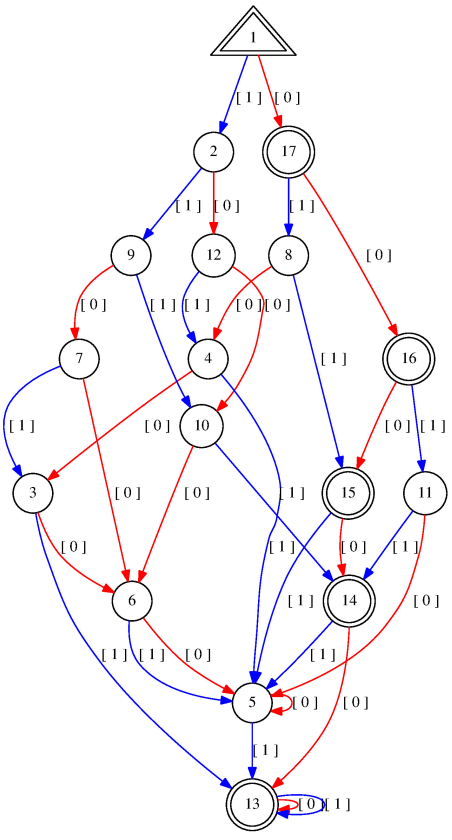


Figure 1.1: A minimal DFA recognizing the numbers which satisfy $P[n]$.

2 Automata theory

2.1 Deterministic and nondeterministic finite automata

Definition 2.1.1 A finite, nonempty set A is called *alphabet* and its elements are called *letters*. Define the sets A^+ and A^* as follows:

$$A^+ := \bigcup_{n \geq 1} A^n \qquad\qquad A^* := \bigcup_{n \geq 0} A^n.$$

A *word* w over the alphabet A is an element from A^* and w_i denotes the i-th *letter* of w.
For a word $w \in A^n$ over the alphabet A the *length* of w is defined as $|w| := n$, where $\varepsilon \in A^0$ denotes the empty word with length 0.
A *language* L over the alphabet A is a subset $L \subseteq A^*$.

[HMU01, 28ff][Pip97, 47]

Remark 2.1.2 Let $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$ be words over an alphabet A. The multiplication $x \cdot y := x_1 \dots x_n y_1 \dots y_m$ is the *concatenation* of the words, often abbreviated as xy.

Definition 2.1.3 Let A be an alphabet. A *deterministic finite automaton* (*DFA*) over the alphabet A is a quintuple (Q, A, δ, q_0, J), where

1. Q is a finite set, which elements are called *states*,
2. $\delta : A \times Q \to Q$ is called the *transition function*,
3. $q_0 \in Q$ is the *initial state* and
4. $J \subseteq Q$ is the set of *final state*s (or accepting states).

[HMU01, 46][Koz97, 15][Pip97, 55][Sip13, 35]

Definition 2.1.4 Let $M := (Q, A, \delta, q_0, J)$ be a deterministic finite automaton. The *extended transition function* $\widehat{\delta} : A^* \times Q \to Q$ of M is defined recursively on the length of the words:

$$\widehat{\delta}(\varepsilon, q) := q$$
$$\widehat{\delta}(xa, q) := \delta(a, \widehat{\delta}(x, q)) \text{ for all } x \in A^*, a \in A.$$

[HMU01, 49f][Koz97, 16]

Definition 2.1.5 Let A be an alphabet. A *nondeterministic finite automaton* (*NFA*) over the alphabet A is a quintuple (Q, A, δ, I, J), where

1. Q is a finite set, which elements are called *states*,
2. $\delta : A \times Q \to \mathcal{P}(Q)$ is called the *transition function*,
3. $I \subseteq Q$ is the set of *initial state*s and
4. $J \subseteq Q$ is the set of *final state*s (or accepting states).

[Koz97, 32][Pip97, 55]

Definition 2.1.6 Let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton. The *extended transition function* $\widehat{\delta} : A^* \times \mathcal{P}(Q) \to \mathcal{P}(Q)$ of N is defined recursively on the length of the words:

$$\begin{aligned} \widehat{\delta}(\varepsilon, P) &:= P \\ \widehat{\delta}(xa, P) &:= \bigcup_{p \in \widehat{\delta}(x,P)} \delta(a, p) \text{ for all } x \in A^*, a \in A \end{aligned}$$

[HMU01, 58][Koz97, 33]

Definition 2.1.7 Let A be an alphabet. An *ε-nondeterministic finite automaton* (*ε-NFA*) over the alphabet A is a quintuple (Q, A, δ, I, J), where

1. Q is a finite set, which elements are called *states*,
2. $\delta : (A \cup \{\varepsilon\}) \times Q \to \mathcal{P}(Q)$ is called the *transition function*,
3. $I \subseteq Q$ is the set of *initial state*s and
4. $J \subseteq Q$ is the set of *final state*s (or accepting states).

[HMU01, 74]

Definition 2.1.8 Let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton or an ε-nondeterministic finite automaton. The ε-closure of a state $q \in Q$ is defined recursively:

1. $q \in \text{ECLOSE}(q)$.

Let $p \in \text{ECLOSE}(q)$.

2. If there exists $r \in Q$ such that $\delta(\varepsilon, p) = r$ then $r \in \text{ECLOSE}(q)$.

Let $P \subseteq Q$ be a set of the states. Then

$$\text{ECLOSE}(P) := \bigcup_{p \in P} \text{ECLOSE}(p).$$

Note that for $P_1, P_2 \subseteq Q$:

$$\text{ECLOSE}(P_1 \cup P_2) = \text{ECLOSE}(P_1) \cup \text{ECLOSE}(P_2)$$

[HMU01, 75]

Definition 2.1.9 Let $N_\varepsilon := (Q, A, \delta, I, J)$ be an ε-nondeterministic finite automaton. The *extended transition function* $\widehat{\delta} : A^* \times \mathcal{P}(Q) \to \mathcal{P}(Q)$ of N_ε is defined recursively on the length of the words:

$$\begin{aligned} \widehat{\delta}(\varepsilon, P) &:= \text{ECLOSE}(P) \\ \widehat{\delta}(xa, P) &:= \text{ECLOSE}\big(\bigcup_{p \in \widehat{\delta}(x,P)} \delta(a, p)\big) \text{ for all } x \in A^*, a \in A \end{aligned}$$

[HMU01, 77]

Definition 2.1.10 Let V be a nonempty set and E an anti-reflexive relation on V. Then $G := (V, E)$ is a *digraph*, where the elements in V are called *vertices* and the elements in E are called *edges*. A *transition diagram* of a deterministic finite automaton or a nondeterministic finite automaton (or a ε-NFA) is the diagram of the digraph such that:

1. The vertices are labelled with the states and are indicated as circles.
2. The edges are labelled with the letters if there is a transition.
3. The initial states are described by an ingoing arrow from nowhere (or by a triangle).
4. The final states are indicated by a double circle.

[HMU01, 48][Sip13, 34]

Example 2.1.11 Let $A := \{a, b\}$ be an alphabet with two letters. We take a set of states $Q := \{0, 1, 2\}$, an initial state $q_0 = 0$, a set of final states $J := \{2\}$ and a transition function

$$\delta : A \times Q \to Q$$

$$\begin{aligned} \delta(a, 0) &:= 0 & \qquad \delta(b, 0) &:= 1 \\ \delta(a, 1) &:= 1 & \qquad \delta(b, 1) &:= 2 \\ \delta(a, 2) &:= 2 & \qquad \delta(b, 2) &:= 2 \end{aligned}$$

Then $M := (Q, A, \delta, q_0, J)$ is a deterministic finite automaton over the alphabet A.

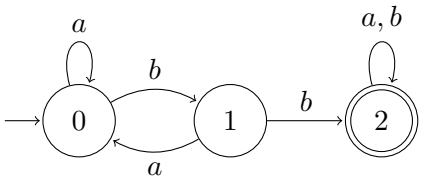


Figure 2.1: The transition diagram of the DFA M.

Example 2.1.12 Let $A := \{a, b\}$ be an alphabet with two letters. We take again the same set of states $Q := \{0, 1, 2\}$, now a set of initial states $I = \{0\}$, a set of final states $J := \{2\}$ and a transition function, where the letter b goes from the state 0 to state 0 and state 1 and without any transition from state 1 with letter a,

$$\delta : A \times Q \to \mathcal{P}(Q)$$

$$\begin{aligned} \delta(a, 0) &:= \{0\} & \qquad \delta(b, 0) &:= \{0, 1\} \\ \delta(a, 1) &:= \emptyset & \qquad \delta(b, 1) &:= \{2\} \\ \delta(a, 2) &:= \{2\} & \qquad \delta(b, 2) &:= \{2\} \end{aligned}$$

Then $N := (Q, A, \delta, I, J)$ is a nondeterministic finite automaton over the alphabet A.

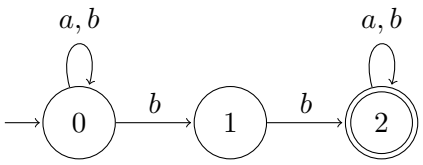


Figure 2.2: The transition diagram of the NFA N.

Definition 2.1.13 Let $M := (Q, A, \delta, q_0, J)$ be a deterministic finite automaton over A and let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton (or an ε-NFA) over A. Let $x \in A^*$ with $x = x_1 \dots x_n$.

1. The deterministic finite automaton M *accepts* x if and only if there exists a sequence $\langle r_i \in Q \mid i \in \{0, \dots, n\}\rangle$ such that
   (a) $r_0 = q_0$,
   (b) $\forall i \in \{1, \dots, n\} \quad \delta(x_i, r_{i-1}) = r_i$,
   (c) $r_n \in J$.
2. The nondeterministic finite automaton (or ε-NFA) N *accepts* x if and only if there exists a sequence $\langle r_i \in Q \mid i \in \{0, \dots, n\}\rangle$ such that
   (a) $r_0 \in I$,
   (b) $\forall i \in \{1, \dots, n\} \quad r_i \in \delta(x_i, r_{i-1})$,
   (c) $r_n \in J$.

[Pip97, 55][Sip13, 40]

Lemma 2.1.14 *Let $M := (Q, A, \delta, q_0, J)$ be a deterministic finite automaton over A and let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton (or an ε-NFA) over A. Let $x \in A^*$ with $x = x_1 \dots x_n$.*

1. *The string x is accepted by the DFA M if and only if $\widehat{\delta}(x, q_0) \in J$.*
2. *The string x is accepted by the NFA (or by the ε-NFA) N if and only if $\widehat{\delta}(x, I) \cap J \neq \emptyset$.*

Proof. 1. Assume the DFA M accept the word $x = x_1 \dots x_n$. Then there exists $r_0, \dots, r_n \in Q$ such that
   (a) $r_0 = q_0$
   (b) $\forall i \in \{1, \dots, n\} \quad \delta(x_i, r_{i-1}) = r_i$,
   (c) $r_n \in J$.

   holds. So the extended transition function applied step by step on $x_1 \dots x_n$ yields

$$\begin{aligned}
\widehat{\delta}(\varepsilon, q_0) &= r_0 \\
\widehat{\delta}(x_1, q_0) &= \delta(x_1, \widehat{\delta}(\varepsilon, r_0)) = \delta(x_1, q_0) = r_1 \\
\widehat{\delta}(x_1 x_2, q_0) &= \delta(x_2, \widehat{\delta}(x_1, q_0)) = \delta(x_2, r_1) = r_2 \\
&\vdots \\
\widehat{\delta}(x, q_0) &= \delta(x_n, \widehat{\delta}(x_1 \dots x_{n-1}, q_0)) = \delta(x_n, r_{n-1}) = r_n \in J.
\end{aligned}$$

   On the other hand assume $x = x_1 \dots x_n \in A^*$ with $\widehat{\delta}(x, q_0) \in J$. Then there exist states $r_0, \dots, r_n \in Q$ such that $q_0 = r_0$ and for all $i = 1, \dots, n$

$$\widehat{\delta}(x_1 \dots x_i, q_0) = \delta(x_i, \widehat{\delta}(x_1 \dots x_{i-1}, q_0)) = \delta(x_i, r_{i-1}) = r_i,$$

   as well as

$$\widehat{\delta}(x_1 \dots x_n, q_0) = r_n \in J.$$

   Therefore x is accepted by the DFA M.

2. Assume the NFA N accept the word $x = x_1 \dots x_n$. Then there exists $r_0, \dots, r_n \in Q$ such that the three conditions hold. Hence the extended transition function applied on x yields $\widehat{\delta}(x, I) \cap J \neq \emptyset$.
   For the opposite direction assume $x = x_1 \dots x_n \in A^*$ with $\widehat{\delta}(x, I) \cap J \neq \emptyset$. So there exists states $r_0, \dots, r_n \in Q$ such that $r_0 \in I$ and for all $i = 1, \dots, n$

$$r_i \in \widehat{\delta}(x_1 \dots x_i, I)$$

   as well as

$$r_n \in \widehat{\delta}(x_1 \dots x_n, I) \cap J.$$

   Hence x is accepted by the NFA N.

□

Definition 2.1.15 Let M be a deterministic finite automaton, nondeterministic finite automaton or ε-NFA over A. Then the *language* of M is defined as:

$$\text{Lang}(M) := \{x \in A^* \mid M \text{ accepts } x\}.$$

[Koz97, 17][Pip97, 55][Sip13, 40]

Definition 2.1.16 Let M be a deterministic finite automaton, nondeterministic finite automaton or ε-NFA over A and let L be a language over A. Then M *recognizes* L if and only if $L = \text{Lang}(M)$.

[Pip97, 55][Sip13, 40]

Definition 2.1.17 Let L be a language over A. Then L is a *recognizable* if there exists a nondeterministic finite automaton N over A that recognizes L.

[Pip97, 55][Sip13, 40]

Example 2.1.18 Let L be the language over the alphabet $A := \{a, b\}$ containing bb as a subword (A *subword* v of a word w consists of at least zero consecutive letters of the word w.)

$$L := \{w \in A^* \mid w \text{ contains } bb \text{ as a subword}\}$$

We show that the language L is recognizable. Since the NFA N from the example 2.1.12 reaches only its final state if there occur bb in the accepted words, it holds that $L = \text{Lang}(N)$. Hence the NFA N recognizes L and the language L is recognizable.

Furthermore, recalling the DFA M from the example 2.1.11, we can see that

$$\text{Lang}(M) = L.$$

Since the language L is recognized by the NFA N and the DFA M, we suggest that there might be a connection between these two different finite automata. Indeed, as the following theorem 2.1.19 proves.

Theorem 2.1.19 (Rabin, Scott, 1959) *Let L be a recognizable language over the alphabet A. Then there exists a deterministic finite automaton M such that $L = \mathrm{Lang}(M)$.*

[RS59][Pip97, 55]

Proof. Let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton recognizing L. We define the DFA M as follows:

$$\begin{aligned} M &:= (Q', A, \delta', q_0, J') \text{ with} \\ Q' &:= \mathcal{P}(Q) \\ q_0 &:= I \\ J' &:= \{K \subseteq Q \mid K \cap J \neq \emptyset\} \end{aligned}$$

and the transition function

$$\begin{aligned} \delta' : A \times \mathcal{P}(Q) &\to \mathcal{P}(Q) \\ \delta'(a, P) &:= \bigcup_{p \in P} \delta(a, p). \end{aligned}$$

We prove that $\mathrm{Lang}(M) = \mathrm{Lang}(N)$. We first show by induction over the length of the words that for $x \in A^*$ and $P \subseteq Q$ the following holds

$$\widehat{\delta}(x, P) = \widehat{\delta'}(x, P).$$

For $n = 0$ it follows by definition:

$$\widehat{\delta}(\varepsilon, P) = P = \widehat{\delta'}(\varepsilon, P).$$

For $n > 0$ we assume that it holds for words x of length n. Let $x \in A^n$ and $a \in A$. Then

$$\begin{aligned} \widehat{\delta}(xa, P) &= \delta(a, \widehat{\delta}(x, P)) && \text{definition of } \widehat{\delta} \\ &= \bigcup_{p \in \widehat{\delta}(x,P)} \delta(a, p) && \text{definition of } \delta \\ &= \bigcup_{p \in \widehat{\delta'}(x,P)} \delta(a, p) && \text{induction hypothesis} \\ &= \delta'(a, \widehat{\delta'}(x, P)) && \text{definition of } \delta' \\ &= \widehat{\delta'}(xa, P) && \text{definition of } \widehat{\delta'} \end{aligned}$$

Hence for $x \in A^*$ holds

$$x \in \mathrm{Lang}(N) \Leftrightarrow \widehat{\delta}(x, I) \cap J \neq \emptyset \Leftrightarrow \widehat{\delta'}(x, q_0') \in J' \Leftrightarrow x \in \mathrm{Lang}(M).$$

So the finite automata N and M accept the same words and thus $\mathrm{Lang}(N) = \mathrm{Lang}(M)$. □

[HMU01, 60ff][Koz97, 32ff][Pip97, 56]

Remark 2.1.20 The construction used in the previous proof is called *subset construction*.

Corollary 2.1.21 *Let N be a nondeterministic finite automaton over the alphabet A. Then there exists a deterministic finite automaton M over A such that* $\mathrm{Lang}(N) = \mathrm{Lang}(M)$.

Proof. This follows by Theorem 2.1.19. □

Example 2.1.22 We recall the NFA N from the example 2.1.12. Let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton with $A := \{a, b\}$, $Q := \{0, 1, 2\}$, $I = \{0\}$, $J := \{2\}$ and

$$\delta : A \times Q \to \mathcal{P}(Q)$$

$$\begin{aligned} \delta(a,0) &:= \{0\} & \qquad \delta(b,0) &:= \{0,1\} \\ \delta(a,1) &:= \emptyset & \qquad \delta(b,1) &:= \{2\} \\ \delta(a,2) &:= \{2\} & \qquad \delta(b,2) &:= \{2\} \end{aligned}$$

Then we obtain the following deterministic finite automaton $M := (\mathcal{P}(Q), A, \delta_1, q_0, J_1)$ by the subset construction. The set of states is

$$\mathcal{P}(Q) = \big\{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{1,2,3\}\big\},$$

the initial state and set of final states are

$$q_0 := \{0\} \qquad\qquad J_1 := \big\{\{2\}, \{0,2\}, \{1,2\}, \{1,2,3\}\big\}$$

and the new transition function $\delta_1 : A \times \mathcal{P}(Q) \to \mathcal{P}(Q)$ is

$$\begin{aligned} \delta_1(a,\{0\}) &:= \{0\} & \qquad \delta_1(b,\{0\}) &:= \{0,1\} \\ \delta_1(a,\{0,1\}) &:= \{0\} & \qquad \delta_1(b,\{0,1\}) &:= \{0,1,2\} \\ \delta_1(a,\{0,1,2\}) &:= \{0,2\} & \qquad \delta_1(b,\{0,1,2\}) &:= \{0,1,2\} \\ \delta_1(a,\{0,2\}) &:= \{0,2\} & \qquad \delta_1(b,\{0,2\}) &:= \{0,1,2\} \\ \delta_1(a,\{1\}) &:= \emptyset & \qquad \delta_1(b,\{1\}) &:= \{2\} \\ \delta_1(a,\{2\}) &:= \{2\} & \qquad \delta_1(b,\{2\}) &:= \{2\} \\ \delta_1(a,\{1,2\}) &:= \{2\} & \qquad \delta_1(b,\{1,2\}) &:= \{2\} \\ \delta_1(a,\emptyset) &:= \emptyset & \qquad \delta_1(b,\emptyset) &:= \emptyset. \end{aligned}$$

Currently, we haven't obtained the DFA from the example 2.1.11, we would have to merge the final states and ignore the inaccessible states.

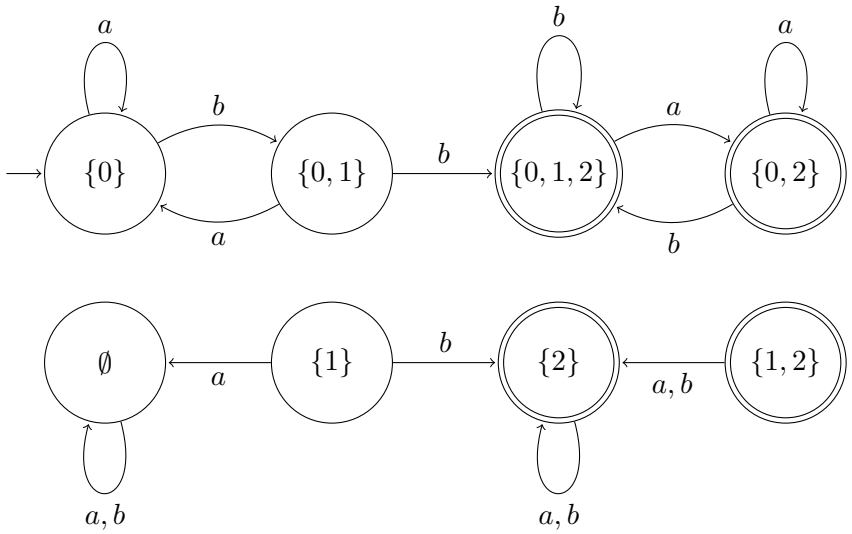


Figure 2.3: The transition diagram of the DFA M created with the subset construction.

Lemma 2.1.23 *Let M be a deterministic finite automaton over the alphabet A. Then there exists a nondeterministic finite automaton N over A such that* $\mathrm{Lang}(M) = \mathrm{Lang}(N)$.

Proof. Let $M := (Q, A, \delta, q_0, J)$ be a DFA over A. Then the following NFA N, defined as $N := (Q, A, \delta', \{q_0\}, J)$ with the transition function

$$\begin{aligned} \delta' : A \times Q &\to \mathcal{P}(Q) \\ \delta'(a, q) &:= \{\delta(a, q)\}, \end{aligned}$$

recognizes the same language as M. □

Definition 2.1.24 Let $N := (Q, A, \delta, I, J)$ be a nondeterministic finite automaton over the alphabet A. Define $\mathrm{Det}(N)$ as the deterministic finite automaton obtained by the subset construction with $\mathrm{Lang}(N) = \mathrm{Lang}(\mathrm{Det}(N))$.

Theorem 2.1.25 *Let N be a nondeterministic finite automaton. Then there exists an ε-NFA N_ε such that* $\mathrm{Lang}(N_\varepsilon) = \mathrm{Lang}(N)$.

Proof. In the NFA N are no ε-transitions, hence for each state $q \in Q$: $\textsc{eclose}(q) = q$ and therefore for each subset $P \subseteq Q$: $\textsc{eclose}(P) = P$.
We define $N_\varepsilon := (Q, A, \delta_\varepsilon, I, J)$ to be an ε-NFA with

$$\begin{aligned} \delta_\varepsilon : (A \cup \{\varepsilon\}) \times Q &\to \mathcal{P}(Q) \\ \delta_\varepsilon(\varepsilon, q) &:= \emptyset \\ \delta_\varepsilon(a, q) &:= \delta(a, q) \end{aligned}$$

The extended transition function $\widehat{\delta}_\varepsilon$ for $x \in A^*, a \in A$:

$$\begin{aligned} \widehat{\delta}_\varepsilon(\varepsilon, P) &= \textsc{eclose}(P) = P \\ \widehat{\delta}_\varepsilon(xa, P) &= \textsc{eclose}\Big(\bigcup_{p \in \widehat{\delta}_\varepsilon(x,P)} \delta_\varepsilon(a, p) \Big) \\ &= \bigcup_{p \in \widehat{\delta}_\varepsilon(x,P)} \delta_\varepsilon(a, p) \\ &= \bigcup_{p \in \widehat{\delta}_\varepsilon(x,P)} \delta(a, p) \end{aligned}$$

Hence the extended transition functions are equal and therefore

$$\mathrm{Lang}(N_\varepsilon) = \mathrm{Lang}(N).$$

□

Theorem 2.1.26 *Let N_ε be an ε-NFA . Then there exists a nondeterministic finite automaton N such that* $\mathrm{Lang}(N) = \mathrm{Lang}(N_\varepsilon)$.

Proof. Let $N_\varepsilon := (Q, A, \delta_\varepsilon, I, J_\varepsilon)$ be an ε-NFA. We define $N := (Q, A, \delta, I, J)$ to be a NFA with the final state set

$$J := \{q \in Q \mid \textsc{eclose}(q) \cap J_\varepsilon \neq \emptyset\}$$

and the transition function

$$\begin{aligned}\delta : A \times Q &\to \mathcal{P}(Q)\\ \delta(a,q) &:= \bigcup_{p \in \text{ECLOSE}(q)} \delta_\varepsilon(a,p).\end{aligned}$$

Note that N accepts the word $w \in A^*$ if $\text{ECLOSE}(\widehat{\delta}(x,I)) \cap J \neq \emptyset$ and N_ε accepts the word $w \in A^*$ if $\widehat{\delta}_\varepsilon(x,I) \cap J \neq \emptyset$.
We show by induction on the length of the words $w \in A^*$ that the following holds:

$$\text{ECLOSE}(\widehat{\delta}(w,I)) = \widehat{\delta}_\varepsilon(w,I).$$

For the word ε of length 0 it holds that

$$\text{ECLOSE}(\widehat{\delta}(\varepsilon,I)) = \text{ECLOSE}(I) = \widehat{\delta}_\varepsilon(\varepsilon,I).$$

Assume it holds for the words of length n.
Let $x \in A^n$ and $a \in A$. Then:

$$\begin{aligned}\text{ECLOSE}(\widehat{\delta}(xa,I)) &= \text{ECLOSE}(\delta(a,\widehat{\delta}(x,I))\\ &= \text{ECLOSE}\left(\bigcup\{\delta_\varepsilon(a,p) \mid p \in \text{ECLOSE}(\widehat{\delta}(x,I))\}\right)\\ &= \text{ECLOSE}\left(\bigcup\{\delta_\varepsilon(a,p) \mid p \in \widehat{\delta}_\varepsilon(x,I)\}\right)\\ &= \bigcup\{\text{ECLOSE}(\delta_\varepsilon(a,p)) \mid p \in \widehat{\delta}_\varepsilon(x,I)\}\\ &= \widehat{\delta}_\varepsilon(xa,I).\end{aligned}$$

Therefore

$$\text{Lang}(N_\varepsilon) = \text{Lang}(N).$$

□

[HMU01, 77ff]

2.2 Closure properties of recognizable languages

The following chapter covers the closure properties of recognizable languages like intersection, union, complement or homomorphism on the alphabet. A given closure property of a recognizable language is usually proven by giving a nondeterministic finite automaton recognizing the language (or due to the subset construction a deterministic finite automaton) and modify it such that it recognizes the language of the closure property.

Definition 2.2.1 Let $x = x_1 \dots x_n$ be a word over the alphabet A.
Then the reversal of the word x is

$$\mathrm{Rev}(x) := x_n \dots x_1.$$

Let L be a language over the alphabet A. Then the *reversal* of L is

$$\mathrm{Rev}(L) := \{\mathrm{Rev}(x) \mid x \in L\}.$$

Lemma 2.2.2 *Let L be a recognizable language. Then* $\mathrm{Rev}(L)$ *is recognizable.*

Proof. Let $N := (Q, A, \delta, I, J)$ be a NFA recognizing L. Then $N' := (Q, A, \delta', J, I)$ with the transition function

$$\begin{aligned} \delta' : A \times Q &\to \mathcal{P}(Q) \\ \delta'(x, q) &:= \{q' \in Q \mid q \in \delta(x, q')\}. \end{aligned}$$

is a NFA recognizing $\mathrm{Rev}(L)$. □

Note that the idea of previous Lemma only works on nondeterministic finite automata. Reversing the direction of the state transition in a deterministic finite automaton may change it to a nondeterministic one. [Pip97, 57]

Definition 2.2.3 Let L be a language over the alphabet A. The *complement* of the language L is

$$\mathrm{Cpl}(L) := A^* \setminus L.$$

Lemma 2.2.4 *Let L be a recognizable language over the alphabet A. Then* $\mathrm{Cpl}(L)$ *is recognizable.*

Proof. Let $M := (Q, A, \delta, q_0, J)$ be a DFA recognizing L. Then $M' := (Q, A, \delta, q_0, Q \setminus J)$ is a DFA recognizing $\mathrm{Cpl}(L)$. □

[Pip97, 58]

Definition 2.2.5 Let $M := (Q, A, \delta, q_0, J)$ be a deterministic finite automaton over the alphabet A. Define

$$\mathrm{Cpl}(M) := (Q, A, \delta, q_0, Q \setminus J)$$

as the *complement deterministic finite automaton* with $\mathrm{Lang}(\mathrm{Cpl}(M)) = \mathrm{Cpl}(\mathrm{Lang}(M))$.

Lemma 2.2.6 *Let L_1 and L_2 be recognizable languages over the alphabet A. Then*

1. *$L_1 \cap L_2$ is recognizable.*
2. *$L_1 \cup L_2$ is recognizable.*

Proof. 1. Let $M_1 := (Q_1, A, \delta_1, p_0, J_1)$ be a deterministic finite automaton recognizing L_1 and let $M_2 := (Q_2, A, \delta_2, q_0, J_2)$ be a deterministic finite automaton recognizing L_2. Then $M := (Q_1 \times Q_2, A, \delta, (p_0, q_0), J_1 \times J_2)$ with the transition function

$$\begin{aligned} \delta : A \times (Q_1 \times Q_2) &\to Q_1 \times Q_2 \\ \delta(x, (q_1, q_2)) &:= (\delta_1(x, q_1), \delta_2(x, q_2)) \end{aligned}$$

is a deterministic finite automaton recognizing $L_1 \cap L_2$.

2. Let $M_1 := (Q_1, A, \delta_1, p_0, J_1)$ be a deterministic finite automaton recognizing L_1 and let $M_2 := (Q_2, A, \delta_2, q_0, J_2)$ be a deterministic finite automaton recognizing L_2 with different sets of states. Then $N := (Q_1 \cup Q_2, A, \delta, \{p_0, q_0\}, J_1 \cup J_2)$ with the transition function

$$\begin{aligned} \delta : A \times (Q_1 \cup Q_2) &\to \mathcal{P}(Q_1 \cup Q_2) \\ \delta(x, q) &:= \begin{cases} \{\delta_1(x, q)\} & \text{if } q \in Q_1 \\ \{\delta_2(x, q)\} & \text{if } q \in Q_2 \end{cases} \end{aligned}$$

is a NFA recognizing $L_1 \cup L_2$.

□

The previous proof for the recognizability of $L_1 \cup L_2$ transforms two deterministic finite automata into a nondeterministic one by having two initial states.
Alternatively, $L_1 \cup L_2 = \mathrm{Cpl}(\mathrm{Cpl}(L_1) \cap \mathrm{Cpl}(L_2))$ uses the Lemma 2.2.4 and 2.2.6(1).
[Pip97, 58]

Definition 2.2.7 Let $M_1 := (Q_1, A, \delta_1, p_0, J_1)$ and $M_2 := (Q_2, A, \delta_2, q_0, J_2)$ be deterministic finite automata over the alphabet A.

1. Define $M_1 \cap M_2 := (Q_1 \times Q_2, A, \delta, (p_0, q_0), J_1 \times J_2)$ with the transition function

$$\begin{aligned} \delta : A \times (Q_1 \times Q_2) &\to Q_1 \times Q_2 \\ \delta(x, (q_1, q_2)) &:= (\delta_1(x, q_1), \delta_2(x, q_2)). \end{aligned}$$

as the *intersection of two deterministic finite automata* with

$$\mathrm{Lang}(M_1 \cap M_2) = \mathrm{Lang}(M_1) \cap \mathrm{Lang}(M_2).$$

2. Define $M_1 \cup M_2 := \mathrm{Cpl}\big(\mathrm{Cpl}(M_1) \cap \mathrm{Cpl}(M_2)\big)$ as the *union of two deterministic finite automata* with

$$\mathrm{Lang}(M_1 \cup M_2) = \mathrm{Lang}(M_1) \cup \mathrm{Lang}(M_2).$$

Lemma 2.2.8 *Let L be a language over the alphabet A. $L \cup \{\varepsilon\}$ is recognizable if and only if $L \setminus \{\varepsilon\}$ is recognizable.*

Proof. Assume $L \cup \{\varepsilon\}$ is recognizable.
Case 1: $\varepsilon \in L$. Then $L = L \cup \{\varepsilon\}$. Let $M := (Q, A, \delta, q_0, J)$ be a DFA recognizing L. Let q_0' be a new state and we define a new transition function as follows:

$$\begin{aligned} \delta' : A \times Q &\to Q \\ \delta'(x, q) &:= \begin{cases} \{\delta_1(x, q_0)\} & \text{if } q = q_0' \\ \{\delta_2(x, q)\} & \text{otherwise.} \end{cases} \end{aligned}$$

Then $M' := (Q, A, \delta', q_0', J)$ is a DFA recognizing $L \setminus \{\varepsilon\}$.

Case 2: $\varepsilon \notin L$. Since $L \cup \{\varepsilon\}$ is recognizable the proof follows with same construction as above.
For the other direction assume $L \setminus \{\varepsilon\}$ is recognizable. The DFA with one state, which is an initial and a final state, and no state transitions recognizes $\{\varepsilon\}$. Therefore with Lemma 2.2.6 (2) follows that $L \setminus \{\varepsilon\} \cup \{\varepsilon\}$ is recognizable. □

[Pip97, 58]

Definition 2.2.9 Let L_1 and L_2 be languages over the alphabet A. Then the product of the languages L_1 and L_2 is

$$L_1 \cdot L_2 := \{x \cdot y \mid x \in L_1, y \in L_2\}.$$

Lemma 2.2.10 *Let L_1 and L_2 be recognizable languages over the alphabet A. Then $L_1 \cdot L_2$ is recognizable.*

Proof. We write $L_1 := L_1^0 \cup L_1^+$, where $L_1^+ \subseteq A^+$ and $L_1^0 \subseteq \{\varepsilon\}$ (either the language contains only the empty word or the language is the empty set). Similarly, we define $L_2 := L_2^0 \cup L_2^+$. Then

$$L_1 \cdot L_2 = L_1^0 \cdot L_2^0 \cup L_1^0 \cdot L_2^+ \cup L_1^+ \cdot L_2^0 \cup L_1^+ \cdot L_2^+.$$

The languages L_1^0 and L_2^0 are recognizable and with the previous Lemma 2.2.8 the languages L_1^+ and L_2^+ are also recognizable. Let $N_1 := (Q_1, A, \delta_1, p_0, J_1)$ be a NFA recognizing L_1^+ and $N_2 := (Q_2, A, \delta_2, q_0, J_2)$ be a NFA recognizing L_2^+ with $Q_1 \cap Q_2 = \emptyset$.
The first term $L_1^0 \cdot L_2^0$ is recognizable, since the NFA $(\{q_0\}, A, \delta, \{q_0\}, J)$, with δ as empty function and $J = \{q_0\}$ if $\varepsilon \in L_1^0$ or $\varepsilon \in L_2^0$ (else $J = \emptyset$), recognizes the language. The second term $L_1^0 \cdot L_2^+$ is recognizable, since N_2 recognizes L_2^+ and L_1^0 turns the initial state of N_2 into a final state if $\varepsilon \in L_1^0$. The third term $L_1^+ \cdot L_2^0$ is recognizable, since N_1 recognizes L_1^+ and neither $L_2^0 = \{\varepsilon\}$ nor $L_2^0 = \emptyset$ changes N_1. The fourth term $L_1^+ \cdot L_2^+$ requires a combination of the automata N_1 and N_2. We define a new transition function

$$\delta : A \times (Q_1 \cup Q_2) \to \mathcal{P}(Q_1 \cup Q_2)$$
$$\delta(x,q) := \begin{cases} \{\delta_1(x,q)\} & \text{if } q \in Q_1 \setminus J_1 \\ \{\delta_1(x,q)\} \cup \{\delta_2(x,q_0)\} & \text{if } q \in J_1 \\ \{\delta_2(x,q)\} & \text{if } q \in Q_2 \end{cases}$$

Then the nondeterministic finite automaton $N := (Q_1 \cup Q_2, A, \delta, p_0, J_2)$ recognizes $L_1^+ \cdot L_2^+$. □

[Pip97, 59]

Definition 2.2.11 Let L be a language over the alphabet A. Then for $n \in \mathbb{N}_0$ L^n is defined recursively:

$$L^0 := \{\varepsilon\}, L^1 := L \text{ and } L^{n+1} := L \cdot L^n \text{ for } n \geq 1.$$

L^+ and L^* are defined as follows:

$$L^+ := \bigcup_{n \geq 1} L^n \qquad\qquad L^* := \bigcup_{n \geq 0} L^n.$$

Lemma 2.2.12 *Let L be a recognizable language over the alphabet A. Then L^+ and L^* are recognizable.*

Proof. Let L be a recognizable language over the alphabet A such that $L \subseteq A^+$ (due to lemma 2.2.8 the empty word ε doesn't restrict the recognizability). Let $M := (Q, A, \delta, q_0, J)$ be a DFA recognizing L. The new transition function is defined as

$$\delta' : A \times Q \to \mathcal{P}(Q)$$
$$\delta'(x,q) := \begin{cases} \{\delta(x,q)\} & \text{if } q \in Q \setminus J \\ \{\delta(x,q)\} \cup \{\delta(x,q_0)\} & \text{if } q \in J, \end{cases}$$

such that the transition form the initial state are also reachable from all final states. Then the NFA $N := (Q, A, \delta', \{q_0\}, J)$ recognizes L^*. Since $L^+ = L^* \setminus \{\varepsilon\}$ the recognizability for L^+ follows from Lemma 2.2.8. □

[Pip97, 59]

Definition 2.2.13 Let L_1 and L_2 be languages over the alphabet A. The *right quotient* of L_1 with L_2 is

$$L_1 / L_2 := \{w \in A \mid \exists x \in L_2 \text{ s.t. } wx \in L_1\}.$$

and the *left quotient* of L_1 with L_2 is

$$L_2 \backslash L_1 := \{w \in A \mid \exists x \in L_2 \text{ s.t. } xw \in L_1\}.$$

Lemma 2.2.14 *Let L_1 and L_2 be recognizable languages over the alphabet A. Then L_1 / L_2 is recognizable.*

Proof. Let $M := (Q, A, \delta, q_0, J)$ be a DFA with $Q := \{q_1, \ldots, q_k\}$ recognizing L_1. Set $J' := J$. For every $1 \leq i \leq k$ take $q_i \in Q$ and construct the DFA $M_i := (Q, A, \delta, q_i, J)$ with the initial state q_i. If there exists $x \in L_2$ such that M_i accepts x (i.e. there is a path from the initial state q_i to a final state) then add the state q_i to J'. After repeating this for all i, the DFA $M' := (Q, A, \delta, q_0, J')$, with the new final state set J' has the language $\mathrm{Lang}(M')$.

Claim: $\mathrm{Lang}(M') = L_1 / L_2$.

"$\subseteq$" Assume $x \in \mathrm{Lang}(M')$. Then there exists $q \in J'$ s.t. $\widehat{\delta}(x, q_0) = q$. By construction there exists $y \in L_2$ such that $\widehat{\delta}(y, q) \in J$. Therefore $xy \in L_1$ and $x \in L_1 / L_2$.

"$\supseteq$" Assume $x := x_1 \ldots x_n \in L_1 / L_2$. Then there exists $y := y_1 \ldots y_m \in L_2$ such that $xy \in L_1$. Therefore there exists a state $q \in Q$ s.t. $\widehat{\delta}(x, q_0) = q$ and $\widehat{\delta}(y, q) \in J$. Therefore by construction $q \in J'$.

□

[Lin12, 119ff]

Corollary 2.2.15 *Let L_1 and L_2 be recognizable languages over the alphabet A. Then $L_2 \backslash L_1$ is recognizable.*

Proof. Follows with Lemma 2.2.14, Lemma 2.2.2 and $L_2 \backslash L_1 = \mathrm{Rev}(\mathrm{Rev}(L_1) / \mathrm{Rev}(L_2))$. □

Definition 2.2.16 Let A and B be finite alphabets. The mapping

$$h : (A^*, \cdot) \to (B^*, \cdot)$$

is a *homomorphism* on A if for all $a_1, a_2 \in A^*$ the following holds

$$h(a_1 \cdot a_2) = h(a_1) \cdot h(a_2)$$
$$h(\varepsilon) = \varepsilon.$$

Let L be a language over the alphabet A. The homomorphic image $h(L)$ and the homomorphic preimage $h^{-1}(L)$ are defined as follows:

$$h(L) := \{h(w) \mid w \in L\}$$
$$h^{-1}(L) := \{w \in A^* \mid h(w) \in L\}$$

[HMU01, 139][Koz97, 61]

Lemma 2.2.17 *Let A and B be finite alphabets and let $h : A^* \to B^*$ be a homomorphism. Let L be a recognizable language over A. Then the homomorphic image $h(L)$ is a recognizable language over B.*

[Koz97, 61][HMU01, 139]

Proof. Let $M := (Q, A, \delta, q_1, J)$ be the DFA recognizing L with the states $Q := \{q_1, \ldots, q_m\}$ and the letters $A := \{a_1, \ldots a_n\}$. For each pair $(a, q) \in A \times Q$ we will introduce $|h(a)| - 1$ new intermediate states (no new states for $h(a) = \varepsilon$ and $|h(a)| = 1$). First we define the function

$$l : \{1, \ldots, n\} \to \mathbb{Z}$$
$$l(i) := |h(a_i)| - 1.$$

For $1 \leq i \leq n$ and $1 \leq j \leq m$ we define for the letter $a_i \in A$ and the state $q_j \in Q$ the set of intermediate states

$$P_{i,j} := \begin{cases} \{p_{i,j,1}, \ldots, p_{i,j,l(i)}\} & \text{if } l(i) \geq 1 \\ \emptyset & \text{otherwise} \end{cases}$$

and the set of all intermediate states

$$P := \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} P_{i,j}.$$

For each pair a_i and q_j our goal is to create new transition for all letters in the word $h(a_i)$ (for the case $|h(a)| \geq 1$). We start in state q_j, go to the first intermediate state $p_{i,j,1}$ with $h(a_i)_1$ (the first letter of the word $h(a_i)$), to the second intermediate state $p_{i,j,2}$ with $h(a_i)_2$, ... and finally going from the state $p_{i,j,l(i)}$ to the state $\delta(a_i, q_j) \in Q$. In the case of $h(a)$ being a word of length zero or one, the transitions are renamed and no new intermediate states are introduced. Formally, we define a new transition function

$$\delta_\varepsilon : B \times (Q \cup P) \to (Q \cup P)$$
$$\delta_\varepsilon(b, q) := \begin{cases} \{\delta(a_i, q) \mid h(a_i) = b\} \cup \{p_{i,j,1} \mid \exists w \in A^+ : h(a_i) = bw\} & \text{if } q = q_j \\ \{p_{i,j,k}\} & \text{if } h(a_i)_k = b,\ q = p_{i,j,k-1} \text{ and } 1 \leq k \leq l(i) \\ \{q_j\} & \text{if } h(a_i)_k = b \text{ and } q = p_{i,j,l(i)} \\ \emptyset & \text{otherwise.} \end{cases}$$

The first row of the transition function returns for $q \in Q$ a state in Q if $|h(a)| = 1$ or $|h(a)| = |\varepsilon| = 0$ and for $|h(a)| \geq$ it returns the first intermediate state. The second row computes the transitions between the intermediate states and the third row returns from an intermediate state to the according state in Q. Then the ε-NFA $N := (Q \cup P, B, \delta_\varepsilon, I, J)$ recognizes $h(L)$. □

Corollary 2.2.18 *Let A and B be finite alphabets and let $h : A^* \to B^*$ be a homomorphism with $|h(a)| = 1$ for all $a \in A$. Let L be a recognizable language over A. The homomorphic image $h(L)$ is a recognizable language over B.*

Proof. Let $M := (Q, A, \delta, q_0, J)$ be the DFA recognizing L. The NFA $N := (Q, B, \delta', \{q_0\}, J)$ with

$$\begin{aligned}\delta' : B \times Q &\to \mathcal{P}(Q)\\ \delta'(b, q) &:= \{\delta(a, q) \mid h(a) = b\}\end{aligned}$$

recognizes $h(L)$. Note that for the finite automaton nondeterministic is necessary, since the homomorphism might not be injective. □

Definition 2.2.19 Let A and B be finite alphabets and let $h : A^* \to B^*$ be a homomorphism with $|h(a)| = 1$ for all $a \in A$. Let $M := (Q, A, \delta, q_0, J)$ be a DFA over A.
Define $h(M) := (Q, B, \delta', I, J)$ with

$$\begin{aligned}\delta' : B \times Q &\to \mathcal{P}(Q)\\ \delta'(h(a), q) &:= \{\delta(a, q) \mid h(a) = b\}\end{aligned}$$

as the NFA over the alphabet B with $\mathrm{Lang}(h(M))) = h(\mathrm{Lang}(M))$.

Lemma 2.2.20 *Let A and B be finite alphabets and let $h : A^* \to B^*$ be a homomorphism. Let L be a recognizable language over B. Then $h^{-1}(L)$ is a recognizable language over A.*

Proof. Let $M := (Q, B, \delta, q_0, J)$ be the DFA recognizing L. Then the DFA $N := (Q, A, \delta', q_0, J)$ with

$$\begin{aligned}\delta' : A \times Q &\to Q\\ \delta'(a, q) &:= \widehat{\delta}(h(a), q)\end{aligned}$$

recognizes $h^{-1}(L)$. □

Corollary 2.2.21 *Let A and B be finite alphabets and let $h : A^* \to B^*$ be a homomorphism with $|h(a)| = 1$ for all $a \in A$. Let L be a recognizable language over B. Then $h^{-1}(L)$ is a recognizable language over A.*

Proof. Let $M := (Q, B, \delta, q_0, J)$ be the DFA recognizing L. Then the DFA $N := (Q, A, \delta', q_0, J)$ with

$$\begin{aligned}\delta' : A \times Q &\to Q\\ \delta'(a, q) &:= \delta(h(a), q)\end{aligned}$$

recognizes $h^{-1}(L)$. □

[HMU01, 142f][Koz97, 62]

Definition 2.2.22 Let A and B be finite alphabets and let $h : A^* \to B^*$ be a homomorphism with $|h(a)| = 1$ for all $a \in A$. Let $M := (Q, B, \delta, q_0, J)$ be a DFA over B.
Define $h^{-1}(M) := (Q, A, \delta', I, J)$ with

$$\begin{aligned}\delta' : A \times Q &\to Q\\ \delta'(a, q) &:= \delta(h(a), q)\end{aligned}$$

as the deterministic finite automaton over the alphabet A with $\mathrm{Lang}(h^{-1}(M)) = h^{-1}(\mathrm{Lang}(M))$.

2.3 Minimization of deterministic finite automata

Definition 2.3.1 Let $M_1 := (Q_1, A, \delta_1, q_o, J_1)$ and $M_2 := (Q_2, A, \delta_2, p_0, J_2)$ be deterministic finite automata. A map f from Q_1 to Q_2 *preserves* the structure of M_1 for M_2, if for all $q \in Q_1$ the following holds:

$$\begin{aligned} f(q_0) &= p_0 \\ f(\delta_1(a,q)) &= \delta_2(a, f(q)) \\ q \in J &\Rightarrow f(q) \in J. \end{aligned}$$

Definition 2.3.2 Let $M = (Q, A, \delta, q_o, J)$ be a deterministic finite automaton. The *accessible states* of M are

$$\mathrm{Acc}(Q) := \{\widehat{\delta}(x, q_0) \mid x \in A^*\}.$$

The *accessible automaton* of M, $\mathrm{Acc}(M)$, is defined as follows:

$$\mathrm{Acc}(M) := (\mathrm{Acc}(Q), A, \delta \restriction_{A \times \mathrm{Acc}(Q)}, q_0, J \cap \mathrm{Acc}(Q)).$$

Definition 2.3.3 Let $M = (Q, A, \delta, q_0, J)$ be a deterministic finite automaton and let $q_1, q_2 \in Q$. Then the *indistinguishable* states are defined as

$$q_1 \sim q_2 :\Leftrightarrow \forall x \in A^* : \ \widehat{\delta}(x, q_1) \in J \Leftrightarrow \widehat{\delta}(x, q_2) \in J,$$

otherwise they are called *distinguishable*. The *distinguishable automaton* $\mathrm{Dist}(M)$ is defined as follows:

$$\mathrm{Dist}(M) := (Q/\!\sim, A, \delta', q_0/\!\sim, \{j/\!\sim \mid j \in J\})$$

with the transition function

$$\begin{aligned} \delta' : A \times Q/\!\sim &\to Q/\!\sim \\ \delta'(a, q/\!\sim) &:= \delta(a,q)/\!\sim. \end{aligned}$$

[Koz97, 80f][Pip97, 61]

Definition 2.3.4 Let A be an alphabet and L be a language over A such that $\{x^{-1}L \mid x \in A\}$ is finite, where

$$x^{-1}L := \{y \in A^* \mid xy \in L\}.$$

Let $x, y \in A^*$. Then we define the following relation

$$x \simeq y :\Leftrightarrow x^{-1}L = y^{-1}L$$

and the following deterministic finite automaton

$$\begin{aligned} \mathrm{Min}(L) &:= (Q, A, \delta, q_0, J) \text{ with} \\ Q &:= \{x/\!\simeq \mid x \in A^*\} \\ \delta(a, x/\!\simeq) &:= (xa)/\!\simeq \\ q_0 &:= \varepsilon/\!\simeq \\ J &:= \{x/\!\simeq \mid x \in L\} \end{aligned}$$

Lemma 2.3.5 *Let L be a language over the alphabet A. Then* $\mathrm{Min}(L)$ *recognizes L.*

Proof. We show that the following holds:

$$\mathrm{Lang}(\mathrm{Min}(L)) = L.$$

In order to show $\mathrm{Lang}(\mathrm{Min}(L)) \subseteq L$, let

$$\mathrm{Min}(L) := (Q, A, \delta, q_0, J)$$

and $x \in \mathrm{Lang}(\mathrm{Min}(L))$. So x is an accepted word of $\mathrm{Min}(L)$ and therefore

$$\widehat{\delta}(x, \varepsilon/\simeq) \in J.$$

Hence there is $y \in L$ such that

$$\widehat{\delta}(x, \varepsilon/\simeq) = y/\simeq$$

and so $x/\simeq = y/\simeq$. Therefore $x \simeq y$ and since $y \in L$ and $y\varepsilon \in L$, also $\varepsilon \in y^{-1}L$. But then since $x \simeq y$ $\varepsilon \in x^{-1}L$ and $z \in L$.
For the opposite direction let $x \in L$. Then

$$\widehat{\delta}(x, \varepsilon/\simeq) = x/\simeq \ \in J,$$

since $x \in L$. Therefore $x \in \mathrm{Lang}(\mathrm{Min}(L))$. □

Theorem 2.3.6 *Let $M := (Q, A, \delta, q_0, J)$ be a deterministic finite automaton over the alphabet A and let $L :=$* $\mathrm{Lang}(M)$. *Let*

$$\mathrm{Dist}(\mathrm{Acc}(M)) := (Q_1, A, \delta_1, q_0/\sim, J_1)$$
$$\mathrm{Min}(L) := (Q_2, A, \delta_2, \varepsilon/\simeq, J_2)$$

be a deterministic finite automaton. Let φ be a map

$$\varphi : Q_1 \to Q_2$$
$$\widehat{\delta}(x, q_0)/\sim \ \mapsto x/\simeq$$

Then φ is a bijection that preserves the structure of $\mathrm{Dist}(\mathrm{Acc}(M))$ *for* $\mathrm{Min}(L)$.

Proof. φ **is well-defined.** Let $x, y \in A^*$ such that

$$\widehat{\delta}(x, q_0) \sim \widehat{\delta}(y, q_0).$$

We show that $x/\simeq \ = y/\simeq$, i.e. $x^{-1}L = y^{-1}L$. Let $z \in x^{-1}L$. Then $xz \in L$ and hence $\widehat{\delta}(xz, q_0) \in J$. Therefore

$$\widehat{\delta}(z, \widehat{\delta}(x, q_0)) \in J, \text{ hence}$$
$$\widehat{\delta}(z, \widehat{\delta}(y, q_0)) \in J \text{ and}$$
$$\widehat{\delta}(yz, q_0) \in J.$$

Thus $yz \in L$ and $x^{-1}L \subseteq y^{-1}L$. The other direction follows similarly.
φ **is surjective.** This follows from the definition.
φ **is injective.** Let $x, y \in A^*$ such that $x/\simeq \ = y/\simeq$. We show that

$$\widehat{\delta}(x, q_0) \sim \widehat{\delta}(y, q_0).$$

Let $z \in A^*$ such that

$$\widehat{\delta}(z, \widehat{\delta}(x, q_0)) \in J.$$

Then $xz \in L$ and since $x/\simeq \; = y/\simeq$ and $x^{-1}L \subseteq y^{-1}L$, also $yz \in L$. Thus

$$\widehat{\delta}(z, \widehat{\delta}(y, q_0)) \in J \text{ and}$$
$$\widehat{\delta}(yz, q_0) \in J.$$

φ **preserves** the structure of $\mathrm{Dist}(\mathrm{Acc}(M))$.

$$\varphi(q_0/\sim) = \varphi(\widehat{\delta}(\varepsilon, q_0)/\sim) = \varepsilon/\simeq$$

and

$$\begin{aligned}\varphi(\delta_1(a, q/\sim)) &= \varphi(\delta(a, \widehat{\delta}(x, q_0))/\sim) \\ &= \varphi(\widehat{\delta}(xa, q_0)/\sim) \\ &= xa/\simeq \\ &= \delta_2(a, x/\simeq)\end{aligned}$$

Therefore also the expanded transitions function is preserved under φ and hence the same words are accepted. □

[Pip97, 61f]

Definition 2.3.7 Let $M_1 := (Q_1, A, \delta_1, q_0, J_1)$ and $M_2 := (Q_2, A, \delta_2, p_0, J_2)$ be a deterministic finite automaton. Then M_1 and M_2 are *equivalent* if there exists an bijection $f : Q_1 \to Q_2$ such that the following holds:

1. $f(q_0) = p_0$,
2. $f(\delta_1(a, q)) = \delta(a, f(q))$ for all $a \in A$ and $q \in Q_1$ and
3. $q \in J_1$ if and only if $f(q) \in J_2$.

Definition 2.3.8 Let M be a deterministic finite automaton. Define the *minimal deterministic finite automaton* of M as $\mathrm{Min}(M) := \mathrm{Dist}(\mathrm{Acc}(M))$ with $\mathrm{Lang}(\mathrm{Min}(M)) = \mathrm{Lang}(M)$.

Remark 2.3.9 Note that a minimal deterministic finite automaton is unique up to renaming of the states.

Example 2.3.10 We recall the DFA M from the example 2.1.22. We create the accessible deterministic finite automaton over the alphabet $A := \{a, b\}$

$$\mathrm{Acc}(M) = (\mathrm{Acc}(Q), A, \delta_1 \restriction A \times \mathrm{Acc}(Q), q_0, J_1 \cap \mathrm{Acc}(J))$$

with the accessible states

$$\mathrm{Acc}(Q) = \big\{\{0\}, \{0,1\}, \{0,2\}, \{0,1,2\}\big\},$$

the accessible final states

$$J_2 := J_1 \cap \mathrm{Acc}(J) = \big\{\{0,2\}, \{0,1,2\}\big\}$$

and transition function $\delta'' := \delta' \restriction_{A \times \mathrm{Acc}(Q)}$:

$$\begin{aligned}
&\delta_2(a,\{0\}) := \{0\} && \delta_2(b,\{0\}) := \{0,1\}\\
&\delta_2(a,\{0,1\}) := \{0\} && \delta_2(b,\{0,1\}) := \{0,1,2\}\\
&\delta_2(a,\{0,1,2\}) := \{0,2\} && \delta_2(b,\{0,1,2\}) := \{0,1,2\}\\
&\delta_2(a,\{0,2\}) := \{0,2\} && \delta_2(b,\{0,2\}) := \{0,1,2\}.
\end{aligned}$$

The only indistinguishable states of $\mathrm{Acc}(M)$ are the states $\{0,2\}$ and $\{0,1,2\}$ since it holds that for all words $w \in A^*$ $\widehat{\delta_2}(w,\{0,2\}) \in J_2$ if and only if $\widehat{\delta_2}(w,\{0,1,2\}) \in J_2$. Furthermore, the states $\{0\}$ and $\{0,1\}$ are distinguishable since the word b reaches a final state from state $\{0,1\}$ but not from state $\{0\}$. Note that final and non-final states are distinguishable due to the empty word ε.

Hence the equivalence relation is

$$\begin{aligned}
\sim := \Big\{&\big(\{0\},\{0\}\big),\big(\{0,1\},\{0,1\}\big),\big(\{0,2\},\{0,2\}\big),\big(\{0,2\},\{0,1,2\}\big),\\
&\big(\{0,1,2\},\{0,2\}\big),\big(\{0,1,2\},\{0,1,2\}\big)\Big\}
\end{aligned}$$

and we get

$$\mathrm{Min}(M) := \mathrm{Dist}(\mathrm{Acc}(M)) = \big(\big\{\{0\}/\!\sim,\{0,1\}/\!\sim,\{0,1,2\}/\!\sim\big\},A,\delta_3,\{0\}/\!\sim,\big\{\{0,1,2\}/\!\sim\big\}\big)$$

as a minimal deterministic finite automaton of M with the transition function:

$$\begin{aligned}
&\delta_3(a,\{0\}/\!\sim) := \{0\}/\!\sim && \delta_3(b,\{0\}/\!\sim) := \{0,1\}/\!\sim\\
&\delta_3(a,\{0,1\}/\!\sim) := \{0\}/\!\sim && \delta_3(b,\{0,1\}/\!\sim) := \{0,1,2\}/\!\sim\\
&\delta_3(a,\{0,1,2\}/\!\sim) := \{0,1,2\}/\!\sim && \delta_3(b,\{0,1,2\}/\!\sim) := \{0,1,2\}/\!\sim
\end{aligned}$$

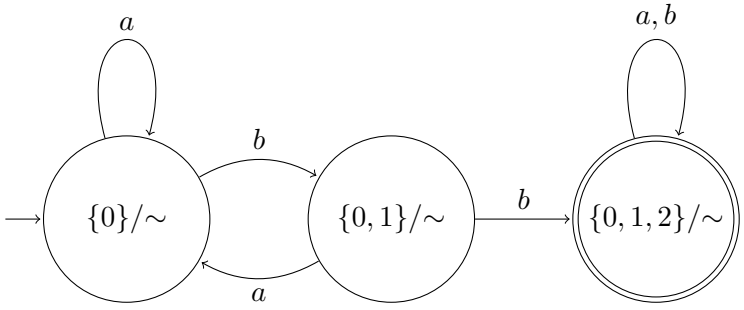


Figure 2.4: The transition diagram of DFA $\mathrm{Dist}(\mathrm{Acc}(M))$.

2.4 Regular expressions

Definition 2.4.1 Let A be an alphabet. Then a *regular expression* over A is defined recursively as follows:
Primitive regular expressions:

1. O is a regular expression.
2. I is a regular expression.
3. Let $a \in A$, $\mathbf{a}$ is a regular expression.

Let F and G be regular expressions, then:

4. $(F \cup G)$ is a regular expression.
5. $(F \cdot G)$ is a regular expression.
6. F^* is a regular expression.
7. (F) is a regular expression.

The parenthesis in a regular expression might be omitted, where the precedence of the asterisk (6) is higher than of the concatenation (5) and the precedence of the concatenation is higher than of the union (4). [HMU01, 86ff][Pip97, 64]

Definition 2.4.2 Let A be an alphabet. Let E be a regular expression over A. Then the *regular language* of E, $\mathrm{Lang}(E)$, is defined recursively as follows:

1. $\mathrm{Lang}(O) := \emptyset$.
2. $\mathrm{Lang}(I) := \varepsilon$.
3. $\mathrm{Lang}(\mathbf{a}) := \{a\}$ for $a \in A$.

Let F and G be regular expressions, then:

4. $\mathrm{Lang}(F \cup G) := \mathrm{Lang}(F) \cup \mathrm{Lang}(G)$.
5. $\mathrm{Lang}(F \cdot G) := \mathrm{Lang}(F) \cdot \mathrm{Lang}(G)$.
6. $\mathrm{Lang}(F^*) := \mathrm{Lang}(F)^*$.
7. $\mathrm{Lang}((F)) := \mathrm{Lang}(F)$.

[HMU01, 86f][Pip97, 64]

Theorem 2.4.3 *Let A be an alphabet and let L be a recognizable language over A. Then there exists a regular expression E such that the regular language* $\mathrm{Lang}(E) = L$.

Proof. Let $M := (Q, A, \delta, q_0, J)$ be a DFA recognizing L with the finite set of states $Q := \{1, \ldots, n\}$ and the initial state $q_0 := 1$. We define the regular expressions $R_{ij}^{(k)}$ with the corresponding regular language

$$\mathrm{Lang}(R_{ij}^{(k)}) := \{w \mid w \text{ is a path from state } i \text{ to state } j \text{ in } M, \text{ where no intermediate state on the path is greater than } k\}$$

Note that i and j itself can be greater than k. We construct the wanted regular expressions $R_{ij}^{(k)}$ for $k = n$ for all $i, j \in Q$, starting at $k = 0$ and then we define the regular expression $k \geq 1$ recursively. For $k = 0$ and $i, j \in Q$ the path length can be either 0 or 1. If $i \neq j$ then path length is 1 and the following cases can happen:

1. If there is no transition from i to j in the DFA M then we define $R_{ij}^{(k)} := O$.
2. If there is transition $a \in A$ from i to j in the DFA M then we define $R_{ij}^{(k)} := \mathbf{a}$.
3. If there are more transitions $a_1, \ldots, a_m \in A$ from i to j in the DFA M then we define $R_{ij}^{(k)} := \mathbf{a}_1 \cup \ldots \cup \mathbf{a}_m$.

If $i = j$ then the path length can be either 0 or 1:

1. There is no loop from state i to i: $R_{ij}^{(k)} := I$.
2. There are one or more transitions from i to i: $R_{ij}^{(k)} := I \cup \mathbf{a}_1 \cup \ldots \cup \mathbf{a}_m$.

For $k \geq 1$ and $i, j \in Q$ two cases can occur:

1. The path from i to j doesn't go through a state greater than k. So the regular expression stays the same and is defined as $R_{ij}^{(k)} := R_{ij}^{(k-1)}$.
2. The path from i to j goes through the state k. Therefore the path can be split into a path p_{ik} and a path p_{kj}, going from state i to state k without going through any state greater equal than k, as well as a path p_{kk}, going from k to k. The path p_{kk} doesn't go through any state greater equal than k. Therefore $R_{ij}^{(k)} := R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{ki}^{(k-1)}$.

So the union of the two previous regular expressions defines the regular expression

$$R_{ij}^{(k)} := R_{ij}^{(k-1)} \cup R_{ij}^{(k)} = R_{ij}^{(k-1)} \cup R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{ki}^{(k-1)}.$$

After finitely many steps the regular expressions $R_{ij}^{(n)}$ for $i, j \in Q$ are defined. Let R be the set of all *accepted* regular expressions:

$$R := \{R_{1j}^{(n)} \mid j \in J\}.$$

Then

$$L = \text{Lang}(M) = \text{Lang}(\bigcup_{r \in R} r).$$

□

[HMU01, 91ff]

Theorem 2.4.4 *Let A be an alphabet, let E be a regular expression over A and let $L := \text{Lang}(E)$ be the regular language of E. Then the language L is recognizable.*

Proof. Let E be a regular expression and let $\text{Lang}(E)$ be the regular language of E.
Claim: There exists an ε-NFA N_ε with the following properties

1. N_ε has exactly one initial state with no transition into it.
2. N_ε has exactly one final state with no transition from it.

Then by structural induction we show that $\text{Lang}(E) = \text{Lang}(N_\varepsilon)$.

Base cases:

1. $E = O$. The automaton N_ε over A with two states $Q := \{q_0, q_1\}$ with the initial state $I := \{q_0\}$ and the final state $I := \{q_1\}$ fulfills the properties 1 and 2. It holds that
$$\text{Lang}(E) = \emptyset = \text{Lang}(N_\varepsilon).$$
2. $E = I$. The automaton N_ε over A with two states $Q := \{q_0, q_1\}$ with the initial state $I := \{q_0\}$, the final state $I := \{q_1\}$ and the transition $\delta(\varepsilon, q_0) := q_1$ fulfills the properties 1 and 2. It holds that:
$$\text{Lang}(E) = \{\varepsilon\} = \text{Lang}(N_\varepsilon).$$
3. $E = \mathbf{a}$. The automaton N_ε over A with two states $Q := \{q_0, q_1\}$ with the initial state $I := \{q_0\}$, the final state $I := \{q_1\}$ and the transition $\delta(a, q_0) := q_1$ fulfills the properties 1 and 2. It holds that:
$$\text{Lang}(E) = \{a\} = \text{Lang}(N_\varepsilon).$$

Structural induction:

1. $E = F \cup G$. Let N_1 be the ε-NFA recognizing $\text{Lang}(F)$ and let N_2 be the ε-NFA recognizing $\text{Lang}(G)$. We define the automaton N_ε over A consisting of a new initial and final state as well as the two automata N_1 and N_2. The automaton N_ε has ε-transitions from the initial state to the previous initial state of N_1 and N_2 as well as from the previous final states of N_1 and N_2 to the final state of N_ε. Then N_ε fulfills the properties 1 and 2 and it holds that
$$\text{Lang}(E) = \text{Lang}(F) \cup \text{Lang}(G) = \text{Lang}(N_1) \cup \text{Lang}(N_2) = \text{Lang}(N_\varepsilon).$$
2. $E = F \cdot G$. Let N_1 be the ε-NFA recognizing $\text{Lang}(F)$ and let N_2 be the ε-NFA recognizing $\text{Lang}(G)$. We define the automaton N_ε over A consisting of a new initial and final state as well as the two automata N_1 and N_2. The automaton N_ε has a ε-transition from the initial state to the previous initial state of N_1, from the previous final states of N_2 to the final state of N_ε as well as from the previous final state of N_1 to the previous initial state of N_2. Then N_ε fulfills the properties 1 and 2 and it holds that
$$\text{Lang}(E) = \text{Lang}(F) \cdot \text{Lang}(G) = \text{Lang}(N_1) \cdot \text{Lang}(N_2) = \text{Lang}(N_\varepsilon).$$
3. $E = F^*$. Let N_1 be the ε-NFA recognizing $\text{Lang}(F)$. We define the automaton N_ε over A consisting of a new initial and final state as well as the automaton N_1. The automaton n has a ε-transition from the initial state to the previous initial state of N_1 and from the previous final states of N_1 to the final state of N_ε. Additionally, it has a ε-transition from the initial state to the final state to recognize the empty word, i.e. $L(F^0)$, as well as a ε-transition from the previous final state to the previous initial state to recognize any star closure, i.e. $L(F^n)$ for $n \geq 1$. Then N_ε fulfills the properties 1 and 2 and it holds that
$$\text{Lang}(E) = \text{Lang}(F)^* = \text{Lang}(N_1) = \text{Lang}(N_\varepsilon).$$
4. $E = (F)$. Let N_ε be the ε-NFA recognizing $\text{Lang}(F)$. Then also N_ε recognizes $\text{Lang}(F)$ since parenthesis doesn't change anything.

□

[HMU01, 102ff]

3 Connection between Presburger arithmetic and automata theory

3.1 Introduction to first-order languages

Definition 3.1.1 Let $\mathcal{R}$ be a set of relation symbols and $\mathcal{F}$ be a set of function symbols such that a non-negative integer, called *arity*, is assigned to each $r \in \mathcal{R}$ and $f \in \mathcal{R}$. Let X be set of *variables*.

1. The *alphabet of a first-order language* is defined as $\mathcal{A}_\mathcal{S} := \mathcal{A} \cup \mathcal{S}$ where $\mathcal{A}$ contains

   (i) the variables $x_1, x_2, \ldots \in X$,
   (ii) the logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$,
   (iii) the quantifier symbols $\forall$, $\exists$,
   (iv) the comma , and the parentheses (and),

   and the *symbol set* $\mathcal{S} := \mathcal{R} \cup \mathcal{F}$ contains the set of relation symbols $\mathcal{R}$ and the set of the function symbols $\mathcal{F}$.

2. The set of *terms* ($\mathcal{S}$-terms) over the alphabet $\mathcal{A}_S$ is the smallest set T^S such that

   (i) $X \subseteq T^\mathcal{S}$,
   (ii) $\mathcal{F}_0 \subseteq T^\mathcal{S}$ with $\mathcal{F}_0 \subseteq \mathcal{F}$ and
   (iii) $f(t_1, \ldots, t_n) \in T^\mathcal{S}$ with $n \in \mathbb{N}_0$, $t_1, \ldots, t_n \in T^\mathcal{S}$ and $f \in \mathcal{F}_n$.

3. The set of *first-order formulas* of type $\mathcal{S}$ ($\mathcal{S}$-formulas) over the alphabet $\mathcal{A}_\mathcal{S}$ is the smallest set $L^\mathcal{S}$ such that

   (i) $t_1 \approx t_2 \in L^\mathcal{S}$ for terms $t_1, t_2 \in T^\mathcal{S}$,
   (ii) $R(t_1, \ldots, t_n) \in L^\mathcal{S}$ for $n \in \mathbb{N}_0, t_1, \ldots, t_n \in T^\mathcal{S}$ and $R \in \mathcal{R}_n$,

   and for the formulas $F, F_1, F_2 \in L^\mathcal{S}$ also

   (iii) $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2 \in L^\mathcal{S}$ and
   (iv) $\forall x\, F$, $\exists x\, F \in L^\mathcal{S}$ for a variable $x \in X$.

   The formulas obtained from (i) and (ii) are called *atomic*.

Then the language $L^\mathcal{S}$ over the alphabet $\mathcal{A}_\mathcal{S}$ is the *first-order language* of $\mathcal{S}$.

[EFT07, 12ff]

Definition 3.1.2 Let $\mathcal{S}$ be a symbol set such that $\mathcal{S} = \mathcal{R} \cup \mathcal{F}$ for a set of relation symbols $\mathcal{R}$ and a set of function symbols $\mathcal{F}$. Let $L^\mathcal{S}$ be a first-order language of type $\mathcal{S}$. The variable x is *free* if it doesn't belong to a subformula of the form $\forall xF$ or $\exists xF$. Otherwise it is called *bounded*.
The set of first-order formulas with n free variables is defined as:

$$L_n^\mathcal{S} := \{F \mid F \text{ is a first-order formula of type } S \text{ with } n \text{ free variables}\}.$$

The set $L_0^\mathcal{S}$ is the set of *sentences*.

[EFT07, 24]

Definition 3.1.3 Let $\mathcal{S}$ be a symbol set such that $\mathcal{S} = \mathcal{R} \cup \mathcal{F}$ for a set of relation symbols $\mathcal{R}$ and a set of function symbols $\mathcal{F}$. A *first-order structure* of type $\mathcal{S}$ ($\mathcal{S}$-structure) is an ordered pair $\mathbf{A} := (A, S)$, where the set $A \neq \emptyset$ is called *universe* of $\mathbf{A}$ and S contains for each relation symbol $R \in \mathcal{R}$ the relational symbol $R^{\mathbf{A}}$ on A and for each function symbol $f \in \mathcal{F}$ the function symbol $f^{\mathbf{A}}$ on A.

[EFT07, 29]

Definition 3.1.4 Let $\mathcal{S}$ be a symbol set. Let $\mathbf{A} := (A, S)$ be a first order structure of type $\mathcal{S}$. A function $\beta : X \to A$ mapping variables into the universe of $\mathbf{A}$ is called *assignment* in $\mathbf{A}$. An *interpretation* of type $\mathcal{S}$ ($\mathcal{S}$-interpretation) is an ordered pair $\mathfrak{I} := (\mathbf{A}, \beta)$.

[EFT07, 30]

Definition 3.1.5 Let $\mathcal{S}$ be a symbol set. Let $\mathfrak{I} := (\mathbf{A}, \beta)$ be an interpretation of type $\mathcal{S}$. Let x be variable and $a \in A$. The *assignment* of x to a is defined as

$$\beta\frac{a}{x}(y) := \begin{cases} a & \text{if } y = x \\ \beta(y) & \text{otherwise} \end{cases}$$

and

$$\mathfrak{I}\,\frac{a}{x} := (\mathbf{A}, \beta\frac{a}{x})$$

Definition 3.1.6 Let $\mathcal{S}$ be a symbol set such that $\mathcal{S} = \mathcal{R} \cup \mathcal{F}$ for a set of relation symbols and a set of function symbols. Let $\mathfrak{I} := (\mathbf{A}, \beta)$ be an interpretation of type $\mathcal{S}$. The *interpretation of a term* is defined as $\mathfrak{I} : T^{\mathcal{S}} \to A$, where

1. $\mathfrak{I}(x) := \beta(x)$ for a variable x and
2. $\mathfrak{I}(f(t_1, \ldots, t_n)) := f^{\mathbf{A}}(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n))$ for $n \in \mathbb{N}_0$, $t_1, \ldots, t_n \in T^{\mathcal{S}}$ and $f \in \mathcal{F}_n$.

[EFT07, 33]

Definition 3.1.7 Let $\mathcal{S}$ be a symbol set. Let $\mathfrak{I} := (\mathbf{A}, \beta)$ be an interpretation of type $\mathcal{S}$. Let $F \in L^{\mathcal{S}}$ be a first-order formula.
Then $\mathfrak{I}$ *satisfies* F (F is true in $\mathfrak{I}$), with the notation $\mathfrak{I} \models F$, is defined recursively:

1. $\mathfrak{I} \models t_1 \approx t_2$ if and only if $\mathfrak{I}(t_1) = \mathfrak{I}(t_2)$ for terms t_1 and t_2.
2. $\mathfrak{I} \models R(t_1, \ldots, t_n)$ if and only if $R^{\mathbf{A}}(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n))$ for terms $t_1, \ldots, t_n$.
3. $\mathfrak{I} \models F_1 \wedge F_2$ if and only if $\mathfrak{I} \models F_1$ and $\mathfrak{I} \models F_2$.
4. $\mathfrak{I} \models F_1 \vee F_2$ if and only if $\mathfrak{I} \models F_1$ or $\mathfrak{I} \models F_2$.
5. $\mathfrak{I} \models \neg F$ if and only if it is not the case that $\mathfrak{I} \models F$ (notation: $\mathfrak{I} \not\models F$).
6. $\mathfrak{I} \models F_1 \to F_2$ if and only if $\mathfrak{I} \not\models F_1$ or $\mathfrak{I} \models F_2$.
7. $\mathfrak{I} \models F_1 \leftrightarrow F_2$ if and only if ($\mathfrak{I} \not\models F_1$ and $\mathfrak{I} \not\models F_2$) or ($\mathfrak{I} \models F_1$ and $\mathfrak{I} \models F_2$).
8. $\mathfrak{I} \models \forall x F$ if and only if $\mathfrak{I}\,\frac{a}{x} \models F$ for every $a \in A$.
9. $\mathfrak{I} \models \exists x F$ if and only if $\mathfrak{I}\,\frac{a}{x} \models F$ for some $a \in A$.

Let $\mathbf{F}$ be a set of first-order formulas. Then $\mathfrak{I}$ *satisfies* $\mathbf{F}$, with the notation $\mathfrak{I} \models \mathbf{F}$, if $\mathfrak{I} \models F$ for all $F \in \mathbf{F}$.
Instead of writing $\mathfrak{I} \models F$, i.e. $(\mathbf{A}, \beta) \models F$, it will be written as $\mathbf{A} \models F$.

[EFT07, 33f]

Definition 3.1.8 Let $\mathcal{S}$ be a symbol set and let $\mathbf{A} := (A, S)$ be a first-order structure of type $\mathcal{S}$. Then the *first-order theory* of $\mathbf{A}$ is defined as:

$$\mathrm{Th}(\mathbf{A}) := \{F \in L_0^{\mathcal{S}} \mid \mathbf{A} \models F\}.$$

[EFT07, 99]

Definition 3.1.9 Let A be an alphabet and let $L \subseteq A^*$ be a language. Then L is *decidable* if there exists an algorithm that on every input $w \in A^*$ terminates and yields as output **True** if $w \in L$ or **False** if $w \notin L$.

[EFT07, 161]

3.2 Recognizable solutions of first-order formulas

Definition 3.2.1 Let $\mathcal{P} := \{0, 1, +\}$ be a symbol set, where 0 and 1 are nullary function symbols and + is binary function (addition). Let $L^{\mathcal{P}}$ be the first-order language of type $\mathcal{P}$. Let $\mathbf{N} := (\mathbb{N}_0, P)$ be the first-order structure of type $\mathcal{P}$.
Then the first-order theory $\mathrm{Th}(\mathbf{N})$ is called *Presburger arithmetic*.

The Presburger arithmetic, named after M. Presburger, is the first-order theory of natural numbers with the addition as the only operation. In 1929, M. Presburger proved its completeness and due to his constructive proof he also showed, using quantifier elimination, that the Presburger arithmetic is decidable [Pre29].
In the following chapters an approach, based on J. R. Büchi (1960), using automata theory will be used [Büc60][Sha13][Sip13, 255ff].

Remark 3.2.2 From now on let $\mathcal{P}$ be the following symbol set:

$$\mathcal{P} := \{0, 1, +\}.$$

Furthermore, the following shortcuts increases the readability:

1. The natural number $n \geq 2$ is defined as

$$n := 0\underbrace{+1 \ldots + 1}_{n\text{-times}}.$$

2. The constant multiplication $\cdot$ for a variable x and a constant $n \in \mathbb{N}$ is defined as

$$n \cdot x := \underbrace{x + \ldots + x}_{n\text{-times}}.$$

3. The relation $\leq$ for variables x and y is defined as

$$x \leq y :\Leftrightarrow \exists\, z : x + z = y.$$

Theorem 3.2.3 (Basis Representation Theorem) *Let $b \in \mathbb{N}, b > 1$ and let $n \in \mathbb{N}$. Then there exists exactly one sequence $\langle r_i \rangle$ such that:*

1. $n = \sum_{i=0}^{k} r_i \cdot b^i$
2. $\forall\, i \in \{0, \ldots, k\}\ 0 \leq r_i < b$
3. $r_k \neq 0$.

This uniquely sequence $\langle r_i \rangle$ is called base b representation *of n*

Proof. Let $n \in \mathbb{N}$ and let $\langle r_i \rangle$ be the base b representation of n with

$$n := \sum_{i=0}^{k} r_i \cdot b^i.$$

Let $s_b(n)$ be the number of ways of writing n in base b representation. Let $j \in \mathbb{N}$ be the index of the first non-zero element of the sequence. Then base b representation of $n-1$ can be derived

from the base b representation of n (for $n > 1$):

$$\begin{aligned} n-1 =& \Big(\sum_{i=0}^{k} r_i \cdot b^i\Big) - 1 \\ =& \Big(\sum_{i=j}^{k} r_i \cdot b^i\Big) - 1 \\ =& \Big(\sum_{i=j+1}^{k} r_i \cdot b^i\Big) + (r_j - 1)\, b^j + b^j - 1 \\ =& \Big(\sum_{i=j+1}^{k} r_i \cdot b^i\Big) + (r_j - 1)\, b^j + \Big(\sum_{i=0}^{j-1} (b-1) b^i\Big) \end{aligned}$$

Therefore $s_b(n) \leq s_b(n-1)$. Note that for $n = 1$ there exists exactly one base b representation, namely $\langle 1 \rangle$. Since there exists at least one base b representation for b^n, it follows that:

$$1 \leq s_b(b^n) \leq s_b(n) \leq s_b(1) = 1$$

and thus $s_b(n) = 1$. □

[BRT]

Definition 3.2.4 Let $n \in \mathbb{N}$ with its unique base 2 representation $\langle r_i \rangle$ such that

$$n = \sum_{i \in \mathbb{N}_0} r_i \cdot 2^i \text{ with } r_i \in \{0, 1\}.$$

Then $[r_0 \dots r_m]_2$ is the *binary representation* of n.

Remark 3.2.5 Note that in the previous definition the base 2 representation of a natural number is read in the **reverse** direction (motivated by the accepted words of finite automata) compared to the usual convention.

Definition 3.2.6 Let $A := \{0, 1\}$ and let $n \in \mathbb{N}_0$. Let $w \in (A^n)^*$ be a word over the alphabet A^n. Let $m \in \mathbb{N}$ and let $w_1, \dots, w_m \in A^n$ such that $w := w_1 \dots w_m$.
For $1 \leq i \leq n$

$$w^{(i)} := w_1(i) \dots w_m(i) \in A^*$$

denotes the word (over the alphabet A) of the i-th component of each letter of w.
The function N, mapping a word over the alphabet A to a natural number, is defined as follows:

$$\begin{aligned} N : A^* &\to \mathbb{N}_0 \\ w &\mapsto \sum_{i=1}^{|w|} w_i \cdot 2^{i-1}. \end{aligned}$$

Definition 3.2.7 Let $A := \{0, 1\}$ and let $n \in \mathbb{N}_0$. Let $L^{\mathcal{P}}$ be the first-order language of type $\mathcal{P}$. Let $\mathbf{N} := (\mathbb{N}, P)$ be a first-order structure of type $\mathcal{P}$ and let $\mathfrak{I} := (\mathbf{N}, \beta)$ be an interpretation of type $\mathcal{P}$. Let $F \in L_n^{\mathcal{P}}$ be a first-order formula of type $\mathcal{P}$ with n free variables.
The *solutions* of F over A^n is the language

$$\mathrm{Sol}_{A^n}(F) := \Big\{ w \in (A^n)^* \mid \mathfrak{I} \, \frac{N(w^{(1)}), \dots, N(w^{(n)})}{x_1, \dots, x_n} \models F \Big\},$$

such that upon interpretation of the words as natural numbers the formula is true in $\mathfrak{I}$.

Remark 3.2.8 Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$.

1. A number in base 2 representation is unique, due to its definition forcing the last entry to be nonzero. However, for a word $w \in A^*$ with $w := w_1 \dots w_m$ the following holds:

$$N(w_1 \dots w_m) = N(w_1 \dots w_m 0) = N(w_1 \dots w_m 00) = \dots = \sum_{i=1}^{m} w_i \cdot 2^{i-1}.$$

   This will lead to the following definition 3.2.9, where as soon as a word is accepted then also the word with added or cancelled *leading zeros* should be accepted.

2. Let the word $w := (0, \dots, 0) \in \mathrm{Sol}_{A^n}(F)$. Then the empty word $\varepsilon \in \mathrm{Sol}_{A^n}(F)$ since

$$N(\varepsilon) = \sum_{i=1}^{0} w_i \cdot 2^{i-1} = 0 = N(0).$$

3. Note that the solutions are defined for formulas with $n \in \mathbb{N}_0$ free variables.
   For $n = 0$ the alphabet is defined as the set of the empty tuple:

$$A^0 := \{(\,)\}.$$

   Let F be a first-order formulas containing no free variables.
   Case 1: $\mathfrak{I} \models F$.
   Then the $\mathfrak{I}$ satisfies the formula F for any $w \in (A^0)^*$, hence

$$\mathrm{Sol}_{A^0}(F) = (A^0)^*.$$

   Case 2: $\mathfrak{I} \not\models F$.
   Then, on the other hand, the $\mathfrak{I}$ doesn't satisfy the formula F for any $w \in (A^0)^*$ and so

$$\mathrm{Sol}_{A^0}(F) = \emptyset.$$

   This will fit to the upcoming concepts of the leading zero completion and the negation of formulas corresponding to the complement of languages.

Definition 3.2.9 Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let L be a language over A^n and let

$$Z_n := \{w \in (\{0\}^n)^*\} \subseteq (A^n)^*$$

be the language over the alphabet A^n accepting only the zero words (words consisting of the *zero letter* $\mathbf{0} := (0, \dots, 0)$). The *leading zero completion* of L is defined as

$$\overline{L}^{\mathbf{0}} := L / Z_n \cup L \cdot Z_n.$$

Definition 3.2.10 Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let L be a language over the alphabet A^n. Then L is *leading zero complete* if $L = \overline{L}^{\mathbf{0}}$.

Note that, due to its construction, $L \subseteq \overline{L}^{\mathbf{0}}$ holds for any language L over the alphabet A^n.

Lemma 3.2.11 *Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $F \in L_n^{\mathcal{P}}$ be a first-order formula of type $\mathcal{P}$ with n free variables. Then $\mathrm{Sol}_{A^n}(F)$ is leading zero complete.*

Proof. Assume $w \in \overline{\mathrm{Sol}_{A^n}(F)}^{\mathbf{0}}$, i.e.

$$w \in \mathrm{Sol}_{A^n}(F) / Z_n \cup \mathrm{Sol}_{A^n}(F) \cdot Z_n.$$

We show that $w \in \mathrm{Sol}_{A^n}(F)$.

Case 1: $w \in \mathrm{Sol}_{A^n}(F) / Z_n$, i.e.

$$w \in \{x \in (A^n)^* \mid \exists y \in Z_n : xy \in \mathrm{Sol}_{A^n}(F)\}.$$

So $wy \in \mathrm{Sol}_{A^n}(F)$ for a zero word $y \in Z_n$. Hence

$$\mathfrak{I}\,\frac{N\big((w\mathbf{0}\dots\mathbf{0})^{(1)}\big),\dots,N\big((w\mathbf{0}\dots\mathbf{0})^{(n)}\big)}{x_1,\dots,x_n} \models F.$$

But since $N\big((w\mathbf{0}\dots\mathbf{0})^{(i)}\big) = N(w^{(i)})$ (for $1 \le i \le n$) it follows that $w \in \mathrm{Sol}_{A^n}(F)$.

Case 2: $w \in \mathrm{Sol}_{A^n}(F) \cdot Z_n$, i.e.

$$w \in \{xy \in (A^n)^* \mid x \in \mathrm{Sol}_{A^n}(F), y \in Z_n\}.$$

Then w has the form $w_1 \dots w_m \mathbf{0} \dots \mathbf{0}$ with $w_1 \dots w_m \in \mathrm{Sol}_{A^n}(F)$. Hence

$$\mathfrak{I}\,\frac{N\big((w_1\dots w_m)^{(1)}\big),\dots,N\big((w_1\dots w_m)^{(n)}\big)}{x_1,\dots,x_n} \models F.$$

But since for $1 \le i \le n$

$$N\big((w_1\dots w_m)^{(i)}\big) = N\big((w_1\dots w_m\mathbf{0}\dots\mathbf{0})^{(i)}\big) = N(w^{(i)})$$

it follows that $w \in \mathrm{Sol}_{A^n}(F)$. □

Lemma 3.2.12 *Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let L be a recognizable language over the alphabet A^n. Then $\overline{L}^{\mathbf{0}}$ is recognizable.*

Proof. The language Z_n is recognizable, since the deterministic finite automaton $(\{0,1\}, \{0,1\}^n, \delta, 0, \{0\})$ with the transition function

$$\delta(a,0) := \begin{cases} 0 & \text{if } a = (0,\dots 0) \\ 1 & \text{otherwise} \end{cases}$$
$$\delta(a,1) := 1 \text{ for } a \in A^n$$

recognizes Z_n.
The recognizability of $\overline{L}^{\mathbf{0}} = L / Z_n \cup L \cdot Z_n$ follows with the lemmas 2.2.14 (right quotient), 2.2.10 (product) and 2.2.6 (union). □

Definition 3.2.13 Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $M := (Q, A^n, \delta, q_0, J)$ be a deterministic finite automaton over the alphabet A^n. The deterministic finite automaton M is called *leading zero complete* if the following two conditions holds:

1. $\forall q \in J\ \delta(\mathbf{0}, q) \in J$
2. $\forall q \in (Q \setminus J)\ \delta(\mathbf{0}, q) \notin J$.

Let N be a deterministic finite automaton over the A^n.

Define $\overline{N}^{\mathbf{0}}$ as a deterministic finite automaton with $\overline{\text{Lang}(N)}^{\mathbf{0}} = \text{Lang}(\overline{N}^{\mathbf{0}})$ such that it is minimal and leading zero complete.

Lemma 3.2.14 *Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $M := (Q, A^n, \delta, q_0, J)$ be a deterministic finite automaton over the alphabet A^n. The language* Lang(M) *is leading zero complete if and only if the deterministic finite automaton M is leading zero complete.*

Proof. Assume M is a leading zero complete deterministic finite automaton.
Let $L := \text{Lang}(M)$ and let $w \in \overline{L}^{\mathbf{0}} = L/Z_n \cup L \cdot Z_n$.

Case 1: $w \in L/Z_n$, i.e.

$$w \in \{x \in (A^n)^* \mid \exists y \in Z_n : xy \in L\}.$$

So $wy \in L$ for a zero word $y \in Z_n$. Since

$$\widehat{\delta}(wy, q_0) \in J$$

and there are no **0**-transitions from a non-final state to a final state (condition 1), it follows that

$$\widehat{\delta}(w, q_0) \in J.$$

Hence $w \in L$.

Case 2: $w \in L \cdot Z_n$, i.e.

$$w \in \{xy \in (A^n)^* \mid x \in L, y \in Z_n\}.$$

Then w has the form $w_1 \dots w_m \mathbf{0} \dots \mathbf{0}$ with $w_1 \dots w_m \in L$. Since

$$\widehat{\delta}(w_1 \dots w_m, q_0) \in J$$

and there's a **0**-transitions from a final state to a final state (condition 2), it follows that

$$\widehat{\delta}(w, q_0) \in J.$$

Hence $w \in L$.

Assume $L := \text{Lang}(M)$ is a leading zero complete language, i.e. $L = L/Z_n \cup L \cdot Z_n$.
Let $q \in J$ with $w \in L$ such that $\widehat{\delta}(w, q_0) = q$. Since L is leading zero complete also $w\mathbf{0} \in L$, i.e. $\widehat{\delta}(w\mathbf{0}, q_0) \in J$. Hence

$$\delta(\mathbf{0}, \widehat{\delta}(w, q_0)) = \delta(\mathbf{0}, q) \in J.$$

Assume $\exists q \in (Q \setminus J) : \delta(\mathbf{0}, q) \in J$. Let $w \in A^n$ s.t $\widehat{\delta}(w, q_0) = q$. Hence

$$\widehat{\delta}(w\mathbf{0}, q_0) = q$$

and $w\mathbf{0} \in L$. Since L is leading zero complete also $w \in L$, therefore the state q is a final state, i.e. $q \in J$. Contradiction to the assumption $q \in Q \setminus J$. □

Lemma 3.2.15 *Let* $A := \{0,1\}$.

1. *Let* $F_1(x)$ *be the first-order formula* $x \approx 0$ *of type* $\mathcal{P}$. *Then* $\mathrm{Sol}_{A^1}(F_1)$ *is a recognizable.*
2. *Let* $F_2(x)$ *be the first-order formula* $x \approx 1$ *of type* $\mathcal{P}$. *Then* $\mathrm{Sol}_{A^1}(F_2)$ *is a recognizable.*
3. *Let* $F_3(x,y)$ *be the first-order formula* $x \approx y$ *of type* $\mathcal{P}$. *Then* $\mathrm{Sol}_{A^2}(F_3)$ *is a recognizable.*
4. *Let* $F_4(x,y,z)$ *be the first-order formula* $x + y \approx z$ *of type* $\mathcal{P}$. *Then* $\mathrm{Sol}_{A^3}(F_4)$ *is a recognizable.*

Proof. 1. The formula F_1 is satisfied for the assignment $x = 0$ and therefore for the words $(0), (0)(0), \ldots$ as well as for the word ε. Hence

$$\mathrm{Sol}_{A^1}(F_1) = \{w \in (\{0\}^1)^*\}.$$

The DFA $M_1 := (\{0,1\}, \{0,1\}^1, \delta, 0, \{0\})$ with the transition function

$$\begin{aligned} \delta((0),0) &:= 0 & \qquad \delta((1),0) &:= 1 \\ \delta((0),1) &:= 1 & \qquad \delta((1),1) &:= 1 \end{aligned}$$

recognizes the language $\mathrm{Sol}_{A^1}(F_1)$.

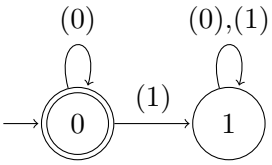


Figure 3.1: The transition diagram of DFA M_1.

2. The formula F_2 is satisfied for the assignment $x = 1$ and therefore for the words $(1), (1)(0), (1)(0)(0), \ldots$ Hence

$$\mathrm{Sol}_{A^1}(F_2) = \{w \in (A^1)^* \mid w = (1)u \text{ with } u \in (\{0\}^1)^*\}.$$

The DFA $M_2 := (\{0,1,2\}, \{0,1\}^1, \delta, 0, \{1\})$ with the transition function

$$\begin{aligned} \delta((0),0) &:= 2 & \qquad \delta((1),0) &:= 1 \\ \delta((0),1) &:= 1 & \qquad \delta((1),1) &:= 2 \\ \delta((0),2) &:= 2 & \qquad \delta((1),2) &:= 2 \end{aligned}$$

recognizes the language $\mathrm{Sol}_{A^1}(F_2)$.

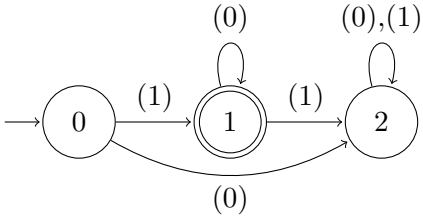


Figure 3.2: The transition diagram of DFA M_2.

3. The formula F_3 is satisfied for the assignments x and y if $x = y$, i.e. the words which consists of the letters $(0,0)$ and $(1,1)$. Hence

$$\mathrm{Sol}_{A^2}(F_3) = \{w \in \{(0,0),(1,1)\}^*\}.$$

The DFA $M_3 := (\{0,1\}, \{0,1\}^2, \delta, 0, \{0\})$ with the transition function

$$\begin{aligned} \delta((0,0),0) &:= 0 & \delta((1,1),0) &:= 0 \\ \delta((0,1),0) &:= 1 & \delta((1,0),0) &:= 1 \\ \delta((0,0),1) &:= 1 & \delta((1,1),1) &:= 1 \\ \delta((0,1),1) &:= 1 & \delta((1,0),1) &:= 1 \end{aligned}$$

recognizes the language $\mathrm{Sol}_{A^2}(F_3)$.

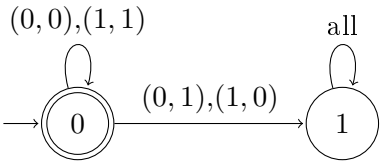


Figure 3.3: The transition diagram of DFA M_3.

4. The formula F_4 is satisfied for the assignments x, y and z if x and y sum up to z. Hence

$$\mathrm{Sol}_{A^3}(F_4) = \{w \in (A^3)^* \mid N(w^{(1)}) + N(w^{(2)}) = N(w^{(3)})\}.$$

The following property for the letter $a = (a_1, a_2, a_3)$, the current state q and the future state p corresponds to the addition of base 2 representations:

$$a_1 + a_2 = a_3 + q + 2(p - q),$$

where the state 0 denotes carry 0 and state 1 denotes carry 1. This defines the DFA $M_4 := (\{0,1,2\}, \{0,1\}^3, \delta, 0, \{0\})$ with the transition function

$$\delta(a,0) := \begin{cases} 0 \text{ if } a = (0,0,0),(1,0,1) \text{ or } (0,1,1) \\ 1 \text{ if } a = (1,1,0) \\ 2 \text{ otherwise} \end{cases}$$

$$\delta(a,1) := \begin{cases} 0 \text{ if } a = (0,0,1) \\ 1 \text{ if } a = (1,0,0),(0,1,0) \text{ or } (1,1,1) \\ 2 \text{ otherwise} \end{cases}$$

$$\delta(a,2) := 1 \text{ for } a \in \{0,1\}^3.$$

Then M_4 recognizes the language $\mathrm{Sol}_{A^3}(F_4)$.

□

Remark 3.2.16 Note that the four previous deterministic finite automata are leading zero complete since they have only one the final state, which

1. loops with the letter $\mathbf{0}$, i.e. $\forall q \in J\ \delta(\mathbf{0}, q) \in J$ and
2. has no ingoing $\mathbf{0}$-transition, i.e. $\forall q \in (Q \setminus J)\ \delta(\mathbf{0}, q) \notin J$.

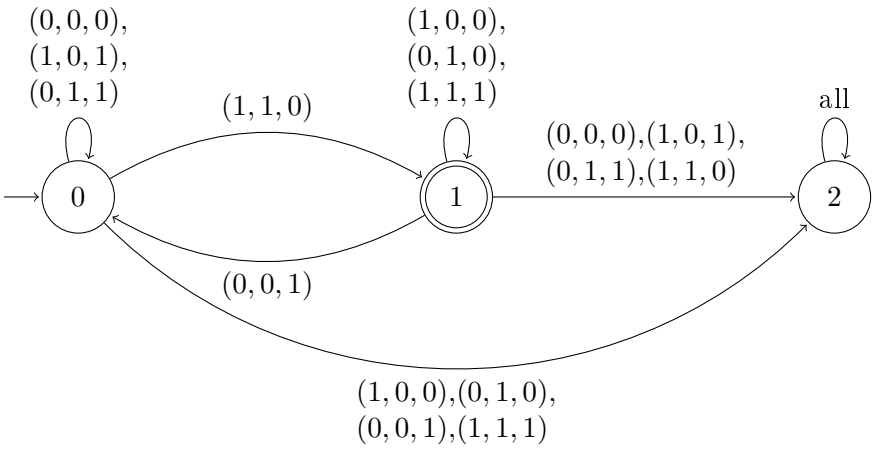


Figure 3.4: The transition diagram of DFA M_4.

Lemma 3.2.17 *Let $A := \{0,1\}$ and $n \in \mathbb{N}_0$. Let $F, F_1, F_2 \in L_n^{\mathcal{P}}$ be first-order formulas of type $\mathcal{P}$ with n free variables such that their solutions are recognizable. Then the following holds:*

1. $\mathrm{Sol}_{A^n}(F_1 \wedge F_2) = \mathrm{Sol}_{A^n}(F_1) \cap \mathrm{Sol}_{A^n}(F_2)$ *and* $\mathrm{Sol}_{A^n}(F_1 \wedge F_2)$ *is recognizable.*
2. $\mathrm{Sol}_{A^n}(\neg F) = \mathrm{Cpl}\big(\mathrm{Sol}_{A^n}(F)\big)$ *and* $\mathrm{Sol}_{A^n}(\neg F)$ *is recognizable.*
3. $\mathrm{Sol}_{A^n}(F_1 \vee F_2) = \mathrm{Sol}_{A^n}(F_1) \cup \mathrm{Sol}_{A^n}(F_2)$ *and* $\mathrm{Sol}_{A^n}(F_1 \vee F_2)$ *is recognizable.*
4. $\mathrm{Sol}_{A^n}(F_1 \to F_2) = \mathrm{Cpl}\big(\mathrm{Sol}_{A^n}(F_1)\big) \cup \mathrm{Sol}_{A^n}(F_2)$ *and* $\mathrm{Sol}_{A^n}(F_1 \to F_2)$ *is recognizable.*
5. $\mathrm{Sol}_{A^n}(F_1 \leftrightarrow F_2) = \big(\mathrm{Cpl}(\mathrm{Sol}_{A^n}(F_1)) \cap \mathrm{Cpl}(\mathrm{Sol}_{A^n}(F_2))\big) \cup \big(\mathrm{Sol}_{A^n}(F_1) \cap \mathrm{Sol}_{A^n}(F_2)\big)$ *and* $\mathrm{Sol}_{A^n}(F_1 \leftrightarrow F_2)$ *is recognizable.*

Let φ_{n+1} be the homomorphism ignoring the $n+1$-th component of each letter:

$$\varphi_{n+1} : A^{n+1} \to A^n$$
$$(a_1, \ldots, a_n, a_{n+1}) \mapsto (a_1, \ldots, a_n).$$

Let $F \in L_{n+1}^{\mathcal{P}}$ be a formula with $n+1$ free variables such that its solutions are recognizable.

6. $\mathrm{Sol}_{A^n}(\exists x_{n+1} : F) = \overline{\varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big)}^{\mathbf{0}}$
7. $\mathrm{Sol}_{A^n}(\forall x_{n+1} : F) = \mathrm{Cpl}\left(\overline{\varphi_{n+1}\big(\mathrm{Cpl}(\mathrm{Sol}_{A^{n+1}}(F))\big)}^{\mathbf{0}}\right)$

Proof. 1. Assume $w \in \mathrm{Sol}_{A^n}(F_1 \wedge F_2)$. Since $w \in \mathrm{Sol}_{A^n}(F_1 \wedge F_2)$, the interpretation of w satisfies $F_1 \wedge F_2$, i.e.

$$\mathfrak{I}\,\frac{N(w^{(1)}), \ldots, N(w^{(n)})}{x_1, \ldots, x_n} \models F_1 \wedge F_2.$$

Hence

$$\mathfrak{I}\,\frac{N(w^{(1)}), \ldots, N(w^{(n)})}{x_1, \ldots, x_n} \models F_1 \text{ and}$$
$$\mathfrak{I}\,\frac{N(w^{(1)}), \ldots, N(w^{(n)})}{x_1, \ldots, x_n} \models F_2.$$

So $w \in \mathrm{Sol}_{A^n}(F_1)$ and $w \in \mathrm{Sol}_{A^n}(F_2)$, therefore

$$w \in \mathrm{Sol}_{A^n}(F_1) \cap \mathrm{Sol}_{A^n}(F_2).$$

Assume $w \in \mathrm{Sol}_{A^n}(F_1) \cap \mathrm{Sol}_{A^n}(F_2)$, i.e. $w \in \mathrm{Sol}_{A^n}(F_1)$ and $w \in \mathrm{Sol}_{A^n}(F_2)$.
Hence

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models F_1 \text{ and}$$
$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models F_2.$$

Therefore

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models F_1 \wedge F_2.$$

So $w \in \mathrm{Sol}_{A^n}(F_1 \wedge F_2)$. The recognizability follows from the closure property of lemma 2.2.6.

2. Assume $w \in \mathrm{Sol}_{A^n}(\neg F)$. We show that $w \in \mathrm{Cpl}\left(\mathrm{Sol}_{A^n}(F)\right)$.
Since $w \in \mathrm{Sol}_{A^n}(\neg F)$,

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models \neg F.$$

So it is not the case that the interpretation of w satisfies F.
Therefore $w \in (A^n)^* \setminus \mathrm{Sol}_{A^n}(F)$ and so

$$w \in \mathrm{Cpl}\left(\mathrm{Sol}_{A^n}(F)\right).$$

Assume $w \in \mathrm{Cpl}\left(\mathrm{Sol}_{A^n}(F)\right)$. We show that $w \in \mathrm{Sol}_{A^n}(\neg F)$.
Since $w \in (A^n)^* \setminus \mathrm{Sol}_{A^n}(F)$, it doesn't hold that

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models F.$$

So

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models \neg F$$

and therefore

$$w \in \mathrm{Sol}_{A^n}(\neg F).$$

The recognizability follows from the closure property of lemma 2.2.4.

3. Since $\mathrm{Sol}_{A^n}(F_1 \vee F_2) = \mathrm{Sol}_{A^n}\left(\neg(\neg F_1 \wedge \neg F_2)\right)$, it follows from the previous statements.

4. Since $\mathrm{Sol}_{A^n}(F_1 \to F_2) = \mathrm{Sol}_{A^n}(\neg F_1 \vee F_2)$, it follows from the previous statements.

5. Since $\mathrm{Sol}_{A^n}(F_1 \to F_2) = \mathrm{Sol}_{A^n}\left((\neg F_1 \wedge \neg F_2) \vee (F_1 \wedge F_2)\right)$, it follows from the previous statements.

6. Assume $w \in \mathrm{Sol}_{A^n}(\exists x_{n+1} : F)$. We show that $w \in \overline{\varphi_{n+1}\big(\mathrm{Sol}_{A^n}(F)\big)}^{\mathbf{o}}$.
For $w \in \mathrm{Sol}_{A^n}(\exists x_{n+1} : F)$ it follows that:

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)})}{x_1,\ldots,x_n} \models \exists x_{n+1} : F.$$

Let u be the word over the alphabet A such that F interpreted upon w and u is satisfied:

$$\mathfrak{I}\,\frac{N(w^{(1)}),\ldots,N(w^{(n)}),N(u)}{x_1,\ldots,x_n,x_{n+1}} \models F.$$

Note that not necessarily $|u| = |w|$.

Case 1: $|u| > |w|$. We define $\tilde{w}$ such that

$$\begin{aligned}\varphi_{n+1}(\tilde{w}) &= w\mathbf{0}\ldots\mathbf{0}, \text{ where the zero word has length } |u| - |w| \text{ and}\\ \tilde{w}^{(n+1)} &= u.\end{aligned}$$

Due its construction

$$\tilde{w} \in \mathrm{Sol}_{A^{n+1}}(F)$$

and hence

$$\varphi_{n+1}(\tilde{w}) \in \varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F)).$$

Therefore

$$w\mathbf{0}\ldots\mathbf{0} \in \varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F)).$$

The set $\varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F))$ is the projection of the solutions from F over A^{n+1} to A^n, allowing no conclusion on its words to be drawn. Hence it might not hold that

$$w \in \varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F))$$

but externally cutting of the zeros from the set yields

$$w \in \varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F))/Z_n.$$

So

$$w \in \overline{\varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F))}^{\mathbf{o}}.$$

Case 2: $|u| \leq |w|$. We define $\tilde{w}$ such that

$$\begin{aligned}\varphi_{n+1}(\tilde{w}) &= w\\ \tilde{w}^{(n+1)} &= u\mathbf{0}\ldots\mathbf{0}, \text{ where the zero word has length } |w| - |u|.\end{aligned}$$

Then

$$\begin{aligned}\tilde{w} &\in \mathrm{Sol}_{A^{n+1}}(F), \text{ hence}\\ \varphi_{n+1}(\tilde{w}) &\in \varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F)) \text{ and}\\ w &\in \varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F)).\end{aligned}$$

Therefore

$$w \in \overline{\varphi_{n+1}(\mathrm{Sol}_{A^{n+1}}(F)}^{\mathbf{o}}.$$

For the opposite direction assume $w \in \overline{\varphi_{n+1}\big(\mathrm{Sol}_{A^n}(F)\big)}^{\mathbf{0}}$, hence

$$w \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big)/Z_n \cup \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big) \cdot Z_n.$$

We show that $w \in \mathrm{Sol}_{A^n}(\exists x_{n+1} : F)$.

Case 1: $w \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big)/Z_n$, i.e.

$$w \in \{x \in A^{n+1} \mid \exists y \in Z_n : xy \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big)\}.$$

So $wy \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big)$ for a zero word $y \in Z_n$. Let $\tilde{w} \in A^{n+1}$ such that $\varphi_{n+1}(\tilde{w}) = w$ and

$$\mathfrak{I}\,\frac{N(\tilde{w}^{(1)}), \ldots, N(\tilde{w}^{(n+1)})}{x_1, \ldots, x_{n+1}} \models F.$$

Hence with the word $\tilde{w}^{(n+1)}$

$$\mathfrak{I}\,\frac{N(\tilde{w}^{(1)}), \ldots, N(\tilde{w}^{(n)})}{x_1, \ldots, x_n} \models \exists x_{n+1} : F$$

and so

$$\mathfrak{I}\,\frac{N((w\mathbf{0}\ldots\mathbf{0})^{(1)}), \ldots, N((w\mathbf{0}\ldots\mathbf{0})^{(n)})}{x_1, \ldots, x_n} \models \exists x_{n+1} : F$$

Since $N((w\mathbf{0}\ldots\mathbf{0})^{(i)}) = N(w^{(i)})$ it follows that $w \in L$.

Case 2: $w \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big) \cdot Z_n$, i.e.

$$w \in \{xy \in A^{n+1} \mid x \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big), y \in Z_n\}.$$

Then w has the form $w_1 \ldots w_m\mathbf{0}\ldots\mathbf{0}$ with $w_1 \ldots w_m \in \varphi_{n+1}\big(\mathrm{Sol}_{A^{n+1}}(F)\big)$.
Let $\tilde{w} \in A^{n+1}$ such that $\varphi_{n+1}(\tilde{w}) = w_1 \ldots w_m$ and

$$\mathfrak{I}\,\frac{N(\tilde{w}^{(1)}), \ldots, N(\tilde{w}^{(n+1)})}{x_1, \ldots, x_{n+1}} \models F.$$

Hence with the word $\tilde{w}^{(n+1)}$

$$\mathfrak{I}\,\frac{N(\tilde{w}^{(1)}), \ldots, N(\tilde{w}^{(n)})}{x_1, \ldots, x_n} \models \exists x_{n+1} : F$$

and for $1 \leq i \leq n$ it holds that $(w_1 \ldots w_m)^{(i)} = \tilde{w}^{(i)}$. So

$$\mathfrak{I}\,\frac{N((w_1 \ldots w_m)^{(1)}), \ldots, N((w_1 \ldots w_m)^{(n)})}{x_1, \ldots, x_n} \models \exists x_{n+1} : F$$

Since $N((w_1 \ldots w_m)^{(i)}) = N((w_1 \ldots w_m\mathbf{0}\ldots\mathbf{0})^{(i)})$ it follows that $w \in \mathrm{Sol}_{A^{n+1}}(F)$.
Therefore $w \in \mathrm{Sol}_{A^{n+1}}(F)$.
The recognizability follows from the lemmas 2.2.18 (homomorphic image) and 3.2.12 (leading zero complete).

7. Since $\mathrm{Sol}_{A^n}(\forall x_{n+1} : F) = \mathrm{Sol}_{A^n}(\neg\exists x_{n+1} : \neg F)$, it follows from the previous statements.

□

Remark 3.2.18 Let $F \in L_1^{\mathcal{S}}$ be a first-order formula of type $\mathcal{P}$ with one free variable x_1. The previous lemma states for solutions of the formula $\exists x_1 F$:

$$\mathrm{Sol}_{A^0}(\exists x_1 F) = \overline{\varphi_1\big(\mathrm{Sol}_{A^1}(F)\big)}^{\mathbf{0}},$$

where $\varphi_1\big(\mathrm{Sol}_{A^1}(F)\big) \subseteq (A^0)^*$ might not be leading zero complete.
Case 1: $\exists w \in (A^0)^* : w \in \varphi_1\big(\mathrm{Sol}_{A^1}(F)\big)$, i.e. the word w will be of the form $w = (\,)\ldots(\,)$. With the leading zero completion, where the zero letter $\mathbf{0} = (\,)$ over the alphabet A^0 is added or cancelled, it follows that:

$$\mathrm{Sol}_{A^0}(\exists x_1 F) = (A^0)^*.$$

Therefore a minimal deterministic finite automaton recognizing these language will be

$$(\{0\}, A^0, \delta, 0, \{0\})$$

with

$$\delta((\,), 0) = 0.$$

Case 2: $\varphi_1\big(\mathrm{Sol}_{A^1}(F)\big) = \emptyset$. Hence also the leading zero completion of this language is the empty set and it is recognized by the minimal deterministic finite automaton

$$(\{0\}, A^0, \delta, 0, \emptyset)$$

with

$$\delta((\,), 0) = 0.$$

It might be the case that two first-order formulas of type $\mathcal{P}$ don't have the same free variables and hence their solutions do not share the same alphabet. However, the following alphabet will introduce the concept of adding additional variables to formula, i.e. increasing the alphabet of the solutions of the the formula.

Lemma 3.2.19 *Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $F \in L_n^{\mathcal{P}}$ be a first-order formula of type $\mathcal{P}$ with n free variables such that $\mathrm{Sol}_{A^n}(F)$ is recognizable.*
Let φ_{n+1} be the homomorphism ignoring the $n+1$-th component of each letter:

$$\begin{aligned} \varphi_{n+1} : A^{n+1} &\to A^n \\ (a_1, \ldots, a_n, a_{n+1}) &\mapsto (a_1, \ldots, a_n). \end{aligned}$$

Then $\varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)$ is leading zero complete and recognizable.

Proof. Let $w \in \overline{\varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)}^{\mathbf{0}}$, i.e.

$$w \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big) / Z_{n+1} \cup \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big) \cdot Z_{n+1}.$$

Case 1: $w \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big) / Z_{n+1}$, i.e.

$$w \in \{x \mid \exists y \in Z_{n+1} : xy \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)\}.$$

So $wy \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)$ for a zero word $y \in Z_{n+1}$. Hence

$$wy \in \{u \in (A^{n+1})^* \mid \varphi_{n+1}(u) \in \mathrm{Sol}_{A^n}(F)\}$$

So

$$\varphi_{n+1}(wy) = \varphi_{n+1}(w)\varphi_{n+1}(y) \in \mathrm{Sol}_{A^n}(F).$$

Since $\mathrm{Sol}_{A^n}(F)$ is leading zero complete, also

$$\varphi_{n+1}(w) \in \mathrm{Sol}_{A^n}(F).$$

Hence $w \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)$.

Case 2: $w \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big) \cdot Z_{n+1}$, i.e.

$$w \in \{xy \mid x \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big), y \in Z_{n+1}\}.$$

So $w = xy$ with $x \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)$ and $y \in Z_{n+1}$. Hence

$$\varphi_{n+1}(x) \in \mathrm{Sol}_{A^n}(F).$$

Since $\mathrm{Sol}_{A^n}(F)$ is leading zero complete, also

$$\varphi_{n+1}(x)\varphi_{n+1}(y) = \varphi_{n+1}(xy) = \varphi_{n+1}(w) \in \mathrm{Sol}_{A^n}(F).$$

Hence $w \in \varphi_{n+1}^{-1}\big(\mathrm{Sol}_{A^n}(F)\big)$.

The recognizability follows with lemma 2.2.21. □

Theorem 3.2.20 *Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $F \in L_n^{\mathcal{P}}$ be a first-order formula of type $\mathcal{P}$ with n free variables.*
Then $\mathrm{Sol}_{A^n}(F)$ is recognizable.

Proof. Any first-order formula F of type $\mathcal{P}$ can be constructed from the four formulas from lemma 3.2.15 together with the first-order constructions from lemma 3.2.17. The solutions of the four formulas from the first lemma are recognizable and the constructions from the second lemma applied on recognizable solutions remain recognizable. □

3.3 Deterministic finite automata of first-order formulas

Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. In the previous chapter we proved that the language $\mathrm{Sol}_{A^n}(F)$, the solutions of F, for a first-order formula $F \in L_n^{\mathcal{P}}$ is a recognizable. This motivates the definition of a deterministic finite automaton of a first order formula.

Definition 3.3.1 Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $F \in L_n^{\mathcal{P}}$ be a first-order formula of type $\mathcal{P}$ with n free variables. The *deterministic finite automaton of a first-order formula* $\mathrm{Aut}_{A^n}(F)$ is a minimal deterministic finite automaton (up to renaming of the states) over the alphabet A^n such that $\mathrm{Sol}_{A^n}(F) = \mathrm{Lang}(\mathrm{Aut}_{A^n}(F))$ holds.

Lemma 3.3.2 *Let $A := \{0,1\}$. Then the first-order formulas $x \approx 0$, $x \approx 1$, $x \approx y$ and $x+y \approx z$ have the following minimal deterministic finite automata:*

1. $\mathrm{Aut}_{A^1}(x \approx 0) \cong (\{0,1\}, \{0,1\}^1, \delta, 0, \{0\})$ *with the transition function*

$$\begin{aligned} \delta((0),0) &:= 0 \\ \delta((1),0) &:= 1 \\ \delta(a,1) &:= 1 \text{ for } a \in \{0,1\}^1. \end{aligned}$$

2. $\mathrm{Aut}_{A^1}(x \approx 1) \cong (\{0,1,2\}, \{0,1\}^1, \delta, 0, \{1\})$ *with the transition function*

$$\begin{aligned} \delta((0),0) &:= 2 \\ \delta((1),0) &:= 1 \\ \delta((0),1) &:= 1 \\ \delta((1),1) &:= 2 \\ \delta(a,2) &:= 2 \text{ for } a \in \{0,1\}^1. \end{aligned}$$

3. $\mathrm{Aut}_{A^2}(x \approx y) \cong (\{0,1\}, \{0,1\}^2, \delta, 0, \{0\})$ *with the transition function*

$$\begin{aligned} \delta((0,0),0) &:= 0 \\ \delta((1,1),0) &:= 0 \\ \delta((0,1),0) &:= 1 \\ \delta((1,0),0) &:= 1 \\ \delta(a,1) &:= 1 \text{ for } a \in \{0,1\}^2. \end{aligned}$$

4. $\mathrm{Aut}_{A^3}(x + y \approx z) \cong (\{0,1,2\}, \{0,1\}^3, \delta, 0, \{0\})$ *with the transition function*

$$\delta(a,0) := \begin{cases} 0 \text{ if } a = (0,0,0), (1,0,1) \text{ or } (0,1,1) \\ 1 \text{ if } a = (1,1,0) \\ 2 \text{ otherwise} \end{cases}$$

$$\delta(a,1) := \begin{cases} 0 \text{ if } a = (0,0,1) \\ 1 \text{ if } a = (1,0,0), (0,1,0) \text{ or } (1,1,1) \\ 2 \text{ otherwise} \end{cases}$$

$$\delta(a,2) := 1 \text{ for } a \in \{0,1\}^3.$$

Proof. Each of the four deterministic finite automata is minimal. Furthermore, the four deterministic finite automata fulfill the property $\mathrm{Sol}_{A^n}(F) = \mathrm{Lang}(\mathrm{Aut}_{A^n}(F))$, which has been proved in lemma 3.2.15. □

Lemma 3.3.3 *Let $A := \{0,1\}$ and let $n \in \mathbb{N}_0$. Let $F, F_1, F_2 \in L_n^{\mathcal{P}}$ be a first-order formula of type $\mathcal{P}$ with n free variables with deterministic finite automata $\mathrm{Aut}_{A^n}(F), \mathrm{Aut}_{A^n}(F_1)$ and $\mathrm{Aut}_{A^n}(F_2)$.*
Then the following holds:

1. $\mathrm{Aut}_{A^n}(F_1 \wedge F_2) \cong \mathrm{Min}\left(\mathrm{Aut}_{A^n}(F_1) \cap \mathrm{Aut}_{A^n}(F_2)\right)$
2. $\mathrm{Aut}_{A^n}(\neg F) \cong \mathrm{Min}\left(\mathrm{Aut}_{A^n}(F)\right)$
3. $\mathrm{Aut}_{A^n}(F_1 \vee F_2) \cong \mathrm{Min}\left(\mathrm{Aut}_{A^n}(F_1) \cup \mathrm{Aut}_{A^n}(F_2)\right)$
4. $\mathrm{Aut}_{A^n}(F_1 \to F_2) \cong \mathrm{Min}\left(\mathrm{Cpl}(\mathrm{Aut}_{A^n}(F_1)) \cup \mathrm{Aut}_{A^n}(F_2)\right)$
5. $\mathrm{Aut}_{A^n}(F_1 \leftrightarrow F_2) \cong \mathrm{Min}\left((\mathrm{Cpl}(\mathrm{Aut}_{A^n}(F_1)) \cap \mathrm{Cpl}(\mathrm{Aut}_{A^n}(F_2))) \cup (\mathrm{Aut}_{A^n}(F_1) \cap \mathrm{Aut}_{A^n}(F_2))\right)$

Let φ_{n+1} be the homomorphism ignoring the $n+1$-th component of each letter:

$$\varphi_{n+1} : A^{n+1} \to A^n$$
$$(a_1, \ldots, a_n, a_{n+1}) \mapsto (a_1, \ldots, a_n).$$

Let $F \in L_{n+1}^{\mathcal{S}}$ be a first-order formula of type $\mathcal{P}$ with $n+1$ free variables with deterministic finite automaton $\mathrm{Aut}_{A^{n+1}}(F)$.

6. $\mathrm{Aut}_{A^n}(\exists x_{n+1} : F) \cong \mathrm{Min}\left(\overline{\mathrm{Det}(\varphi_{n+1}(\mathrm{Aut}_{A^{n+1}}(F)))}^{\mathbf{o}}\right)$
7. $\mathrm{Aut}_{A^n}(\forall x_{n+1} : F) \cong \mathrm{Min}\left(\mathrm{Cpl}\left(\overline{\mathrm{Det}(\varphi_{n+1}(\mathrm{Cpl}(\mathrm{Aut}_{A^{n+1}}(F))))}^{\mathbf{o}}\right)\right)$

Proof. Obviously each of the seven deterministic finite automata is minimal. Furthermore, the equality of the solutions of the first order formulas and the language of the automata follows with lemma 3.2.17. □

Shortcut 1 Let $n \in \mathbb{N}, n \geq 2$. There are several ways to define the formula $x \approx n$ and hence to create the deterministic finite automaton recursively:

1. $(\forall y\ n-1 \leq y \leq x \Rightarrow y = n) \wedge n \leq x$
2. $\exists y : y + 1 = x \wedge y = n$.

But instead of creating the deterministic finite automaton for the first-order formula $x \approx n$ recursively, it is more efficient to turn the base 2 representation of n into a word and create the deterministic finite automaton accepting this word with leading zeros.

Lemma 3.3.4 *Let $n \in \mathbb{N}, n \geq 2$. Let $[r_0 \ldots r_m]_2$ be the base 2 representation of n (with $m = \lfloor \log_2(n) \rfloor + 1$). The deterministic finite automaton*

$$M := (\{0, \ldots, m+1\}, \{0,1\}^1, \delta, 0, \{m\}) \text{ where}$$
$$\delta(a,q) := \begin{cases} q+1 & \text{if } a = (r_q) \text{ and } q < m \\ m & \text{if } a = (0) \text{ and } q = m \\ m+1 & \text{otherwise} \end{cases}$$

recognizes the language

$$L := \{w \in (A^1)^* \mid N(w) = n\}.$$

Then $\mathrm{Sol}_{A^1}(x \approx n) = L$ and $\mathrm{Aut}_{A^1}(x \approx n) \cong M$ holds.

Proof. The base 2 representation with length m needs $m+1$ states in order to label the each of the m transitions between these states with $(r_0), \ldots, (r_m)$, starting in the initial state 0 and ending in the final state m (first line of the transition function δ). The second line of the transition function δ ensures that the word with any leading zero is accepted. Any other transition will end in the sink state $m+1$. Since $r_m \neq 0$, there can't be any deterministic finite automaton with less states accepting the word $(r_0)\ldots(r_m)$ (while still recognizing the same language), hence the deterministic finite automaton M is minimal. □

Shortcut 2 The deterministic finite automaton of a linear equation with natural number coefficients, allowing constant multiplications, can be created iteratively by intersecting automatons of the formulas $x + y \approx z$ and $x \approx y$ with the accordingly substituted variables. But the deterministic finite automaton for the linear equation

$$\mathrm{Aut}_{A^n}\big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot x_j\big)$$

can also be also created directly. An addition step is true for a triple $a_1, \ldots a_n \in \{0,1\}$, the current carry q and the future carry p if the following holds:

$$\sum_{i=1}^{n_1} s_i \cdot a_i = \sum_{j=1}^{n_2} t_i \cdot a_{j+n_1} + q + 2(p-q).$$

Hence for a given letter $a = (a_1, \ldots, a_n)$ and a current carry q, which will identify the q-th state, we can compute the future carry p. The accepted words are those who end in the final state 0, i.e. carry 0.

Lemma 3.3.5 *Let $A := \{0,1\}$ and let $N, n_1, n_2 \in \mathbb{N}_0$ such that $n = n_1 + n_2 > 0$.*
Let $x_1, \ldots, x_{n_1}$, $y_1, \ldots, y_{n_2}$ be variables and let $s_1, \ldots, s_{n_1}, t_1, \ldots, t_{n_2} \in \mathbb{N}$. Define

$$S_1 := \sum_{i=1}^{n_1} s_i, \qquad S_2 := \sum_{j=1}^{n_2} t_j, \qquad S := S_1 + S_2 \qquad \text{and} \qquad Q := \{S_2 + 1, \ldots, S_1 - 1\}.$$

1. *For $S_1 = 0$ or $S_2 = 0$ the deterministic finite automaton M_0 over the alphabet A^n is defined as*

$$M_0 := (\{0,1\}, A^n, \delta, 0, \{0\}) \text{ where}$$
$$\delta(a,q) := \begin{cases} 0 & \text{if } a = \mathbf{0} \text{ and } q = 0 \\ 1 & \text{otherwise} \end{cases}$$

and recognizes the language containing only the zero words, i.e.

$$L_0 := \{w \in (A^n)^* \mid N(w^{(i)}) = 0 \text{ for } 1 \leq i \leq n\}.$$

Then

$$\mathrm{Sol}_{A^n}\Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot y_j\Big) = L_0 \text{ and}$$
$$\mathrm{Aut}_{A^n}\Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot y_j\Big) \cong M_0.$$

2. *Otherwise, let*

$$T : A^n \times Q \to \mathbb{Q}$$
$$(a, q) \mapsto \Big(\sum_{i=1}^{n_1} s_i \cdot a_i - \sum_{j=1}^{n_2} t_j \cdot a_{j+n_1} + q \Big)/2$$

The deterministic finite automaton M over the alphabet A^n is defined as

$$M := (Q \cup \{S\}, A^n, \delta, 0, \{0\}) \text{ where}$$
$$\delta(a, q) := \begin{cases} T(a, q) & \text{if } T(a, q) \in Q \text{ and } q < S \\ S & \text{otherwise} \end{cases}$$

and recognizes the language

$$L := \big\{ w \in (A^n)^* \mid \sum_{i=1}^{n_1} s_i \cdot N(w^{(i)}) = \sum_{j=1}^{n_2} t_j \cdot N(w^{(j+n_1)}) \big\}.$$

Then

$$\mathrm{Sol}_{A^n} \Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot y_j \Big) = L \text{ and}$$
$$\mathrm{Aut}_{A^n} \Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot y_j \Big) \cong M.$$

Proof. 1. Assume $S_2 = 0$. Hence $n_2 = 0$ and $n = n_1 > 0$. The only solution to the equation is $x_i = 0$ for $1 \leq i \leq n_1$. The deterministic finite automaton M accepts only the words consisting of $\mathbf{0}$ and therefore recognizes the language

$$\mathrm{Sol}_{A^n} \Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx 0 \Big).$$

Since M_0 is minimal also

$$\mathrm{Aut}_{A^n} \Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx 0 \Big) \cong M_0.$$

holds. The case $S_1 = 0$ follows similarly.

2. Let $w = w_1 \dots w_m$ be a word accepted by the DFA M for $m \in \mathbb{N}_0$. Hence there exists $r_0, \dots, r_m \in Q$ such that $r_0 = 0$, $r_m \in \{0\}$ and $\delta(w_i, r_{i-1}) = r_i$ for $i \in \{1, \dots, m\}$. The transition for the letter w_1 starts in state 0 (which is the state with carry 0) and goes to state r_1 (the state with carry r_1). Hence it can be seen as the first step of the binary addition. After m steps, simulating all m addition steps, the letter w_m reaches the state $r_m = 0$ and a valid binary addition was performed. So the deterministic finite automaton M recognizes the language L and

$$\mathrm{Sol}_{A^n} \Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot y_j \Big) = L.$$

The deterministic finite automaton M has $S = S_1 + S_2$ states, where

$$\underbrace{S_1 - 1}_{\text{negative carries}} + \underbrace{S_2 - 1}_{\text{positive carries}} + \underbrace{1}_{\text{carry 0}} + \underbrace{1}_{\text{sink state}} = S$$

We assume that there exists a deterministic finite automaton with less than S states recognizing the same language. Since it cannot accept all words, there must be a sink state. Hence this automaton misses a state which can be identified as a carry. Therefore the words which reach this state, i.e. this carry, during the addition cannot be accepted by this deterministic finite automaton.

Therefore M is minimal and

$$\mathrm{Aut}_{A^n}\Big(\sum_{i=1}^{n_1} s_i \cdot x_i \approx \sum_{j=1}^{n_2} t_j \cdot y_j\Big) \cong M.$$

□

Example 3.3.6 With the previous theorem the first-order formula $x + y \approx z$ yields: $S_1 = 0$, $S_2 = 2$, $S = 2$ and $Q = \{0, 1\}$. Then the function $T : A^3 \times \{0, 1\} \to \mathbb{Q}$ and the transition function $\delta : A^3 \times \{0, 1, 2\} \to \{0, 1, 2\}$ are defined as follows:

$T(a, q)$	0	1
$(0, 0, 0)$	0	$-\frac{1}{2}$
$(1, 0, 0)$	$\frac{1}{2}$	1
$(0, 1, 0)$	$\frac{1}{2}$	1
$(1, 1, 0)$	1	$\frac{3}{2}$
$(0, 0, 1)$	$-\frac{1}{2}$	0
$(1, 0, 1)$	0	$\frac{1}{2}$
$(0, 1, 1)$	0	$\frac{1}{2}$
$(1, 1, 1)$	$\frac{1}{2}$	1

$\delta(a, q)$	0	1	2
$(0, 0, 0)$	0	2	2
$(1, 0, 0)$	2	1	2
$(0, 1, 0)$	2	1	2
$(1, 1, 0)$	1	2	2
$(0, 0, 1)$	2	0	2
$(1, 0, 1)$	0	2	2
$(0, 1, 1)$	0	2	2
$(1, 1, 1)$	2	1	2

Hence DFA $\mathrm{Aut}_{A^3}(x + y \approx z) := (\{0, 1, 2\}, \{0, 1\}^3, \delta, 0, \{0\})$ (compare with Lemma 3.3.3).

Extensions Exploiting the alphabet $A := \{0, 1\}$, corresponding to the base 2 representation, we can define the following minimal deterministic finite automaton M:

$$M := (\{0, 1, 2\}, A^2, \delta, 0, \{1\}) \text{ with}$$

$$\delta(a, 0) := \begin{cases} 0 \text{ if } a = (0, 0) \\ 1 \text{ if } a = (1, 1) \\ 2 \text{ otherwise} \end{cases}$$

$$\delta(a, 1) := \begin{cases} 0 \text{ if } a = (0, 0) \text{ or } (1, 0) \\ 2 \text{ otherwise} \end{cases}$$

$$\delta(a, 2) := 2$$

with the language

$$\mathrm{Lang}(M) = \{w \in A^n \mid N(w^{(2)}) \text{ is the greatest power of 2 dividing } N(w^{(1)})\}.$$

Definition 3.3.7 Let $\mathcal{B} := \{0, 1, +, V_k(x)\}$ be a symbol set and let $L^{\mathcal{B}}$ be a first-order language, where 0 and 1 are nullary operations, + is binary operation (addition) and $V_k(x)$ denotes the largest power of k dividing x. Let $\mathbf{M} := (\mathbb{N}, B)$ be a first-order structure of type $\mathcal{B}$.
The *Büchi arithmetic* of base k is the first-order theory of $\mathbf{M}$.

Replacing all occurrences of $\mathcal{P} := \{0, 1, +\}$ with $\mathcal{B} := \{0, 1, +, V_2(x)\}$ in the previous definitions yields the following theorem:

Theorem 3.3.8 *Let $A := \{0, 1\}$ and let $n \in \mathbb{N}_0$. Let $\mathcal{B} := \{0, 1, +, V_2(x)\}$ and let F be a first-order formula of type $\mathcal{B}$ with n free variables.*
Then $\mathrm{Sol}_{A^n}(F)$ is recognizable.

Proof. The proof follows from theorem 3.2.20 and the deterministic finite automaton M from above, since $\mathrm{Aut}_{A^n}(V_2(x) \approx y) \cong M$ and therefore $\mathrm{Sol}_{A^2}(V_2(x) \approx y) = \mathrm{Lang}(M)$. □

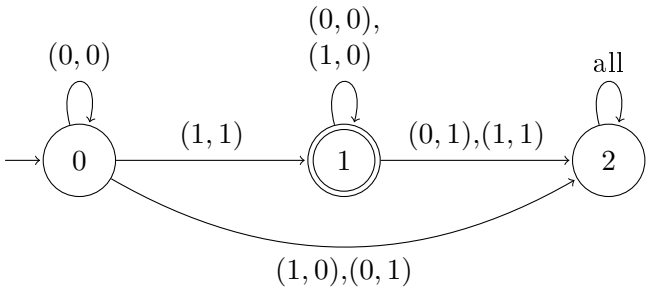


Figure 3.5: The transition diagram of DFA M recognizing $\mathrm{Sol}_{A^2}(V_2(x) \approx y)$.

3.4 Deciding Presburger arithmetic

Theorem 3.4.1 *The* Presburger arithmetic *is decidable.*

Proof. Let $\mathbf{N} := (\mathbb{N}, P)$ be the first-order structure of type $\mathcal{P}$ and let $\mathrm{Th}(\mathbf{N})$ be the Presburger arithmetic, i.e. the first-order formulas $F \in L_0^{\mathcal{P}}$ satisfied by $\mathbf{N}$:

$$\mathrm{Th}(\mathbf{N}) := \{F \in L_0^{\mathcal{P}} \mid \mathbf{N} \models F\}$$

The language $\mathrm{Th}(\mathbf{N})$ is decidable if there exists an algorithm that for any first-order formula $F \in L_0^{\mathcal{P}}$ terminates and returns **True** if $F \in \mathrm{Th}(\mathbf{N})$ holds and otherwise **False**.
Let $F \in L_0^{\mathcal{P}}$. With lemma 3.2.20 the language $\mathrm{Sol}_{A^0}(F)$ is recognizable and $\mathrm{Aut}_{A^0}(F)$ denotes a minimal deterministic finite automaton such that

$$\mathrm{Lang}(\mathrm{Aut}_{A^0}(F)) = \mathrm{Sol}_{A^0}(F).$$

Since $\mathrm{Sol}_{A^0}(F)$ is leading zero complete also the language $\mathrm{Lang}(\mathrm{Aut}_{A^0}(F))$ is leading zero complete and hence $\mathrm{Aut}_{A^0}(F)$ (with lemma 3.2.14). If the deterministic finite automaton $\mathrm{Aut}_{A^0}(F)$ contains a final state, then

$$\mathrm{Lang}(\mathrm{Aut}_{A^0}(F)) = (A^0)^*.$$

So F is interpreted as **True** for $F \in \mathrm{Th}(\mathbf{N})$.
On the other hand, if $\mathrm{Aut}_{A^0}(F)$ doesn't contain a final state then

$$\mathrm{Lang}(\mathrm{Aut}_{A^0}(F)) = \emptyset$$

and F is interpreted as **False** for $F \notin \mathrm{Th}(\mathbf{N})$. □

Corollary 3.4.2 *The* Büchi arithmetic *of base* 2 *is decidable.*

Proof. The language $\mathrm{Sol}_{A^2}(V_2(x) \approx y)$ is recognizable and hence $\mathrm{Aut}_{A^2}(V_2(x) \approx y)$ exists. So the Büchi arithmetic of base 2 is decidable. □

4 Implementation of the GAP package *Predicata*

This chapter contains the manual of the implemented package *Predicata*, written in *GAP – Groups, Algorithms, and Programming* [GAP18], using the data structure from the package *Automata* [DLM11]. The theoretical concepts of the previous chapters are used to create automata from first-order formulas, such that the solution of the formula is equal to the language of the automaton. Furthermore, the package decides if sentences are either true or false.

The main object type `Predicaton` consists of an automaton and a list and represents a first-order formula 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` (4.3.1) 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 section 4.3, especially to start with the examples. The structure of the manual follows the structure of the package, thus the chapter 4.1 and 4.2 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.

4.1 Creating Predicata

4.1.1 Predicaton – an extended finite automaton

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` (4.3.1) 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 ]. >
```

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 ]
```

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
```

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 ]
```

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 ]. >
```

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 ]);;"
```

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 ]
```

4.1.2 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 4.1.3.

Furthermore, note that the following functions can be either called with an `Automaton` or a `Predicaton`.

DisplayAut

▷ `DisplayAut(P)` (function)

The function `DisplayAut` prints the `Automaton` or `Predicaton` *P* (called by `Display` (4.1.1)).

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 ]
```

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 ]
```

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
```

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
```

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"
```

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
```

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 ] ]
```

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
```

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 ] ]
```

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 st
%

         |  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 ]
```

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 ]
```

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 ]
```

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 ]
```

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 ]
```

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 ]
```

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 ]
```

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 ]
```

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 ]
```

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 ]
```

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` (4.1.3).

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 ]
```

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` (4.1.3).

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 ]
```

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
```

4.1.3 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).

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 ]
```

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
```

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` (4.1.2), 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 ] ]);
true
```

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 ] ]
```

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
```

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 *1* 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
```

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 >
```

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 ]
```

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` (4.1.3).

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 ]
```

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 ]*
```

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@)
```

NormalizedLeadingZeroPredicaton

▷ `NormalizedLeadingZeroPredicaton(`*P*`)` (function)

The function `NormalizedLeadingZeroPredicaton` returns the union of `ProductLZeroPredicaton` (4.1.3) and `RightQuotientLZeroPredicaton` (4.1.3) 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 ]*
```

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 ]
```

FormattedPredicaton

▷ `FormattedPredicaton(P)` (function)

The function computes first the `NormalizedLeadingZeroPredicaton` (4.1.3) and then the `MinimalAut` (4.1.2) 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 ]*
```

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
```

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 ]
```

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@
```

NegatedProjectedNegatedPredicaton

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

The function `NegatedProjectedNegatedPredicaton` returns the negated (`NegatedAut` (4.1.2)), projected (`ProjectedPredicaton` (4.1.3) 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 ]
```

IntersectionPredicata

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

The function `IntersectionPredicata` returns the intersection (`IntersectionAut` (4.1.2)) of the `Predicata` of *P1* and *P2* after resizing (`ExpandedPredicaton` (4.1.3)) and sorting (`SortedAlphabetPredicaton` (4.1.3)) 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 ] ]
```

UnionPredicata

▷ `UnionPredicata(`*P*`)` (function)

The function `UnionPredicata` returns union (`UnionAut` (4.1.2)) of the `Predicata` of *P1* and *P2* after resizing (`ExpandedPredicaton` (4.1.3)) and sorting (`SortedAlphabetPredicaton` (4.1.3)) 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 ] ]
```

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 ]
```

PredicatonFromAut

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

The function `PredicatonFromAut` returns the according to *n* resized (`ExpandedPredicaton` (4.1.3)) `Predicaton` of the `Automaton` *A* with the permuted alphabet (`PermutedAlphabetPredicaton` (4.1.3)), 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 ]
```

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
```

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 ]*
```

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 ] ],
  [ [  ] ] ]
```

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 ]);;
gap> r:=PredicatonToRatExp(A);;
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 ] ]
```

4.1.4 Special functions on Predicata

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
```

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:   [ ]
```

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 ]
```

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 ]
```

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 ]
```

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, [ 5, 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
```

AdditionPredicatonNSummands

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

The function `AdditionPredicatonNSummands` calls the function `AdditionPredicatonNSummandsExplicit` (4.1.5) 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` (4.1.5) and `AdditionPredicatonNSummandsRecursive` (4.1.5) 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 ]
```

TimesNPredicaton

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

The function `TimesNPredicaton` returns the `Predicaton` calls the function `TimesNPredicatonExplicit` (4.1.5) 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 ] ]
```

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
```

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 (4.1.4).

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   --  --  --
```

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
```

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
```

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
```

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
```

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, [ 5, 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
```

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, [ 5, 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
```

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, [ 5, 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 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
```

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, [ 5, 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  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
```

4.1.5 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.

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 ]. >
```

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 ]. >
```

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 ]. >
```

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` (4.1.5) 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 ]. >
```

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
```

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 ]. >
```

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 ]
```

4.2 Parsing first-order formulas

4.2.1 PredicataFormula – strings representing first-order formulas

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" ]
```

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` (4.2.1).

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
```

PredicataGrammarVerification

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

The function `PredicataGrammarVerification` checks if the string *S*, with the optional argument `PredicataRepresentation` (4.2.3) *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` (4.3.1)). 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
```

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` (4.2.1)) and returns a `PredicataFormula` (use `PredicataGrammar` (4.3.1) for an overview of the rules). The optional input *P* is explained at `PredicataRepresentation` (4.2.3), however on default the predefined variable `PredicataList` (4.2.3) is used.

Example

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

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
```

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 ) ).
```

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 >
```

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\");"
```

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" ]
```

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" ]
```

PredicataFormulaFormatted

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

The function `PredicataFormulaFormatted` adds missing parenthesis to the `PredicataFormula` *f* for unambiguous parsing in `PredicataFormulaFormattedToTree` (4.2.2).

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 ) ) >
```

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
```

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)).
```

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 ) >
```

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)))\"));"
```

4.2.2 PredicataTree – converting first-order formulas into trees

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: [ "" ] >
```

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
```

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" ].
```

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" ] >
```

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" ]
```

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
```

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"
```

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" ].
```

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", [ ], [ ] ].
```

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" ] ].
```

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", [ ], [ ] ], [ ] ].
```

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" ] ].
```

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" ] >
```

PredicataFormulaFormattedToTree

▷ `PredicataFormulaFormattedToTree(F)` (function)

The function converts a `PredicataFormulaFormatted` (4.2.1) 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" ] ] ] ] >
```

FreeVariablesOfPredicataTree

▷ `FreeVariablesOfPredicataTree(t)` (function)

The function `FreeVariablesOfPredicataTree` returns the free variables of the `PredicataTree` t, which have been carried over from the `PredicataFormula` (4.2.1) and the `PredicataFormulaFormatted` (4.2.1).

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" ]
```

BoundedVariablesOfPredicataTree

▷ `BoundedVariablesOfPredicataTree(t)` (function)

The function `BoundedVariablesOfPredicataTree` returns the bounded variables of the `PredicataTree` *t*, which have been carried over from the `PredicataFormula` (4.2.1) and the `PredicataFormulaFormatted` (4.2.1).

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" ]
```

PredicataTreeToPredicaton

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

The function `PredicataTreeToPredicaton` calls `PredicataTreeToPredicatonRecursive` (4.2.2) to turn a `PredicataTree` (4.2.2) *t* into a `Predicaton` (4.1.1). 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" ].
```

PredicataTreeToPredicatonRecursive

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

The function `PredicataTreeToPredicatonRecursive` is called by `PredicataTreeToPredicaton` (4.2.2) 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 ]
```

PredicataRepresentationOfPredicataTree

▷ `PredicataRepresentationOfPredicataTree(t)` (function)

The function `PredicataRepresentationOfPredicataTree` returns the `PredicataRepresentation` of a `PredicataTree` *t*. For more details see `PredicataRepresentation` (4.2.3).

Example

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

4.2.3 PredicataRepresentation – Predicata assigned with names and arities

This section explains how to assign a name and an arity to a `Predicata` such that it can be reused in a `PredicataFormula` (4.2.1). The idea is to create elements containing the name, arity and the `Predicata` and combining them in a `PredicataRepresentation` (4.2.3). Additionally, there is a predefined variable `PredicataList` (4.2.3), which is called by the `PredicataFormula` on default, trying to simplify these quite lengthy construction.

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` (4.2.1) 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. >
```

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
```

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 ]
```

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. >
```

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 ]))"
```

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"
```

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
```

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 >
```

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. >
```

PredicataRepresentation

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

The function `PredicataRepresentation` creates a collection of elements (`PredicatonRepresentation` (4.2.3)), where the list *l* may contain arbitrary many of them. The `PredicataRepresentation` is an optional input for the function `PredicataFormula` (4.2.1) (On default it uses the predefined variable `PredicataList` (4.2.3)). 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 ] >
```

IsPredicataRepresentation

▷ `IsPredicataRepresentation(P)` (function)

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

Example

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

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 ]
```

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: [ ]. >
```

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 ]\
)))"
```

NamesOfPredicataRepresentation

▷ `NamesOfPredicataRepresentation(`*P*`)` (function)

The function `NamesOfPredicataRepresentation` returns the names of *P*.

Example

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

AritiesOfPredicatonRepresentation

▷ `AritiesOfPredicatonRepresentation(`*P*`)` (function)

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

Example

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

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 > ]
```

ElementOfPredicataRepresentation

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

The function `ElementOfPredicataRepresentation` returns the *i*-th element as a `PredicatonRepresentation` (4.2.3).

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. >
```

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" ]. >
```

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" ]. >
```

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" ]. >
```

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" ]. >
```

PredicataList

▷ `PredicataList` (global variable)

The variable `PredicataList` is a `PredicataRepresentation` (4.2.3) which is called on default by the `PredicataFormula` (4.2.1). Together with the functions `AddToPredicataList` (4.2.3) and `RemoveFromPredicataList` (4.2.3) 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 ] >
```

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 ] >
```

ClearPredicataList

▷ `ClearPredicataList()` (function)

The function `ClearPredicataList` clears the `PredicataList`.

Example

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

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: [ ]. >
```

4.2.4 Converting PredicataFormulas via PredicataTrees into Predicata

PredicataFormulaToPredicaton

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

The function `PredicataFormulaToPredicaton` takes a `PredicataFormula` (4.2.1) *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` (4.3.1). 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 ]. >
```

StringToPredicaton

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

The function `StringToPredicaton` is the simpler version of `PredicataFormulaToPredicaton` (4.2.4), 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" ].

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

4.3 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].

4.3.1 Creating Predicata from first-order formulas

PredicataGrammar

▷ `PredicataGrammar()` (function)

The function `PredicataGrammar` returns the accepted grammar which is allowed as an input for `PredicataFormula` (4.2.1).

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
```

PredicataPredefinedPredicates

▷ `PredicataPredefinedPredicates()` (function)

The function `PredicataPredefinedPredicates()` returns the predefined predicates which are allowed as an input for `PredicataFormula` (4.2.1).

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
```

Predicaton (PredicataFormula)

▷ `Predicaton(f)` (method)

The method `Predicaton` with a `PredicataFormula` *f* as an argument calls `PredicataFormulaToPredicaton` (4.2.4) 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 ]. >
```

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` (4.2.4) 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 ]. >
```

Predicaton (String)

▷ `Predicaton(`*S*`)` (method)

The method `Predicaton` with a `String` *S* as an argument calls `StringToPredicaton` (4.2.4) 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 ]. >
```

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` (4.2.4) 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 ]. >
```

VariableListOfPredicaton

▷ `VariableListOfPredicaton(P)` (function)

The function `VariableListOfPredicaton` returns the variable list of a `Predicaton` *P* containing variable strings (see `PredicataIsStringType` (4.2.1)). 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" ].
 Output:
< Predicaton: deterministic finite automaton on 4 letters with 4 states
and the variable position list [ 1, 2 ]. >
gap> VariableListOfPredicaton(P);
[ "u3", "z5" ]
```

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` (4.3.1) 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" ].
```

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` (4.3.1) 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` (4.1.3) and `SetVariablePositionListOfPredicaton` (4.1.3).

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" ].

 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" ].
```

VariableAdjustedPredicata

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

The function `VariableAdjustedPredicata` does the same as `VariableAdjustedPredicaton` (4.3.1) 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" ]
```

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 ]. >
```

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" ].
```

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 (4.3.1)).

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" ].
```

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" ].
```

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" ].
 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" ].
```

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` (4.3.1)).

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" ].
```

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` (4.3.1)).

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 ]
```

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 ]. >
```

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 ]. >
```

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 ]. >
```

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 ] ]
```

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
```

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
```

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.

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 ] ]
```

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.

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 ] ]
```

4.3.2 Examples

Example 1: Getting familiar

The following example introduces the two ways of getting a `Predicaton`, either created from a first-order formula (see `PredicataGrammar` (4.3.1)), 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();
```

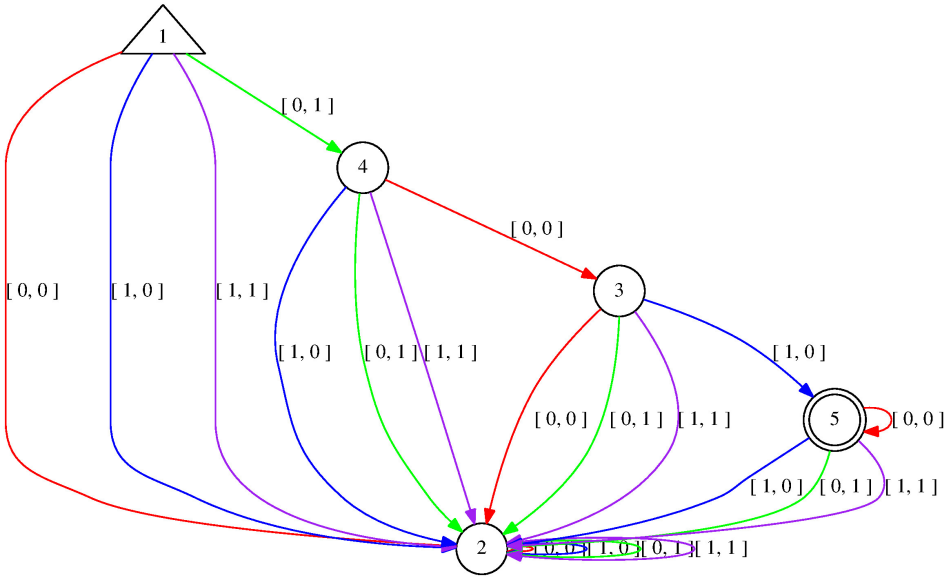


Figure 4.1: A minimal DFA recognizing $x = 4$ (x corresponding to the first component of each letter) and $y = 1$ (y corresponding to the second component).

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 ]. >
```

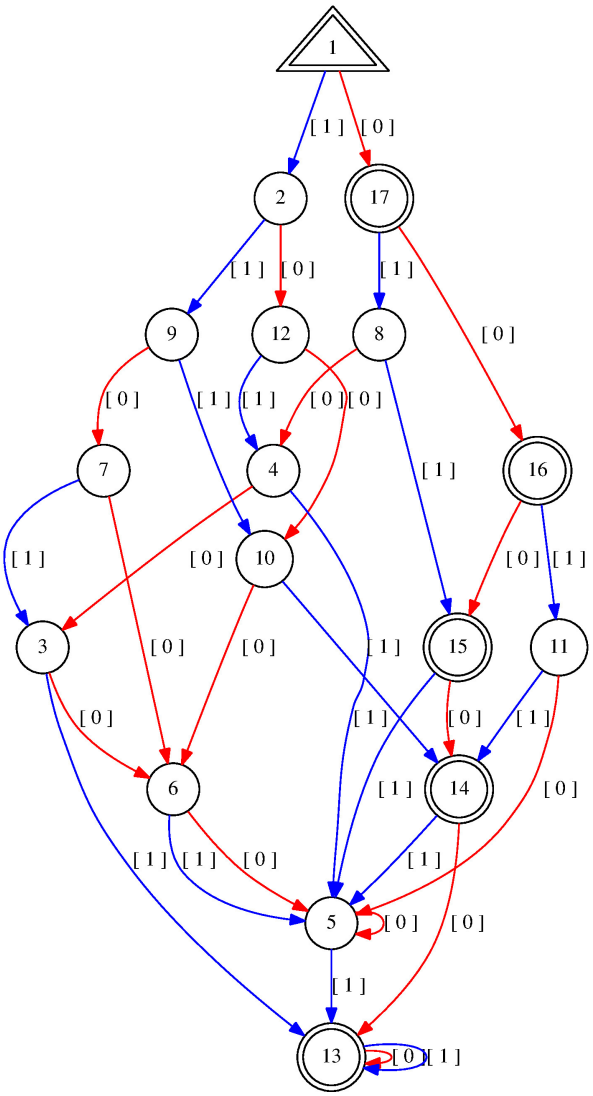


Figure 4.2: A minimal DFA recognizing the numbers which can be purchased by the formula of `A`.

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 handwaving.
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` (4.1.3)), 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);
```

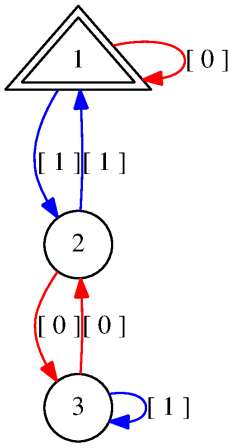


Figure 4.3: A minimal DFA recognizing the numbers divisible by 3.

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
```

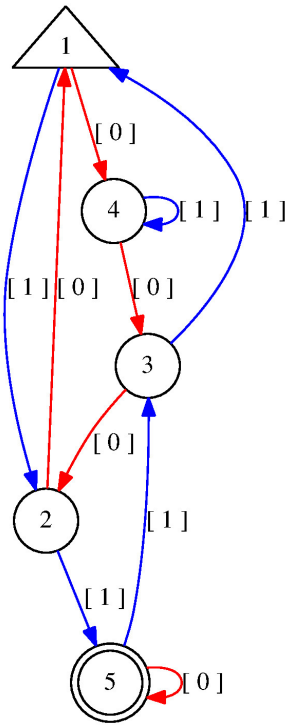


Figure 4.4: A minimal DFA recognizing the solutions of the linear congruence `A`.

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 ] ]
```

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 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 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 [ ]. >
```

Index

Bibliography

[BRT] Basis Representation Theorem. `https://proofwiki.org/wiki/Basis_Representation_Theorem`. Accessed: 2018-07-01.

[Büc60] J. Richard Büchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6*, pages 66–92, 1960.

[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. https://www.graphviz.org/.

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

[EFT07] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Springer-Verlag, Berlin, Heidelberg, 5th edition, 2007.

[GAP18] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.9.1*, 2018.

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

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

[Lin12] Peter Linz. *An Introduction to Formal Languages and Automata, Fifth Edition*. Jones and Bartlett Publishers, Inc., USA, 5th edition, 2012.

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

[Pre29] Mojżesz 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*, pages 92–101. Warszawa, 1929.

[RS59] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

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

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, Boston, MA, USA, 3rd edition, 2013.

[Sta84] Ryan 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.